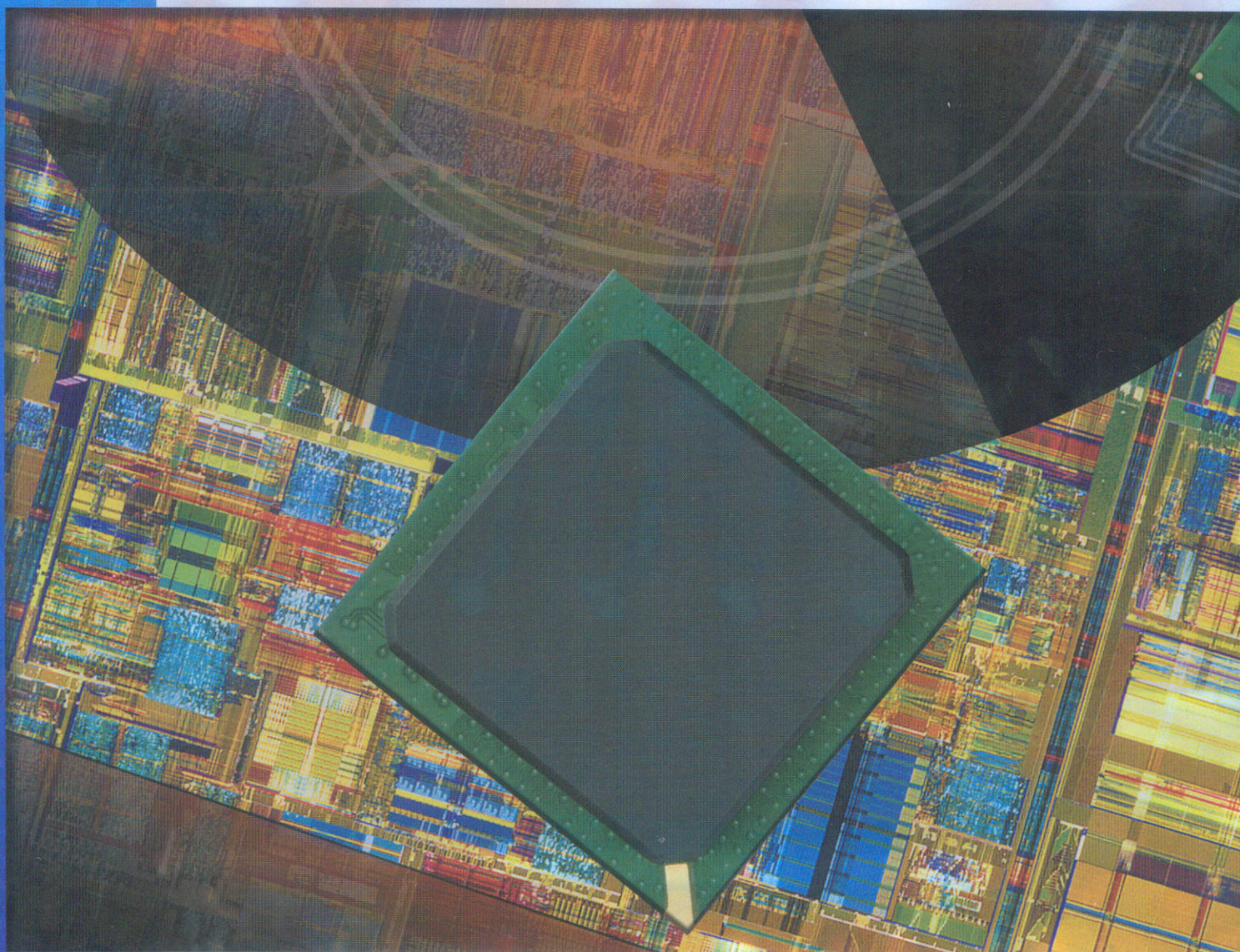
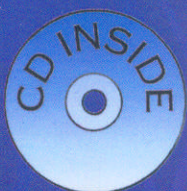


# **The Intel® Microprocessor Family: Hardware and Software Principles and Applications**



James L. Antonakos





# **THE INTEL MICROPROCESSOR FAMILY**



---

# THE INTEL MICROPROCESSOR FAMILY

---

## Hardware and Software Principles and Applications

---

**JAMES L. ANTONAKOS**  
Broome Community College

© Jerome Valencia

*Merencilla*  
Engr. Niño E. Merencilla  
3/11/07

**THOMSON**  
  
**DELMAR LEARNING**

---

Australia • Brazil • Canada • Mexico • Singapore • Spain • United Kingdom • United States



---

# PREFACE

---

The rapid advancement of microprocessors into our everyday affairs has both simplified and complicated our lives. Whether we use a computer in our everyday job, or come in contact with one elsewhere, most of us have used a computer at one time or another. Most people know that a microprocessor is lurking somewhere inside the machinery, but what a microprocessor is, and what it does, remains a mystery.

---

## PURPOSES OF THE BOOK

This book is intended to help remove the mystery concerning the Intel 80x86 microprocessor family through detailed coverage of its hardware and software and examples of many different applications. Some of the more elaborate applications are visible to us. A large collection of personal computers use 80x86-based architecture, as do some popular commercial video games, electronic engravers, and speech recognition systems. Industry and the government have adopted the 80x86 for many commercial and military applications as well.

The book is intended for 2- or 4-year electrical engineering, engineering technology, and computer science students. Professional people, such as engineers and technicians, will also find it a handy reference. The material is intended for a one-semester course in microprocessors. Prior knowledge of digital electronics, including combinational and sequential logic, decoders, memories, Boolean algebra, and operations on binary numbers, is helpful. This presumes knowledge of standard computer-related terms, such as RAM, EPROM, TTL, and so forth. Appendix B is included as a reference on binary numbers, and arithmetic and logic functions for those students who would like a quick review.



---

## FROM THE 8088 UP

Keep in mind that many of the concepts developed for the 8088 apply to the entire 80x86\* family (via real-mode on the advanced processors). It is not the intention of this book to show the reader how to design a 512 MB motherboard for a Pentium III CPU or write protected-mode assembly language code. Instead, the theory and details of hardware interfacing are presented, along with design techniques, for the 8088 and 8086 CPUs, with the techniques easily extended to more advanced designs. For example, when designing a memory address decoder, the same decoding methods that work with 256 MB memory systems also work with smaller 512 KB memory systems. Because real-mode is available on every Intel 80x86 processor, the majority of the book is spent covering assembly language and software applications.

The 8088 and 8086 CPUs are still widely available as 40-pin DIPs (and even as FPGA software cores for programmable devices) and there continue to be existing and new applications that can be handled by these processors, without the need for the power and speed of the more advanced 32-bit architectures. The 16-bit 8088-based single-board computer system presented in Chapter 12 is still a challenging exercise for the students and a lot easier to construct than any 32-bit 386, 486, or Pentium system could be.

---

## OUTLINE OF COVERAGE

For those individuals who have no prior knowledge of microprocessors, **Chapter 1, Microprocessor-Based Systems**, is a good introduction to the microprocessor, how it functions internally, and how it is used in a small system. Chapter 1 is a study of the overall operation of a microprocessor-based system.

**Chapter 2, An Introduction to the 80x86 Microprocessor Family**, highlights the main features of the 80x86. Data types, addressing modes, and instructions are surveyed. All processors in the 80x86 family are examined as well.

**Chapter 3, 80x86 Instructions, Part 1**, and **Chapter 4, 80x86 Instructions, Part 2**, introduce the entire real-mode instruction set of the 80x86. These instructions include data transfer, string, arithmetic, logical, bit manipulation, program transfer, and processor control instructions. Addressing modes, flags, and the structure of a source file are also covered. Over ninety examples are provided to help the student grasp the material. These two chapters are not intended to be read from beginning to end, but instead given a detailed scan. When new instructions are encountered in examples, Chapters 3 and 4 should be consulted, and the sections containing the new instructions read carefully.

**Chapter 5, Interrupt Processing**, covers the basic sequence of an interrupt, as well as multiple interrupts, special interrupts, and interrupt service routines. The instructor may choose to cover this chapter after Chapter 6 to get right to the programming examples.

\*The term "80x86" is used throughout the text to refer to the entire family of compatible Intel processors, the 8088, 8086, 80286, 80386, 80486, and all the Pentium's. Please think of "80x86" as a single machine as well, one processor having many capabilities.



**Chapter 6, An Introduction to Programming the 80x86**, contains the first real programming efforts. Numerous programming examples show how the 80x86 performs routine functions involving binary and BCD mathematics, string operations, data-table manipulation, and number conversions.

**Chapter 7, Advanced Programming Applications**, introduces the student to many advanced concepts, such as linking multiple object files, instruction execution time, interrupt handling, memory management, math coprocessor programming, and macro usage. Assembly language in the Windows and Linux environment is also examined. Fifteen more programming applications are included to support the new concepts.

The hardware operation of the 8088 is covered in **Chapter 8, Hardware Details of the 8088**. All CPU pins are discussed, as are timing diagrams, the difference between min-mode and maxmode operation, the Personal Computer Bus Standards, and two chips that are essential to 8088-based systems: the 8284 clock generator and the 8288 bus controller.

**Chapter 9, Memory System Design**, covers the details needed to design an operational RAM and EPROM-based memory system for the 8088. Static and dynamic RAMs, EPROMs, DMA, and full and partial address decoding are covered.

The I/O system is covered in **Chapter 10, I/O System Design**. In this chapter the difference between the processor's memory space and port space is covered, as are the techniques needed to design port address decoders. Two 8085-based peripherals are covered: the 8255<sup>TM</sup> parallel interface adapter and the 8251 UART<sup>TM</sup>, with 8088 interfacing and programming examples provided.

**Interfacing with the 80x86** is the subject of **Chapter 11**. In this chapter, three peripherals designed to interface with the 80x86 are examined. These peripherals implement interval timing, interrupt control, and floating-point operations. Programming and interfacing are discussed, with specific examples. Several examples are included that show how interfacing is performed with the Personal Computer.

Many textbooks rarely cover hardware and software with an equal amount of detail. This book was written to give equal treatment to both, culminating in a practical exercise: *building and programming your own single-board computer!* **Chapter 12, Building a Working 8088 System**, is included to give students a chance to design, build, and program their own 8088-based computers. The system contains 8KB of EPROM, 8KB of RAM, a serial I/O device, a parallel I/O device, and 8-bit D/A and A/D converters. Future memory expansion is built in. The hardware is designed first, followed by design of the software monitor program.

Some books choose to explain the operation of a commercial system, such as the SDK-86<sup>TM</sup>. This approach is certainly worthwhile, but does not give the student the added advantage of knowing *why* certain designs were used. The hardware and software designs in Chapter 12 are sprinkled with many questions, which are used to guide the design toward its final goal.

The single-board computer presented in Chapter 12 can be easily wire-wrapped in a short period of time (some students have constructed a working computer in seven days), directly from the schematics provided in the chapter. It is reasonable to say that most students can build a working system in one semester.

The hardware operation of the first Pentium CPU is covered in **Chapter 13, Hardware Details of the Pentium**. Descriptions of each CPU pin are included, as are explanations of the various methods employed by the Pentium to access data over its buses. The operation of the Pentium's superscaler architecture, internal pipelining, branch prediction, instruction and data caches, and floating-point unit are discussed.



**Chapter 14, Protected-Mode Operation**, presents the details associated with protected-mode operation. The virtual memory techniques made possible by the use of segments and paging are explained. Important issues such as protection, exceptions, multitasking, and input/output are also discussed. Virtual-8086 mode is also covered.

Finally, **Chapter 15, The Pentium II and Beyond**, takes a guided tour of the architectural changes and improvements in the Pentium line, beginning with the Pentium II, and ending with a look into Intel's future plans for their microprocessor technology. Mobile processors are also examined, with emphasis placed on their special need for low power.

---

## USES OF THE BOOK

Due to the information presented, some chapters are much longer than others. Even so, it is possible to cover certain sections of selected chapters out of sequence, or to pick and choose sections from various chapters. Chapters 3 and 4 could be covered in this way, with emphasis placed on additional addressing modes or groups of instructions at a rate deemed appropriate by the instructor. Some instructors may wish to cover hardware (Chapters 8 through 12), before programming (Chapters 3 through 7). There is no reason this cannot be done.

To aid the instructor, answers to selected odd-numbered end-of-chapter study questions are included in the text and are also provided in a detailed solutions manual. The solutions manual is designed in such a way that solutions to all odd-numbered questions are grouped together, followed by solutions to all even-numbered questions. This allows the instructor to release selected odd-only or even-only answers to students, while retaining others for testing purposes.

In summary, over 250 illustrations and 70 different applications are used to give the student sufficient exposure to the 80x86. The added benefit of Chapter 12, where a working system is developed, makes this book an ideal choice for a student wishing to learn about microprocessors. The material in Chapters 13 through 15 lays the groundwork for more advanced hardware and software development. Furthermore, even though this book deals only with the 80x86 family, the serious microprocessor student should also be exposed to other CPUs as well. But to try to cover two or more different microprocessors in one text does not do either microprocessor justice. For this reason, all attention is paid to the 80x86 family and not to other CPUs.

---

## THE COMPANION CD

The CD included with the book contains all of the source files presented in the book. The files are stored in separate directories related to their specific chapters. In addition, the object code library NUMOUT.LIB from Chapter 7 and the various binary and executable files related to the single-board computer from Chapter 12 are also included. Many useful data sheets and references are provided in PDF format, and there are also fifteen software laboratory experiments to challenge the student.

---

## ACKNOWLEDGMENTS

I would like to thank my editor, Michelle Ruelos Cannistraci for her help while I was putting this book together. In addition, I would like to thank my copyeditor, Libby Larson, as well as Benjamin Gleeksman who managed the book through production. I especially wish to thank Steve Helba, who offered the opportunity to give this book a new life.

The following individuals provided many useful comments and I am grateful for their advice:

Lance Crimm, Southern Polytechnic State University, Marietta, GA

Sang Lee, DeVry University, Addison, IL

Faramarz Mortezaie, DeVry University, Fremont, CA

Max Rabiee, University of Cincinnati, Cincinnati, OH

Lew Rakocy, DeVry University, Columbus, OH

James L. Antonakos  
antonakos\_j@sunybroome.edu  
[http://www.sunybroome.edu/~antonakos\\_j](http://www.sunybroome.edu/~antonakos_j)



---

# BRIEF CONTENTS

---

<b>PART I:</b>	<b>INTRODUCTION</b>	
	1. MICROPROCESSOR-BASED SYSTEMS	3
	2. AN INTRODUCTION TO THE 80x86 MICROPROCESSOR FAMILY	21
<b>PART II:</b>	<b>SOFTWARE ARCHITECTURE</b>	
	3. 80x86 INSTRUCTIONS PART 1: ADDRESSING MODES, FLAGS, DATA TRANSFER, AND STRING INSTRUCTIONS	53
	4. 80x86 INSTRUCTIONS PART 2: ARITHMETIC, LOGICAL, BIT MANIPULATION, PROGRAM TRANSFER, AND PROCESSOR CONTROL INSTRUCTIONS	99
	5. INTERRUPT PROCESSING	141
<b>PART III:</b>	<b>PROGRAMMING</b>	
	6. AN INTRODUCTION TO PROGRAMMING THE 80x86	165
	7. ADVANCED PROGRAMMING APPLICATIONS	222
<b>PART IV:</b>	<b>HARDWARE ARCHITECTURE</b>	
	8. HARDWARE DETAILS OF THE 8088	287
	9. MEMORY SYSTEM DESIGN	308
	10. I/O SYSTEM DESIGN	337
	11. INTERFACING WITH THE 80x86	366
	12. BUILDING A WORKING 8088 SYSTEM	415
<b>PART V:</b>	<b>ADVANCED TOPICS</b>	
	13. HARDWARE DETAILS OF THE PENTIUM	457
	14. PROTECTED-MODE OPERATION	489
	15. THE PENTIUM II AND BEYOND	522
	APPENDIXES	540
	SOLUTIONS AND ANSWERS TO SELECTED ODD-NUMBERED STUDY QUESTIONS	591
	INDEX	605

---

# CONTENTS

---

## **PART I: INTRODUCTION**

<b>1. MICROPROCESSOR-BASED SYSTEMS</b>	<b>3</b>
1.1 Introduction	3
1.2 Evolution of Microprocessors	4
1.3 System Block Diagram	5
1.4 Microprocessor Operation	8
1.5 Hardware/Software Requirements	10
1.6 The Personal Computer	11
1.7 Developing Software for the Personal Computer	14
1.8 Troubleshooting Techniques	18
Summary	18
Study Questions	19
 <b>2. AN INTRODUCTION TO THE 80x86 MICROPROCESSOR FAMILY</b>	 <b>21</b>
2.1 Introduction	22
2.2 Real-Mode and Protected-Mode Operation	22
2.3 The Software Model of the 80x86 Family	23
2.4 Processor Registers	24
2.5 Data Organization	27
2.6 Instruction Types	30
2.7 Addressing Modes	35
2.8 Interrupts	40
2.9 The 8086: The First 80x86 Machine	41
2.10 A Summary of the 80286	43
2.11 A Summary of the 80386	44
2.12 A Summary of the 80486	45
2.13 A Summary of the Pentium	46
2.14 Troubleshooting Techniques	48
Summary	48
Study Questions	49



**PART II: SOFTWARE ARCHITECTURE**

- |   |                |
|---|----------------|
| <b>3. 80x86 INSTRUCTIONS, PART 1: ADDRESSING MODES, FLAGS, DATA TRANSFER, AND STRING INSTRUCTIONS</b>                                 | <b>53</b>      |
| 3.1 Introduction  | 53             |
| 3.2 Assembly Language Programming   | 54             |
| 3.3 Instruction Types   | 59             |
| 3.4 Addressing Modes  | 60             |
| 3.5 The Processor Flags (Condition Codes)   | 70             |
| 3.6 Data Transfer Instructions  | 74             |
| 3.7 String Instructions   | 89             |
| 3.8 Troubleshooting Techniques  | 96             |
| Summary   | 96             |
| Study Questions   | 96             |
| <br><b>4. 80x86 INSTRUCTIONS, PART 2: ARITHMETIC, LOGICAL, BIT MANIPULATION, PROGRAM TRANSFER, AND PROCESSOR CONTROL INSTRUCTIONS</b> | <br><b>99</b>  |
| 4.1 Introduction  | 99             |
| 4.2 Arithmetic Instructions   | 100            |
| 4.3 Logical Instructions  | 110            |
| 4.4 Bit Manipulation Instructions   | 116            |
| 4.5 Program Transfer Instructions   | 121            |
| 4.6 Processor Control Instructions  | 134            |
| 4.7 How an Assembler Generates Machine Code   | 135            |
| 4.8 The Beauty of Relocatable Code  | 136            |
| 4.9 Troubleshooting Techniques  | 137            |
| Summary   | 137            |
| Study Questions   | 138            |
| <br><b>5. INTERRUPT PROCESSING</b>  | <br><b>141</b> |
| 5.1 Introduction  | 141            |
| 5.2 Hardware and Software Interrupts  | 142            |
| 5.3 The Interrupt Vector Table  | 143            |
| 5.4 The Interrupt Processing Sequence   | 146            |
| 5.5 Multiple Interrupts   | 147            |
| 5.6 Special Interrupts  | 148            |
| 5.7 Interrupt Service Routines  | 154            |
| 5.8 Troubleshooting Techniques  | 159            |
| Summary   | 159            |
| Study Questions   | 159            |

**PART III: PROGRAMMING**

- |  |            |
|--|------------|
| <b>6. AN INTRODUCTION TO PROGRAMMING THE 80x86</b> | <b>165</b> |
| 6.1 Introduction                                   | 166        |
| 6.2 Tackling a Large Programming Assignment        | 166        |
| 6.3 Writing a Software Tester                      | 172        |
| 6.4 Data Gathering                                 | 174        |
| 6.5 Searching Data Tables                          | 177        |
| 6.6 String Operations                              | 180        |
| 6.7 Sorting  | 185        |

- 6.8 Computational Routines 187
- 6.9 Control Applications 201
- 6.10 Number Conversions 208
- 6.11 Data Structures 211
- 6.12 Troubleshooting Techniques 217
  - Summary 218
  - Study Question 219
  - Additional Programming Exercises 219

## **7. ADVANCED PROGRAMMING APPLICATIONS 222**

- 7.1 Introduction 222
- 7.2 Using the EXTRN and PUBLIC Assembler Directives 223
- 7.3 Using Macros 226
- 7.4 Instruction Execution Times 234
- 7.5 Working with Interrupt Vectors 237
- 7.6 Multitasking 240
- 7.7 Memory Management 245
- 7.8 Using the Mouse 251
- 7.9 Writing a Memory-Resident Program 254
- 7.10 TICTAC: A Game for a Change 260
- 7.11 Protected-Mode Detection 264
- 7.12 Interfacing C with Assembly Language 266
- 7.13 Assembly Language in the Windows Environment 274
- 7.14 Assembly Language in the Linux Environment 277
- 7.15 Troubleshooting Techniques 281
  - Summary 281
  - Study Questions 282
  - Additional Programming Exercises 282

## **PART IV: HARDWARE ARCHITECTURE**

### **8. HARDWARE DETAILS OF THE 8088 287**

- 8.1 Introduction 287
- 8.2 CPU Specifications 288
- 8.3 CPU Pin Descriptions 290
- 8.4 The 8284 Clock Generator 296
- 8.5 The 8288 Bus Controller 298
- 8.6 System Timing Diagrams 300
- 8.7 Early Personal Computer Bus Standards 303
- 8.8 Troubleshooting Techniques 305
  - Summary 305
  - Study Questions 306

### **9. MEMORY SYSTEM DESIGN 308**

- 9.1 Introduction 308
- 9.2 The 8088 Address and Data Buses 309
- 9.3 Bus Buffering 309
- 9.4 Accessing Memory 311
- 9.5 Designing a Memory Address Decoder 313
- 9.6 Partial-Address Decoding 319
- 9.7 Generating Wait States 322

9.8	A Complete RAM/EPROM Memory	323
9.9	Dynamic RAM Interfacing	325
9.10	Direct Memory Access	331
9.11	Memory-Mapped I/O	332
9.12	Troubleshooting Techniques	333
	Summary	334
	Study Questions	334
<b>10.</b>	<b>I/O SYSTEM DESIGN</b>	<b>337</b>
10.1	Introduction	337
10.2	The 8088 Port Addressing Space	338
10.3	Designing a Port Address Decoder	339
10.4	Operation of a Buffered Input Port	342
10.5	Operation of a Latched Output Port	343
10.6	Simple I/O Applications	344
10.7	Parallel Data Transfer: The 8255 PPI	347
10.8	Serial Data Transfer: The 8251 UART	354
10.9	Troubleshooting Techniques	362
	Summary	362
	Study Questions	362
<b>11.</b>	<b>INTERFACING WITH THE 80x86</b>	<b>366</b>
11.1	Introduction	366
11.2	The 8259 Programmable Interrupt Controller	367
11.3	The 8254 Programmable Interval Timer	376
11.4	The Floating-Point Unit (80x87 Coprocessor)	386
11.5	Interfacing with the Personal Computer	399
11.6	Troubleshooting Techniques	411
	Summary	412
	Study Questions	412
<b>12.</b>	<b>BUILDING A WORKING 8088 SYSTEM</b>	<b>415</b>
12.1	Introduction	415
12.2	Minimal System Requirements	416
12.3	Designing the Hardware	416
12.4	The Minimal System Parts List	426
12.5	Construction Tips	426
12.6	Writing the Software Monitor	427
12.7	A Sample Session with the Single-Board Computer	448
12.8	Troubleshooting Techniques	450
	Summary	451
	Study Questions	451
<b>PART V: ADVANCED TOPICS</b>		
<b>13.</b>	<b>HARDWARE DETAILS OF THE PENTIUM</b>	<b>457</b>
13.1	Introduction	457
13.2	CPU Pin Descriptions	458
13.3	RISC Concepts	465
13.4	Bus Operations	468
13.5	The Pentium's Superscalar Architecture	473



13.6	Pipelining	474	
13.7	Branch Prediction	475	
13.8	The Instruction and Data Caches	477	
13.9	The Floating-Point Unit	483	
13.10	Troubleshooting Techniques	485	
	Summary	486	
	Study Questions	486	
<b>14.</b>	<b>PROTECTED-MODE OPERATION</b>		<b>489</b>
14.1	Introduction	489	
14.2	Segmentation	492	
14.3	Paging	496	
14.4	Protection	500	
14.5	Multitasking	502	
14.6	Exceptions and Interrupts	506	
14.7	Input/Output	513	
14.8	Virtual-8086 Mode	514	
14.9	A Protected-Mode Application	516	
14.10	Troubleshooting Techniques	519	
	Summary	520	
	Study Questions	520	
<b>15.</b>	<b>THE PENTIUM II AND BEYOND</b>		<b>522</b>
15.1	Introduction	522	
15.2	The Pentium Pro	523	
15.3	The Pentium II	526	
15.4	The Pentium III	528	
15.5	The Pentium 4	529	
15.6	The Celeron	532	
15.7	The Xeon	532	
15.8	IA-64 Architecture: The Itanium and Itanium 2	533	
15.9	Intel's Mobile CPUs	535	
15.10	Troubleshooting Techniques	537	
	Summary	538	
	Study Questions	538	
	<b>APPENDICES</b>		<b>540</b>
	A. Instruction Execution Times	540	
	B. A Review of Number Systems, Binary Arithmetic, and Logic Functions	548	
	C. Assembler Reference	556	
	D. DEBUG and CodeView Reference	560	
	E. ASCII Character Set	576	
	F. MMX Technology	578	
	<b>SOLUTIONS AND ANSWERS TO SELECTED ODD-NUMBERED STUDY QUESTIONS</b>		<b>591</b>
	<b>INDEX</b>		<b>605</b>

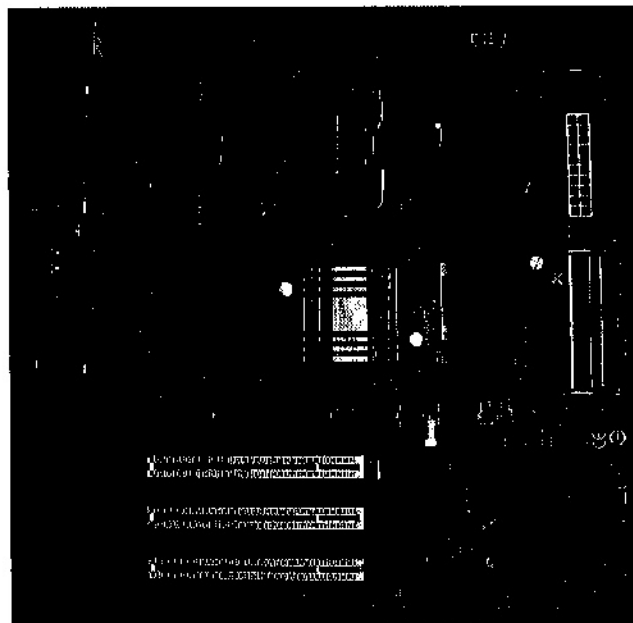
# PART 1

## Introduction

---

- 1** Microprocessor-Based Systems
- 2** An Introduction to the 80x86 Microprocessor Family

A typical Pentium  
motherboard for the  
Personal Computer



---

# CHAPTER 1

---

## Microprocessor-Based Systems

---

### OBJECTIVES

In this chapter you will learn about:

- The block diagram of a microprocessor-based system and the function of each section
- The processing cycle of a microprocessor
- The way software is used to initialize hardware and peripherals
- The history of the microprocessor and of the different generations of computers
- The technique used to create software for the personal computer
- Some typical errors encountered during program development

### KEY TERMS

Assembler	Interrupt	Random access memory
Assembly language	Machine language	Read-only memory
Cache	Math coprocessor	Source file
Central processing unit	Microcontrollers	System bus
Direct memory access	Motherboard	Watchdog monitor
Expansion slots	Nonvolatile memory	

---

## 1.1 INTRODUCTION

What is a microprocessor? You may have heard the term, or even looked inside your personal computer to locate it. A microprocessor is a digital machine capable of executing many different software instructions and controlling a wide variety of electronic devices. The invention of the microprocessor has had a profound impact on many aspects of our lives. Today even the most mundane chores are being accomplished under its supervision—which allows us more time for other productive endeavors. Even a short list of the devices using the microprocessor shows how dependent on it we have become:

1. Pocket calculators
2. Digital watches (some with calculators built in)



3. Automatic tellers (at banks and food stores)
4. Cell phones
5. CD and MP3 players
6. Home security and control devices
7. Realistic video games
8. Talking dolls and other toys
9. VCRs and DVD players
10. Personal computers
11. PDAs
12. Digital cameras

The purpose of this chapter is to show how a microprocessor is used in a small system and to introduce the operation of the personal computer. We will see what types of hardware may be connected to the microprocessor, and why each type is needed. We will also see how software is used to control the hardware, and how that software can be developed.

Section 1.2 shows how the microprocessor has evolved over time, from the initial 4-bit machines to today's 32-bit processors. Section 1.3 covers the block diagram of a typical microprocessor-based system and explains each functional unit. Section 1.4 explains the basic operation of a microprocessor. Section 1.5 discusses the hardware and software requirements of a small microprocessor control system. Section 1.6 brings the material of the first five sections together in a technical description of the personal computer. Section 1.7 shows how software is developed for, and used by, the personal computer. Finally, section 1.8 introduces the first of a series of troubleshooting techniques.

---

## 1.2 EVOLUTION OF MICROPROCESSORS

We have come a long way since the early days of computers, when ENIAC (Electronic Numerical Integrator and Computer) was state of the art and occupied thousands of feet of floor space. Constructed largely of vacuum tubes, it was slow, prone to breakdowns, and performed a limited number of instructions. Even so, ENIAC ushered in what was known as the first generation of electronic computers.

Today, thanks to advances in technology, we have complete computers that fit on a piece of silicon no larger than your fingernail and that far outperform ENIAC.

With the invention of the transistor, computers shrank in size and increased in power, leading to the second generation of computers. Third-generation computers came about with the invention of the integrated circuit, which allowed hundreds of transistors to be packed on a small piece of silicon. The transistors were connected to form logic elements, the basic building blocks of digital computers. With third-generation computers, we again saw a decrease in size and increase in computing power. Machines like the 4004™ and 8008™ by Intel® found some application in simple calculators, but they were limited in computation power and memory addressing capability. When improvements in integrated circuit technology enabled us to place *thousands* of transistors on the same piece of silicon, computers really began to increase in power. This new technology, called large-scale integration (LSI), was even faster than the previous medium- and small-scale integration (MSI and SSI, respectively) technologies, which dealt with only tens or hundreds of transistors on a chip. LSI technology created the fourth generation of computers. An advanced form

of LSI technology, VLSI, meaning very large scale integration, is now being used to increase processing power.

The first microprocessors that became available with third-generation computers had limited instruction sets, slow execution speed, and limited memory addressing, and thus restricted computing abilities. Although they were suitable for use in electronic calculators, they simply did not have the power needed to operate more complex systems, such as guidance systems or scientific applications. Even some of the early fourth-generation microprocessors had limited capabilities because of the lack of addressing modes and instruction types. Eight-bit machines like the 8080™, Z80™, and 6800™ were indeed more advanced than previous microprocessors, but they still did not possess multiply and divide instructions. How frustrating and time consuming to have to write a program to do these operations!

Within the last few decades, microprocessor technology has improved tremendously. Thirty-two-bit processors can now multiply and divide, operate on many different data types (4-, 8-, 16-, 32-bit numbers), and address *billions* of bytes of information. Processors of the 1970s were limited to 64KB of memory for programs and data, a small amount of memory by today's standards.

Each new microprocessor to hit the market boasts a fancier instruction set and faster clock speed, and our need for faster and better processors keeps growing. Taking a different approach, a technology called RISC (Reduced Instruction Set Computer) has gained acceptance. This technology is based on the fact that most microprocessors use only a small portion of their entire instruction set. By designing a machine that uses only the more common types of instructions, processing speed can be increased without the need for a significant advance in integrated circuit technology. The Pentium microprocessor, manufactured by Intel, uses many of the architectural techniques employed by RISC machines.

Why the need for super-fast machines? Consider a microprocessor dedicated to displaying three-dimensional color images on a video screen. Rotating the three-dimensional image around an imaginary axis in real time (in only a few seconds or less) may require millions or even billions of calculations. A slow microprocessor could not do the job.

Eventually we will see fifth-generation computers. The whole artificial intelligence movement is pushing toward that goal, with the desired outcome being the production of a machine that can think. Until then, we will have to make the best use of the technology we have available.

---

## 1.3 SYSTEM BLOCK DIAGRAM

Any microprocessor-based system must, of necessity, have some standard elements such as memory, timing, and input/output (I/O). Depending on the application, other exotic circuitry may be necessary as well. Analog-to-digital (A/D) converters and their counterparts, digital-to-analog (D/A) converters, interval timers, math coprocessors, complex interrupt circuitry, speech synthesizers, and video display controllers are just a few of the special sections that may also be required. Figure 1.1 depicts a block diagram of a system containing some standard circuitry and the functions normally used.

As the figure shows, all components communicate via the **system bus**. The system bus is composed of the processor address, data, and control signals. The **central processing unit (CPU)** is the heart of the system, the master controller of all operations that can be performed. The terms CPU and microprocessor are interchangeable. The CPU executes

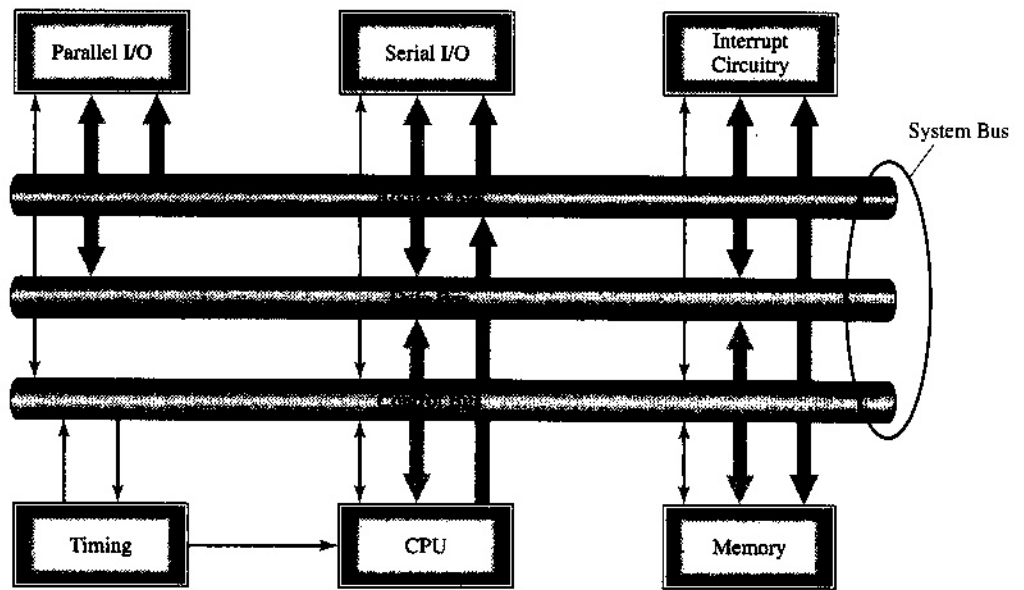


FIGURE 1.1 Standard block diagram of a microprocessor-based system

instructions that are stored in the memory section. For the sake of future expansion, the system bus is commonly made available to the outside world (through a special connector). Devices can then be added easily as the need arises. Commercial systems have predefined buses that accomplish this. All devices on the system bus must communicate with the processor, usually within a tightly controlled period of time. The timing section governs all system timing and thus is responsible for the proper operation of all system hardware. The timing section usually consists of a crystal oscillator and timing circuitry (counters designed to produce the desired frequencies) set up to operate the processor at its specified clock rate. Using a high-frequency crystal oscillator and dividing it down to a lower frequency provides for greater stability.

The CPU section consists of a microprocessor and the associated logic circuitry required to enable the CPU to communicate with the system bus. These logic elements may consist of data and address bus drivers, a bus controller to generate the correct control signals, and possibly a math coprocessor. **Math coprocessors** are actually microprocessors themselves; their instruction set consists mainly of simple instructions for transferring data, and complex instructions for performing a large variety of mathematical operations. Coprocessors perform these operations at very high clock speeds with a great deal of precision (80-bit results are common). In addition to the basic add/subtract/multiply/divide operations, coprocessors are capable of finding square roots, logarithms, a host of trigonometric functions, and more.

The actual microprocessor used depends on the complexity of the task that will be controlled or performed by the system. Simple tasks require nothing more complicated than an 8-bit CPU, which operates on 8 bits of data at a time. A computerized cash register would be a good example of this kind of system. Nothing more complicated than binary coded decimal (BCD) arithmetic—and possibly some record keeping—is needed. But for something as complex as a flight control computer for an aircraft or a digital guidance system for a missile, a more powerful 16- or 32-bit microprocessor must be used.

The memory section usually has two components; **read-only memory (ROM)** and **random access memory (RAM)**. Some systems may be able to work properly without RAM, but all require at least a small amount of ROM. The ROM is included to provide the system with its intelligence, which is ordinarily needed at start-up (power-on) to configure or initialize the peripherals, and sometimes to help recover from a catastrophic system failure (such as an unexpected power failure). Some systems use the ROM program to download the main program into RAM from a larger, external system, such as a personal computer (PC) or a mainframe computer. In any event, provisions are usually made for adding additional ROM as the need arises.

There are three types of RAM. For small systems that do not process a great deal of data, the choice is static RAM. Static RAM is fast and easy to interface, but comes in small sizes (as little as 16 bytes per chip). Larger memory requirements are usually met by using dynamic RAM, a different form of memory that has high density (256K bits per chip or more), but that, unfortunately, requires numerous refreshing cycles to retain the stored data. Even so, dynamic RAM is the choice when large amounts of data must be stored, as in a system gathering seismic data at a volcano or receiving digitized video images from a satellite.

Both static and dynamic RAM lose their information when power is turned off, which could cause a problem in certain situations. Previous solutions involved adding battery backup circuitry to the system to keep the RAMs supplied with power during an outage. But batteries can fail, so a better method was needed; thus the invention of **nonvolatile memory (NVM)**, which is memory that retains its information even when power is turned off. NVM comes in small sizes and therefore is used to store only the most important system variables in the event of a power outage.

Another type of storage media is the floppy disk or hard disk. Both types of disks provide the system with large amounts of storage for programs and data, although the data are accessed at a much slower rate than that of RAM or ROM. Floppy disks and hard disks also require complex hardware and software to operate and are not needed in many control applications.

With a microprocessor used in control applications, there will be times when the system must respond to special external circumstances. For example, a power failure on a computer-controlled assembly line requires immediate attention by the system, which must contain software designed to handle the unexpected event. The event actually *interrupts* the processor from its normal program execution to service the unexpected event. The system software is designed to handle the power-fail interrupt in a certain way and then return to the main program. An **interrupt**, then, is a useful way to grab the processor's attention, get it to perform a special task, and then resume execution from the point where it left off.

Not all types of interrupts are unexpected. Many are used to provide systems with useful features, such as real-time clocks, multitasking capability, and fast I/O operations.

The interrupt circuitry needed from system to system will vary depending on the application. A system used for keeping time has to use only a single interrupt line connected to a timing source. A more complex system, such as an assembly line controller that may need to monitor multiple sensors, switches, and other items, may require many different prioritized interrupts and would therefore need more complex interrupt circuitry.

Some systems may require serial I/O for communication with an operator's console or with a host computer. In serial communication, data is transmitted one bit at a time between devices. In Figure 1.2, we see how a small system might communicate with other devices or systems via serial communication. Although this type of communication is slower than parallel communication, it has the advantage of simplicity: only two wires (for

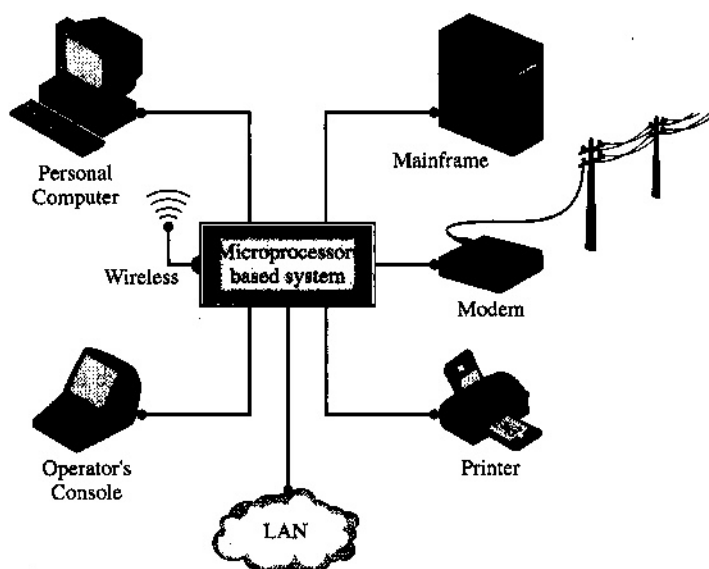


FIGURE 1.2 Serial communication possibilities in a small system

receive and transmit) plus a ground are needed. Serial communication is easily adapted for use in fiber-optic cables. Parallel I/O, on the other hand, requires more lines (at least eight), but has the advantage of being very fast. A special parallel operation called **direct memory access (DMA)** is used to transfer data from a hard disk to a microcomputer's memory. Other uses for parallel I/O involve reading switch information, controlling indicator lights, and transferring data to A/D and D/A converters and other types of parallel devices.

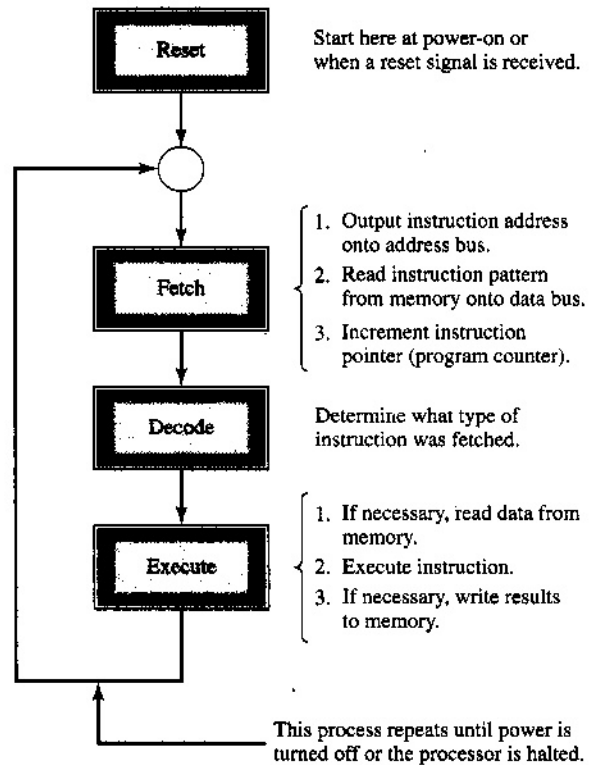
All of these sections have their uses in a microprocessor-based system. Whether they are actually used depends on the designer and the application.

## 1.4 MICROPROCESSOR OPERATION

No matter how complex microprocessors become, they will still follow the same pattern of operations during program execution: endless fetch, decode, and execute cycles. During the fetch cycle, the processor loads an instruction from memory into its internal instruction register. Some advanced microprocessors load more than one instruction into a special buffer to decrease program execution time. The idea is that while the microprocessor is decoding the current instruction, other instructions can be read from memory into the instruction **cache**, a special type of internal high-speed memory. In this fashion, the microprocessor performs two jobs at once, thus saving time.

During the decode cycle, the microprocessor determines what type of instruction has been fetched. Information from this cycle is then passed to the execute cycle. To complete



**FIGURE 1.3** Microprocessor operation

the instruction, the execute cycle may need to read more data from memory or write results to memory. This process is illustrated in Figure 1.3.

While these cycles are proceeding, the microprocessor is also paying attention to other details. If an interrupt signal arrives during execution of an instruction, the processor latches onto the request, holding off on interrupt processing until the current instruction finishes execution. The processor also monitors other signals such as WAIT, HOLD, or READY inputs. These are usually included in the architecture of the microprocessor so that slow devices, such as memories, can communicate with the faster processor without loss of data.

Most microprocessors also include a set of control signals that allows external circuitry to take over the system bus. In a system where multiple processors share the same memory and devices, these types of control signals are necessary to resolve bus contention (two or more processors needing the system bus at the same time). Multiple-processor systems are becoming more popular now as we continue to strive toward faster execution of our programs. Parallel processing is a term often used to describe multiple-processor systems and their associated software.

Special devices called **microcontrollers** are often used in simple control systems because of their many features. Microcontrollers are actually souped-up microprocessors with built-in features such as RAM, ROM, interval timers, parallel I/O ports, and even A/D converters. Microcontrollers are not used for large systems, however, because of their small memory addressing capability and limited processing speed. Unfortunately, we have yet to get everything we want on a single chip!

---

## 1.5 HARDWARE/SOFTWARE REQUIREMENTS

We saw earlier that it is necessary to have at least some ROM in a system to take care of peripheral initialization. What type of initialization is required by the peripherals? The serial device must have its baud rate (communication speed), parity, and number of data and stop bits programmed. Parallel devices must be configured because most of them allow the direction (input or output) of their I/O lines to be programmed in many different ways. It is then necessary to set the direction of these I/O lines when power is first applied. For a system containing a D/A converter, it may be important to output an initial value required by the external hardware. Because we can never assume that correct conditions exist at power-on, the microprocessor is responsible for establishing them.

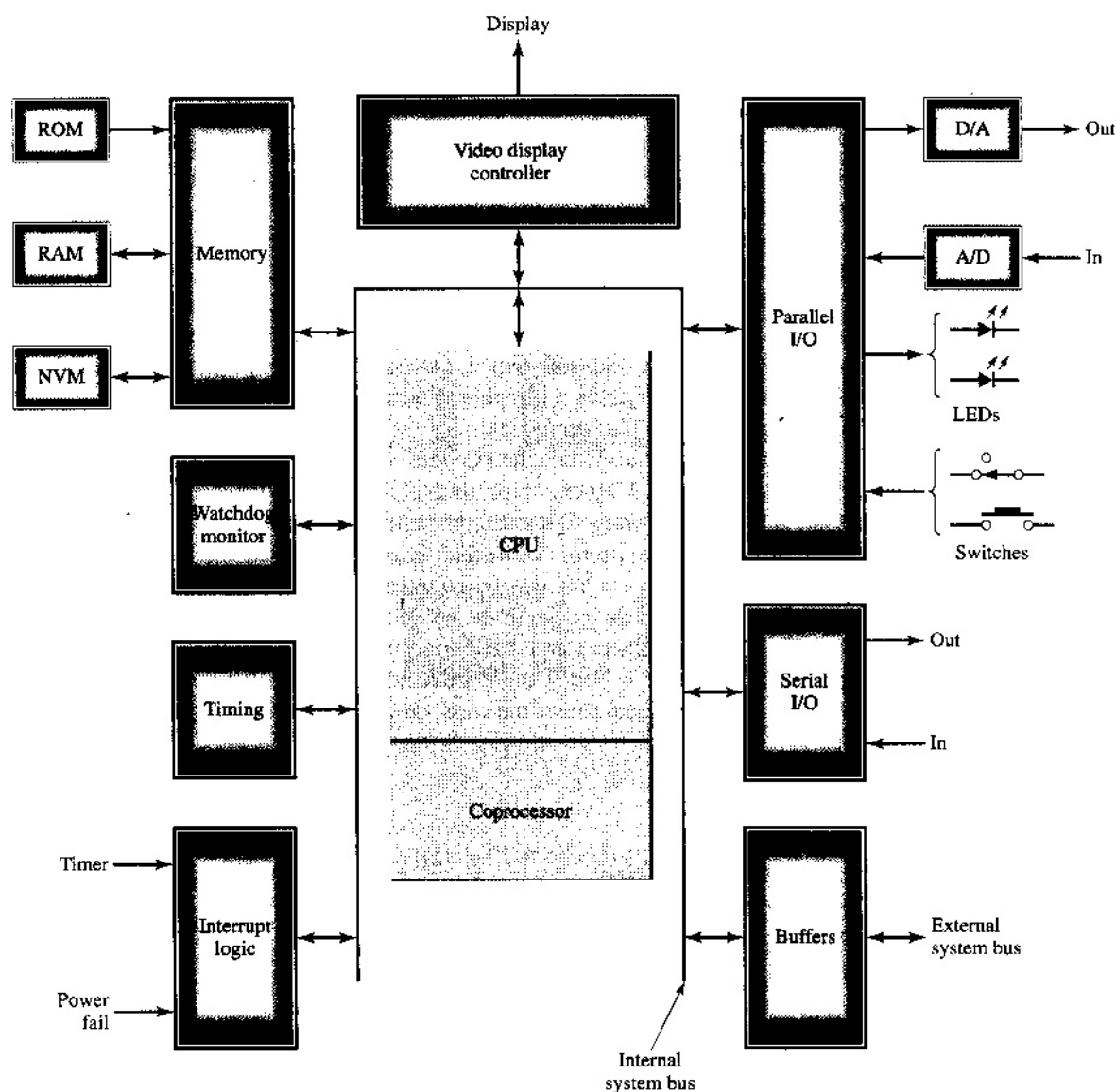
Suppose a certain system contains a video display controller. Start-up software must select the proper screen format and initialize the video memory so that an intelligent picture (possibly a menu) is generated on the screen of the display. If the system uses light-emitting diode (LED) displays or alphanumeric displays for output, they must be properly set as well. High-reliability systems may require that memory be tested at power-on. While this adds to the complexity of the start-up software and the time required for initialization, it is a good practice to follow. Bad memory devices will certainly cause a great deal of trouble if they are not identified.

Other systems may employ a special circuit called a **watchdog monitor**. The circuit operates like this: during normal program execution, the watchdog monitor is disabled. Should the program veer from its proper course, the monitor will automatically reset the system. A simple way to make a watchdog monitor is to use a binary counter, clocked by a known frequency. If the counter is allowed to increment up to a certain value, the processor is automatically reset. The software's job, if it is working correctly, is to make sure that the counter never reaches this count. A few simple logic gates can be used to clear the counter under microprocessor control, possibly whenever the CPU examines a certain memory location.

For flexibility, the system may have been designed to download its main program from a host system. If this is the case, the system software will be responsible for knowing how to communicate with the host and place the new program into the proper memory locations. To guarantee that the correct program is loaded, the software should also perform a running test on the incoming data, requesting the host to retransmit portions of the data whenever it detects an error.

Sometimes preparing for a power-down is as important as doing the start-up initialization. A power supply will quite often supply voltage in the correct operating range for a few milliseconds after the loss of AC power. During these few milliseconds the processor must execute the shutdown code, saving important system data in nonvolatile RAM or doing whatever is necessary for a proper shutdown. If the system data can be preserved, it may be possible to continue normal execution when power is restored.

For systems that will be expanded in the future, the system bus must be made available to the outside world. To protect the internal system hardware, all signals must be properly buffered. This involves using tri-state buffers or similar devices to isolate the internal system bus from the bus available to the external devices. Sometimes optoisolators are used to completely separate the internal system signals from the external ones. The only connection in optoisolators is a beam of light, which makes them ideal when electrical isolation is required.



**FIGURE 1.4** Expanded block diagram of a microprocessor-based system

Figure 1.4 sums up all of these concepts with an expanded block diagram of a microprocessor-based control system. Notice once again that all devices in the system communicate with the CPU via the system bus.

## 1.6 THE PERSONAL COMPUTER

All of the material in this chapter, up to this point, has dealt with general microprocessor-based systems. In this section, we will see how a specific microprocessor-based system, the PC, uses many of the hardware features already described. Although the PC has been

around for many years and has evolved into a powerful machine containing very advanced technology, it began as a much simpler machine constructed around the 8/16-bit Intel 8088 microprocessor. The 8088 came out in the late 1970s and offered a higher level of computing power than the 8-bit processors of the time. When IBM® chose the 8088 for use in its new PC, it paved the way for worldwide acceptance of the new processor. Many companies began copying the architecture of the PC and offered their own compatible 8088-based computer systems. Thus began the PC market.

One reason the PC market grew as fast as it did was due to the usefulness of the features the PC offered. The initial PC contained a keyboard for entering commands and data, a monochrome video display for viewing text and simple graphics, one or two floppy disk drives for storing information and running programs, and a memory large enough for many useful applications. It also came equipped with a software program called DOS, for Disk Operating System, which made it possible to access files on the disk drives and run programs with the use of simple commands.

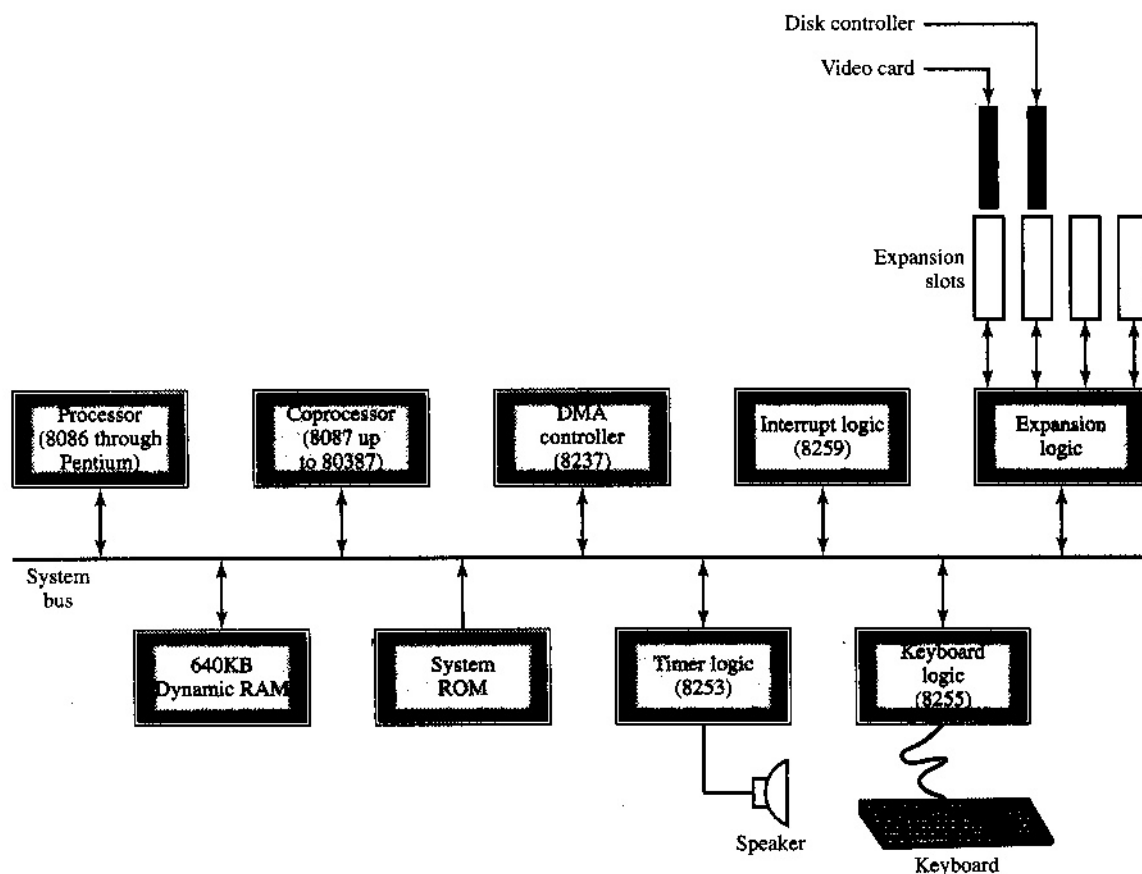
Most of the electronics within the PC were contained on a single printed circuit board called the **motherboard**. Memory chips, timing circuitry, interrupt logic, the 8088 microprocessor, and other hardware all resided on the motherboard. Included were a number of expansion slots, plastic connectors with metal fingers into which other circuit boards could be plugged. The PC's system bus was wired to each expansion slot, so any card plugged into an expansion slot had the power of the entire machine available to it. Expansion cards were used to add new features to the basic machine, such as a color video display, a hard disk, or additional memory. Today, there are hundreds of different expansion cards available. A small sample of them shows the wide variety of hardware applications:

- Modem/Fax
- LAN controller
- Data acquisition
- Sound/speech synthesis
- High-resolution color graphics
- Image processing
- CD-ROM drive
- Hand-held and flatbed scanner
- Serial/parallel I/O

Clearly, with the right number and type of expansion cards, the PC can be configured to do just about anything. For our purposes, we will concentrate on the hardware that comes with a base machine, with a few add-ons, namely the hard disk and color-display cards.

Let us now take a detailed look at the inside of the personal computer. Figure 1.5 is the block diagram for a typical PC motherboard from the early days of the PC. As shown, all communication is through the system bus. The microprocessor may be an 8088 (as found on the original PC), or one of the newer 32-bit processors from Intel, such as the 80386™ or 80486™. A nice feature of the advanced Intel microprocessors is that they all execute programs written for the 8088. So, even if your machine is new, all of the software presented in this textbook will run on it.

If a motherboard contains an 8088 microprocessor, there is usually a socket provided for an additional chip, the 8087 Floating-Point Coprocessor. This device is capable of performing mathematical calculations much faster than the 8088 and is designed to work in parallel with the processor. Motherboards based on the 80486 do not contain this socket because the coprocessor is built into the processor itself.



**FIGURE 1.5** Block diagram of a typical PC motherboard, circa 1980s–1990s

For high-speed data transfers involving memory, the motherboard contains an 8237™ DMA Controller. This device can be easily programmed to move large chunks of data without assistance from the processor.

The PC has many features that require the use of the interrupt system. An 8259™ Programmable Interrupt Controller is included to handle the interrupts generated by the PC's time-of-day clock, keyboard, serial and parallel I/O devices, and disk drives.

The motherboard contains a small amount of ROM as well. This ROM is referred to as system ROM and is used to control the PC when it is turned on. The system ROM is responsible for checking and initializing all peripherals and devices on the motherboard and for starting up the disk drive to load DOS.

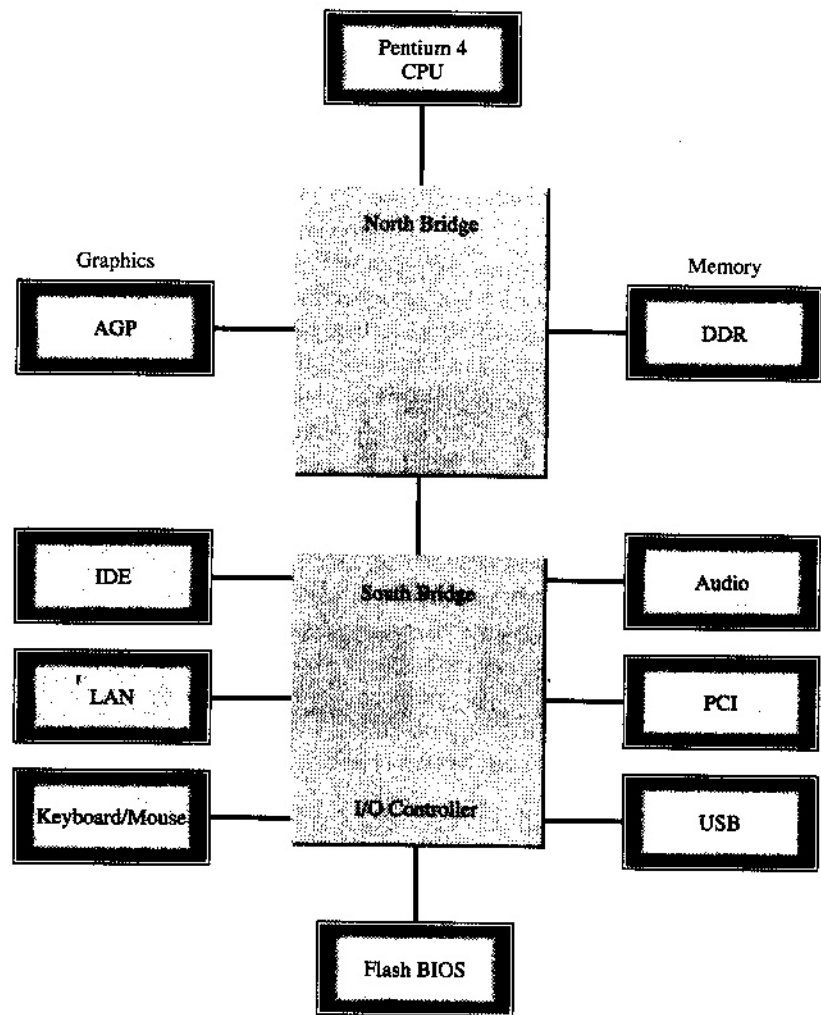
As mentioned before, the PC maintains a time-of-day clock. This clock is a combination of software and hardware. A special timing device, the 8253™ Programmable Interval Timer, is used to generate timing pulses at regular intervals. These pulses interact with the interrupt logic and DOS to simulate the passage of time. The 8253 also controls the PC's speaker. With proper programming, it is possible to make the speaker beep and generate other sounds.

A parallel I/O device, the 8255 Programmable Peripheral Interface, is used to monitor and read the PC's keyboard and motherboard option switches.

Finally, expansion logic is used to drive the system bus signals on the expansion slots. This makes it possible for circuitry on an expansion card to access every device on the



**FIGURE 1.6** Chipset-based motherboard in use today



motherboard. As processors evolved, so did motherboard technology. Figure 1.6 shows the block diagram of a typical motherboard used today. Each microprocessor now has an associated chipset on the motherboard to interface it with the motherboard components. The chipset consists of a pair of powerful integrated circuits that essentially act as communication hubs. The North Bridge controls high-speed data transfers between the CPU, memory, and the graphics system. Slower I/O is accomplished through the South Bridge, an integrated I/O controller that provides all the motherboard I/O needs in a single package.

We do not have to know a great deal about the hardware just described to write programs for the personal computer.

## 1.7 DEVELOPING SOFTWARE FOR THE PERSONAL COMPUTER

To get the most use out of the power of the PC, it is necessary to understand how to control and use the hardware on the motherboard and the software capabilities of the processor. We will explore the software architecture of the 80x86 family in great detail in the

following chapters, first by examining the instruction set and then by looking at programming examples. What we will see is that the 80x86 processors speak a different language than we do.

## Machine Language vs. Assembly Language

Our language is one of words and phrases. The 80x86 language is a string of 1s and 0s. For example, the instruction

```
ADD AX, BX
```

contains a word, **ADD**, that means something to us. Apparently, we are adding **AX** and **BX** together, whatever they are. So, even though we might be unfamiliar with the 80x86 instruction set, the instruction **ADD AX, BX** means something to us.

If we were instead given the binary string

```
0000 0001 1101 1000
```

or the hexadecimal equivalent

```
01 D8
```

and asked its meaning, we might be hard-pressed to come up with anything. We associate more meaning with **ADD AX, BX** than we do with **01 D8**, which is the way the instruction is actually represented. All programs for the 80x86 will simply be long strings of binary numbers.

Because of the processor's internal decoders, different binary patterns represent different instructions. Here are a few examples to illustrate this point:

01 D8	ADD	AX, BX	;add BX to AX, result in AX
29 D8	SUB	AX, BX	;subtract BX from AX, result in AX
21 D8	AND	AX, BX	;AX equals AX AND BX
40	INC	AX	;add 1 to AX
4B	DEC	BX	;subtract 1 from BX
8B C3	MOV	AX, BX	;copy BX into AX

Can you guess the meaning of each instruction just by reading it? Do the hexadecimal codes for each instruction mean anything to you? What we see here is the difference between **machine language** and **assembly language**. The machine language for each instruction is represented by the hexadecimal codes. This is the binary language of the machine. The assembly language is represented by the word-like terms that mean something to us. Putting groups of these word-like instructions together is how a program is constructed. In the next few sections, we will see how an assembly language program is written, converted into machine language, and executed.

## NUMOFF: Our First Machine Language Program

When the personal computer is first turned on, instructions in the start-up software turn the **NUM-LOCK** indicator on. This indicator is located near the **NUM-LOCK** button on the keyboard. Pushing **NUM-LOCK** manually every time the PC is turned on (or even rebooted) is annoying. Luckily, there is a single bit stored in a specific memory location used by DOS that controls the state of the **NUM-LOCK** indicator. We are about to see that it is possible to write an 80x86 program to manipulate the **NUM-LOCK** status bit.

## One Way to Create an Executable Program

One way to create a working program would involve these steps:

**Step 1:** Using a PC-based word processor, enter the following text file exactly as you see it. Save the file under the name of NUMOFF.ASM.

```
;NUMOFF.ASM: Turn NUM-LOCK indicator off.
;
        .MODEL SMALL
        .STACK
        .CODE
        .STARTUP
MOV      AX,40H          ;set AX to 0040H
MOV      DS,AX           ;load data segment with 0040H
MOV      SI,17H          ;load SI with 0017H
AND      BYTE PTR [SI], 0DFH ;clear NUM-LOCK bit
        .EXIT
        END
```

The twelve lines of code constitute a **source file**, the starting point of any 80x86-based program. Thus, NUMOFF.ASM is a source file.

To convert NUMOFF.ASM into a group of hexadecimal bytes that represents the corresponding machine language, we make use of two additional programs: ML and LINK. ML is an **assembler**, a program that takes a source file as input and determines the machine language for each source statement. ML creates two additional files. These are the *list* and *object* files. The list file contains all of the text from the source file, plus additional information, as we will soon see. The object file contains only the machine language.

**Step 2:** To assemble NUMOFF.ASM, enter the following command at the DOS prompt:

```
ML /c /F1 NUMOFF.ASM <cr>
```

where <cr> indicates a carriage return. Note: This command does not work until you install the assembler software. This command instructs ML to assemble NUMOFF.ASM and create NUMOFF.LST (the list file) and NUMOFF.OBJ (the object file). The list file created by ML looks like this:

```
;NUMOFF.ASM: Turn NUM-LOCK indicator off.
;
        .MODEL SMALL
        .STACK
0000      .CODE
        .STARTUP
0017  B8 0040  MOV      AX,40H          ;set AX to 0040H
001A  8E D8    MOV      DS,AX           ;load data segment with 0040H
001C  BE 0017  MOV      SI,17H          ;load SI with 0017H
001F  80 24 DF  AND      BYTE PTR [SI],0DFH ;clear NUM-LOCK bit
        .EXIT
        END
```

Here, it is obvious that ML has determined the machine language for each source statement. The first column is the set of memory locations where the instructions are stored. The second column is the group of machine language bytes that represent the actual 80x86 instructions.

**Step 3:** To make the object file executable, we run the LINK program, which converts NUMOFF.OBJ into NUMOFF.EXE. The DOS command to do this is:

```
LINK NUMOFF;<cr>
```

Now we have our first working 80x86 program, NUMOFF.EXE! To test it, press the NUM-LOCK button on the PC's keyboard until the NUM-LOCK light goes on. Then execute NUMOFF.EXE by entering:

```
NUMOFF.EXE<cr>
```

at the DOS prompt. The NUM-LOCK light should go off. This is what NUMOFF.EXE does. This program may have no effect in Windows XP and other environments, due to increased OS control over the hardware. However, it is an example of how a few simple instructions can do something useful.

Many of the programs in later chapters will be assembled and linked in this fashion.

## Generating Machine Code with DEBUG

An alternate technique for generating and executing machine code is through the use of the DEBUG program. Unlike the ML assembler program, DEBUG comes with the DOS and Windows operating systems. Let us briefly look at how DEBUG can be used to perform the same task as the NUMOFF.EXE program. We'll take a more detailed look at DEBUG later.

From the DOS prompt, start up DEBUG with:

```
DEBUG<cr>
```

You will get a minus sign (--) as a prompt, which helps to distinguish the DEBUG environment from the DOS environment.

Now enter all of the following text shown in bold. You will almost surely see different addresses on your machine than those that appear here, but this will not affect what we are trying to do.

```
-a<cr>
7F2D:0100 mov ax,40<cr>
7F2D:0103 mov ds,ax<cr>
7F2D:0105 mov si,17<cr>
7F2D:0108 and byte ptr [si],df<cr>
7F2D:010B <cr>
```

The <cr> on the last line gets you out of the "a" mode. Because "a" stands for *assemble*, DEBUG has assembled each statement entered by the user and placed the corresponding machine code into memory. This is indicated by the way addresses are incrementing in the address field (0100 to 0103 to 0105, etc.).

Now push the NUM-LOCK button on your keyboard so that the NUM-LOCK indicator is on. To execute the program statement by statement, enter a single "t" at DEBUG's prompt:

```
-t<cr>
```

This is the *trace* command, and it is used to single-step through an 80x86 program. Each time you enter "t," DEBUG will execute one instruction of your program and display the results.

If you hit "t" three more times, you should see the NUM-LOCK indicator go off. This completes the program. To exit from DEBUG back to DOS, enter:

`-q<cr>`

This shows that DEBUG is also a useful way of executing machine language programs. We will write and execute a number of programs this way and learn much more about how DEBUG works. We will also look at ways to develop programs specifically for protected-mode Windows and for Linux systems.

---

**Programming Exercise 1.1:** Can you think of a way to do the opposite of NUMOFF? That is, can you change NUMOFF so that it turns NUM-LOCK on?

*Hint:* Use an OR operation.

---

**Programming Exercise 1.2:** Can you think of a way to toggle the NUM-LOCK indicator? This would cause NUM-LOCK to alternate between on and off.

*Hint:* Use an XOR operation.

---

## 1.8 TROUBLESHOOTING TECHNIQUES

You may think it premature to begin discussing troubleshooting techniques, when we have been exposed to so little of the 80x86 family architecture. Even so, we have already seen a number of places where errors can occur, and it would be worthwhile to discuss them. For example, the NUMOFF.ASM source file could have contained one or more *typographical* errors, such as a misspelled instruction (MVO versus MOV), or a missing comma, or a comma where a semicolon was expected. Generally, when errors such as these are present in a source file, the assembler will report them with a brief error message.

Even if the source file does not have any typographical errors, we could still run into trouble. We could enter the command to invoke ML or DEBUG incorrectly, or not use the correct options.

When the source file correctly assembles and links, and an executable program has been created, there is still the possibility of a *run-time* error in the program. Run-time errors are typically caused by incorrect sequences of instructions and incomplete or faulty logical thinking.

To avoid a loss of time and effort, it is good to keep these common stumbling blocks in mind. Paying attention to the details will really pay off, as you learn to create a working program with a minimum of time and effort.

---

## SUMMARY

In this chapter, we have examined the operation of microprocessor-based systems. We saw that the complexity of the hardware, and thus of the software, is a function of the type of application. Through the use of many different types of peripherals, such as parallel and serial devices, analog-to-digital converters, and others, a system can be tailored to perform

almost any job. We also reviewed the basic fetch, decode, and execute cycle of a microprocessor, and examined the other duties the CPU performs, one of which was interrupt handling.

We also covered the initialization requirements of peripherals used in a microprocessor-based system, and why it is necessary to perform initialization in the first place. Other types of hardware and software requirements were also examined, such as the use of a watchdog monitor and a nonvolatile memory.

Four different generations of computers were presented and their differences highlighted. Current computing trends dealing with parallel processing and artificial intelligence were also introduced.

This was followed by an introduction to the motherboard hardware of a typical personal computer. Any *compatible* PC must use the same hardware. Because software is needed to control the hardware, we finished with a quick look at two techniques for creating and executing machine language programs. The first technique used ML and LINK, and the second technique used DEBUG. Both methods will be covered in detail in following chapters.

---

## STUDY QUESTIONS

1. Make a list of ten additional products containing microprocessors that we use every day.
2. (a) The *cycle time* of a microprocessor is the time for one complete clock cycle. For example, if the clock frequency of a microprocessor is 2 million cycles per second (2 MHz), then each cycle takes 500 ns (500 billionths of a second). Compare the cycle time of a microprocessor running at 2 MHz with one running at 50 MHz, 500 MHz, and 2 GHz.  
(b) If a certain 80x86 instruction requires four clock cycles to execute, how long does the instruction take to execute if the processor clock speed is 25 MHz? Repeat for a clock speed of 800 MHz.
3. Speculate on the uses for timing signals in the serial I/O, memory, and interrupt sections.
4. Why do math coprocessors enhance the capabilities of an ordinary CPU?
5. Draw a block diagram for a computerized cash register. The hardware should include a numerical display, a keyboard, and a compact printer.
6. What kind of initialization software would be required for the cash register in Question 5?
7. What would be the difference in system RAM requirements for two different cash registers, one without record keeping and one with?
8. What type of information should be stored in NVM during a power failure in a system designed to control navigation in an aircraft?
9. What types of interrupts may be required in a control system designed to monitor all doors, windows, and elevators in an office complex?
10. Name some advantages of downloading the main program into a microprocessor-based system. Are there any disadvantages?
11. Suppose that a number of robots making up a portion of an automobile assembly line are connected to a master factory computer. What kinds of information might be passed between the factory computer and the microprocessors controlling each robot?
12. A certain hard disk transfers data at the rate of 8 million bits per second. Explain why the CPU may not be able to perform the transfer itself, thus requiring the use of a DMA controller.



13. How does a plug-in card use motherboard hardware?
14. What additional kinds of plug-in cards can you think of?
15. Suppose that three microprocessors are used in the design of a new video game containing color graphics and complex sounds. How might each microprocessor function?
16. Windows is one operating system that runs on Intel microprocessors. What are three others?
17. List five different applications that might need the fast computing power of an RISC-based machine.
18. Besides Intel, what other companies make microprocessors? Search the Web for answers if you do not already know at least three.
19. A backward compatible microprocessor is one that can execute instructions from earlier models. How would a designer of the new CPU implement backward compatibility?
20. Which of the statements in the NUMOFF.ASM source file actually generated code?
21. It takes a certain amount of time to execute each instruction in NUMOFF. Which instruction do you think takes the longest?
22. What are all the NUMOFF files created in Section 1.7?
23. How are DEBUG statements different from statements in NUMOFF.ASM?
24. What advantages does a microcontroller have over a microprocessor? What disadvantages does it have?
25. Why do typographical errors prevent a source file from being assembled?

---

## CHAPTER 2

---

# An Introduction to the 80x86 Microprocessor Family

---

### OBJECTIVES

In this chapter you will learn about:

- Real-mode and protected-mode operation
- The register set of the 80x86 family
- The addressing capabilities and data types that may be used
- The different addressing modes and instruction types available
- The usefulness of interrupts
- Some of the differences between the 8086 and the 80286, 80386, 80486, and Pentium microprocessors

### KEY TERMS

Address size prefix	Double-word	Page fault
Addressing mode	Extended registers	Physical address
Base pointer	Extra segment	Protected mode
Byte	Flag register	Real mode
Byte-swapping	Gate descriptor	RISC
Cache	Gigabyte	Segment
Cache hit	Interrupt vector table	Segment descriptor
Cache miss	Instruction pointer	Segment registers
CISC	Little-endian	Source index
Code segment	Maximum mode	Stack pointer
Control flags	Megabyte	Stack segment
Data segment	Minimum mode	Status flags
Define byte	Multitasking	Word
Define word	Nonmaskable interrupt	
Destination index	Operand size prefix	

---

## 2.1 INTRODUCTION

Since its arrival on the computer scene, the 80x86 microprocessor family has quickly taken the lead in the personal computer market. The 80x86 processor is used everywhere, from dedicated control systems to superfast network file servers. Today, the Pentium offers compatibility with all previous 80x86 machines, with many new architectural improvements.

In this chapter, we will examine the features of the 80x86 microprocessor family. Only basic material will be covered, leaving the hardware and software details for upcoming chapters. From this chapter you will learn that the 80x86 is a machine with many possibilities.

Section 2.2 discusses the differences between real-mode and protected-mode operation. Section 2.3 covers the software model of the 80x86 family. Section 2.4 introduces the numerous registers contained within the processor, followed by a brief discussion of data organization, instruction types, and addressing modes in Sections 2.5, 2.6, and 2.7, respectively. Interrupts are the subject of Section 2.8. Sections 2.9 through 2.13 deal with the upward-compatible architectures found in the 8086, 80286, 80386, 80486, and Pentium microprocessors, respectively. Troubleshooting techniques are presented in Section 2.14.

---

## 2.2 REAL-MODE AND PROTECTED-MODE OPERATION

The first processor in the 80x86 family was the 16-bit 8086, which was capable of addressing 1MB of memory, a significant improvement over the 8-bit machines available in the late 1970s that typically addressed only 64KB of memory. Twenty address lines were provided on the processor to access the 1MB of memory ( $2^{20}$  equals 1,048,576 bytes, or 1MB). The advanced processors that followed the 8086, beginning with the 80286, all contained additional address lines. The 80386, 80486, and Pentium all contain 32 address lines, giving them the ability to access  $2^{32}$  bytes, or 4096MB, of memory. This large addressing space allows the advanced Intel processors to perform many operating system chores—such as multitasking (running more than one program at a time)—that are difficult, or even impossible, on the 8086.

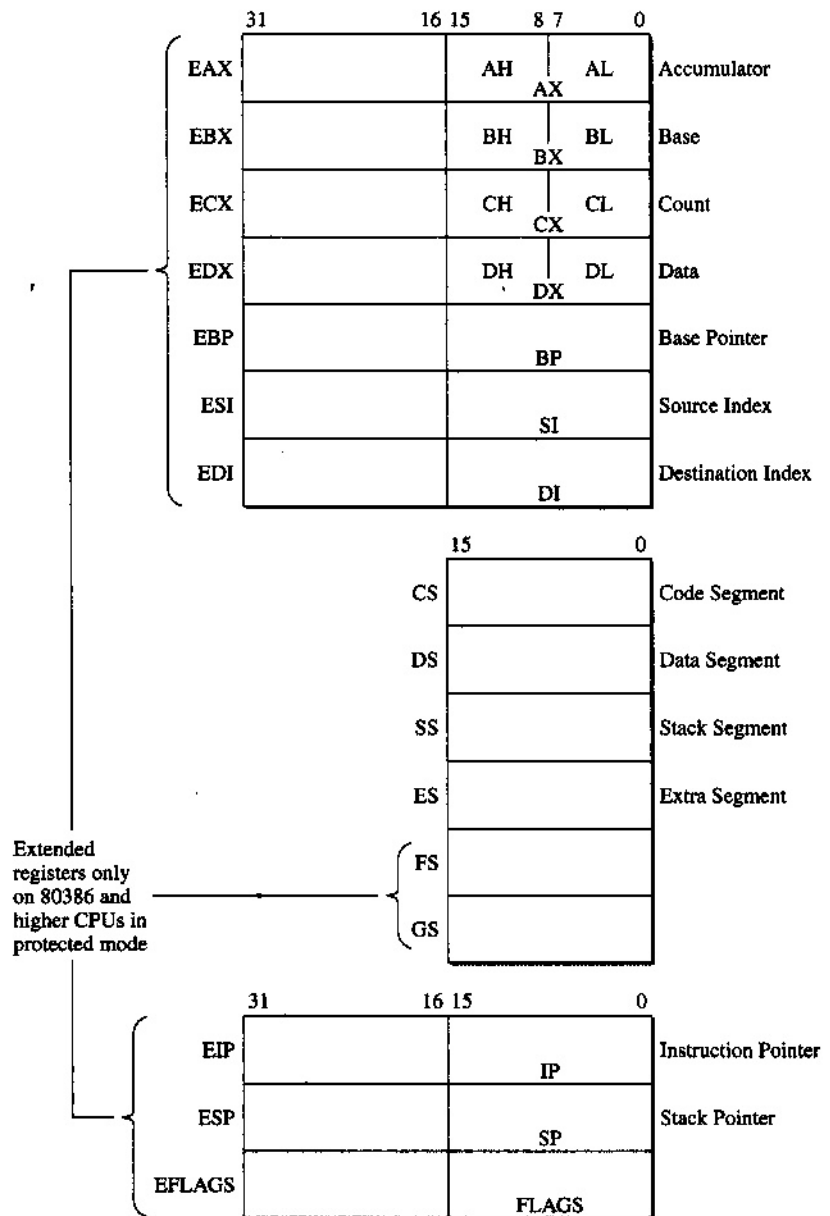
Beginning with the 80286, the advanced Intel processors all contained the ability to operate in two different modes of operation, **real mode** and **protected mode**. In real mode, the advanced processors, including the Pentium, simply operate like very fast 8086s, with the associated 1MB memory limit. Real-mode operation is automatically selected upon power-up. So a Pentium-based PC that boots up into DOS is operating in real mode (DOS is a real-mode operating system).

In protected mode, the full 4096MB (4 **gigabytes**) of memory is available to the processor, as are special privileged instructions and many other architectural goodies, including support for multitasking, virtual memory addressing, memory management and protection, and control over the internal data and instruction cache. The Windows operating system runs in protected mode to take advantage of these improvements. Writing programs that run in protected mode requires special background knowledge of operating systems theory. For this reason, the majority of the examples presented in this book are developed for real-mode operation. We will, however, in Chapter 14, explore the details of protected mode.

## 2.3 THE SOFTWARE MODEL OF THE 80x86 FAMILY

The 80x86 family of microprocessors contains four data registers referred to as AX, BX, CX, and DX. All are 16 bits wide and may be split up into two halves of 8 bits each. Figure 2.1 shows how each half is referred to by the programmer. Five other 16-bit registers are available for use as pointer or index registers. These registers are the **stack pointer** (SP), **base pointer** (BP), **source index** (SI), **destination index** (DI), and **instruction pointer** (IP). None of the five may be divided up in a manner similar to the data registers. AX, BX, CX, DX, BP, SI, and DI are referred to as *general purpose* registers.

**FIGURE 2.1** Software model of the 80x86 microprocessor family



Beginning with the 80386, the register sizes were extended to 32 bits. These new **extended registers** are referred to as EAX, EBX, ECX, and EDX. All are 32 bits wide. The lower 16 bits of each register are the original AX, BX, CX, and DX registers and may still be split up into halves of 8 bits each. Pointer and index registers are also extended. Two additional segment registers (FS and GS) were also added.

Normally the 32-bit registers are not fully available when the processor is operating in real mode. To maintain compatibility with earlier 80x86 machines, only the 16-bit registers AX, BX, CX, DX, BP, SI, and DI are available for use. A special technique can be used to utilize a 32-bit register (such as EAX) on an instruction-by-instruction basis. This will be demonstrated in Section 2.5.

A major difference between the 80x86 and many other CPUs on the market has to do with the next group of registers, the **segment registers**. The processor uses four segment registers to control all accesses to memory and I/O, and these registers must be maintained by the programmer. The **code segment** (CS) is used during instruction fetches, the **data segment** (DS) is most often used by default when reading or writing data, the **stack segment** (SS) is used during stack operations such as subroutine calls and returns, and the **extra segment** (ES) is used for anything the programmer wishes. All segment registers are 16 bits long. Section 2.4 explains in more detail how the segment registers are used.

Finally, a 32-bit **flag register** is used to indicate the results of arithmetic and logical instructions. Included are zero, parity, sign, and carry flags, as well as flags that are only used in protected mode. Together, these sixteen registers make an impressive set!

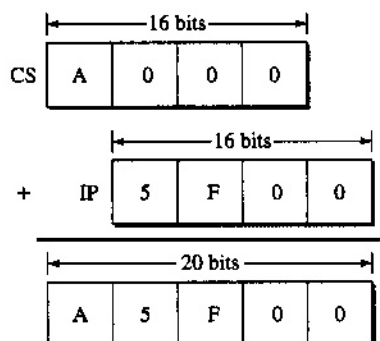
---

## 2.4 PROCESSOR REGISTERS

The 16-bit real-mode registers introduced in Section 2.3 are combined in an interesting way to form the necessary 20-bit address required by memory. If you recall, there were no 20-bit registers shown in the software model. How then does the processor generate 20-bit addresses in real mode?

### Segment Registers

The six segment registers, CS, DS, SS, ES, FS, and GS are all 16-bit registers controlled by the programmer. A real-mode segment, as defined by Intel, is a 64KB block of memory starting on any 16-byte boundary. Thus, 00000, 00010, 00020, 20000, 8CE90, and E0840 are all examples of valid segment addresses. The information contained in a segment register is combined with the address contained in another 16-bit register to form the required 20-bit address. Figure 2.2 shows how this is accomplished. In this example, the code segment register contains A000 and the instruction pointer contains 5F00. The processor forms the 20-bit address A5F00 in the following way: first, the data in the code segment register is shifted 4 bits to the left. This has the effect of turning A000 into A0000. Then the contents of the instruction pointer are added, giving A5F00. All external addresses are formed in a similar manner, with one of the segment registers used in each case. As we will see in Chapter 3, each segment register has a default usage. The processor knows which segment register to use to form an address for a particular application (instruction fetch, stack operation, and so on). The processor also allows the programmer to specify a different segment register when generating some addresses.

**FIGURE 2.2** Generating a 20-bit address in real mode

In protected mode, the segment registers are used as selectors that point to predefined **segment descriptors**, which are described in Chapter 14.

## General Purpose Registers

The seven general purpose registers (AX, BX, CX, DX, BP, SI, and DI) available to the programmer in real mode can be used in many different ways, and they also have some specific roles assigned to them. For instance, the accumulator (AX) is used automatically in multiply and divide operations and also in instructions that access I/O ports. The count register (CX) is used as a counter in certain loop instructions, providing up to 65,536 passes through a loop before termination. The lower half of CX, the 8-bit CL register, is also used as a counter in shift/rotate operations. Data register DX is used in multiply and divide operations and also as a pointer when accessing I/O ports. The last two registers are the source index and destination index (referred to as SI and DI, respectively). These registers are used as pointers in string operations.

Even though these registers have specific uses, they may be used in many other ways simply as general purpose registers, allowing for many different 16-bit operations.

Recall that the general purpose registers are actually 32 bits wide. However, when the processor operates in real mode, it defaults to the original 16-bit 8086 register sizes. It is possible to take advantage of the 32-bit extended registers while running in real mode. A special 1-byte code called the **operand size prefix** is inserted before each instruction that uses a 16-bit register. For these instructions, the processor will use the full 32-bit register length. This 1-byte code has the value 66 hexadecimal. Let us examine an actual problem to solve using the 32-bit registers.

### ■ EXAMPLE 2.1

The distance from Earth to the sun is approximately 93 million miles. The speed of light is roughly 186,000 miles per second. How can we determine the time required for a ray of light to reach Earth?

**Solution:** First, convert the input numbers into hexadecimal:

93,000,000	=	058B1140
186,000	=	0002D690

Both numbers require more than 16 bits of storage and will not fit into any of the 16-bit general purpose registers. If the operand size prefix code is used, both numbers are easily



stored in the extended registers. We can use the DEBUG program to solve the problem. Here are the contents of a short text file called SPEED.SCR:

```
a
db 66
sub dx,dx
db 66
mov ax,1140
dw 058b
db 66
mov bx,d690
dw 0002
db 66
div bx

r
t 4
q
```

The first line contains DEBUG's **a** (assemble) command. This command instructs DEBUG to begin assembling real-mode instructions. The second line contains the **db** (define byte) command and is used to supply the single-byte operand size prefix. Note the use of **db 66** before each of the four instructions. The three lines

```
db 66
mov ax,1140
dw 058b
```

will execute as if they were written like this:

```
mov eax,058b1140
```

The **dw** (define word) command is used to supply the upper 16 bits of the extended register value. This is how the distance value is set up in register EAX, with the processor running in real mode. A similar method is used to initialize the speed of light in register EBX.

The **div bx** instruction (with its prefix of 66) will divide EAX (the distance) by EBX (the speed of light). The blank line after the **DIV** instruction is used to terminate the assemble command. The **r** (register) command on the next line causes DEBUG to display the initial register values. Then comes the **t** (trace) command, which traces the execution of the four instructions entered with the assemble command. The last line contains the **q** (quit) command, to quit DEBUG and return to DOS.

All of these commands can be entered manually from the keyboard after starting DEBUG. Because they are all contained in the SPEED.SCR file, redirecting the DOS input device with the command:

```
C> DEBUG < SPEED.SCR
```

will cause DEBUG to read the commands from the SPEED.SCR file instead.

Either way, the final register display from DEBUG looks like this:

```
AX=01F4 BX=D690 CX=0000 DX=1140 SP=FFEE BP=0000 SI=0000 DI=0000
DS=229B ES=229B SS=229B CS=229B IP=0112 NV UP EI NG NZ NA PO NC
229B:0112 C6F70A      MOV  BH,0A
```

When the **DIV** instruction executes, the value 058B1140 is divided by 2D690, giving a final result in AX of 01F4 (which indicates a time of 500 seconds, or 8 minutes and 20 seconds).

Try running DEBUG with the **db 66** statements removed. The results are drastically different. ■

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	NT	IOPL	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF	

**FIGURE 2.3** Lower word of flag register

## Flag Register

Figure 2.3 shows the eleven flag assignments within the lower 16 bits of the flag register. The flags are divided into two groups: **control flags** and **status flags**. The control flags are IF (*interrupt enable flag*), DF (*direction flag*), and TF (*trap flag*). The status flags are CF (*carry flag*), PF (*parity flag*), AF (*auxiliary carry flag*), ZF (*zero flag*), SF (*sign flag*), OF (*overflow flag*), NT (*nested task*), and IOPL (*input/output privilege level*). Most of the instructions that require the use of the ALU affect the flags. Remember that the flags allow ALU instructions to be followed by conditional instructions. Programs make decisions based on the state of its flags. So the flags influence how code executes on the processor.

The content/operation of each flag is as follows:

CF: Contains carry out of MSB of result

PF: Indicates if result has even parity

AF: Contains carry out of bit 3 in AL

ZF: Indicates if result equals zero

SF: Indicates if result is negative

OF: Indicates that an overflow occurred in result

IF: Enables/Disables interrupts

DF: Controls pointer updates during string operations

TF: Provides single-step capability for debugging

IOPL: Priority level of current task

NT: Indicates if current task is nested

The upper 16 bits of the flag register are used for protected-mode operation. See Chapter 14 for details.

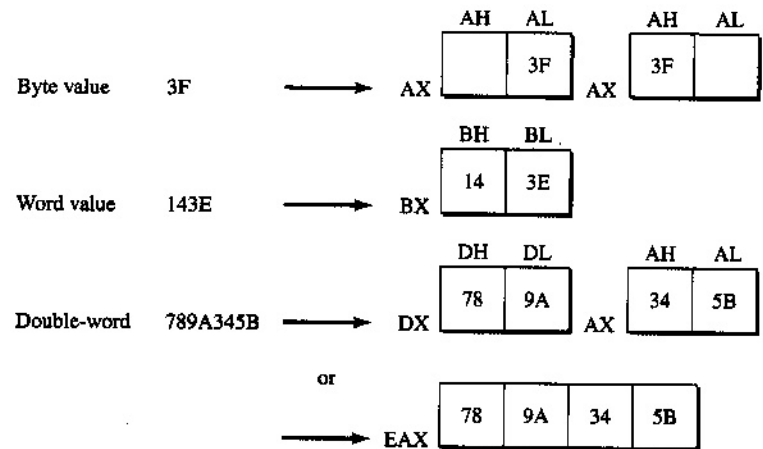
---

## 2.5 DATA ORGANIZATION

The 80x86 has the capability of performing operations on many different types of data. In this section, we will examine what some of the more common data types are and how they are represented and used by the processor.

### Bits, Bytes, and Words

The processor contains instructions that directly manipulate single bits and other instructions that use 8-, 16-, and even 32-bit integers. By common practice, 8-bit binary numbers are referred to as **bytes**. Processor register halves AL, BH, and CL are examples of where bytes might be stored and used.

**FIGURE 2.4** Storing different data types in registers

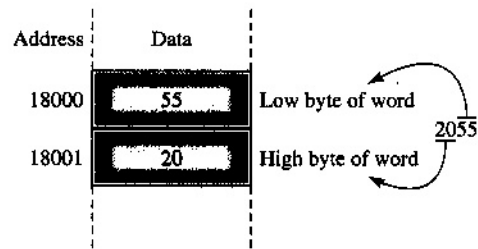
Sixteen-bit numbers are known as **words** and require an entire real-mode (16-bit) processor register for storage. Registers DX, BP, and SP are used to hold word data types. In register DX, DH will contain the upper 8 bits of the number, and DL will hold the lower 8 bits.

Some instructions (particularly multiply and divide) require the use of 32-bit numbers. These data types are called **double-words** (or long-words). In this case, the 32-bit number is stored in registers DX and AX, with DX holding the upper 16 bits of the number. Even though an extended register, such as EAX, may be used to store a 32-bit number, it is difficult to work with extended registers in real mode. These data types are illustrated in Figure 2.4.

It is important to keep track of the data type being used in an instruction, because incorrect or undefined types may lead to incorrect program assembly or execution.

One of the differences between the Intel line of microprocessors and those made by other manufacturers is Intel's way of storing 16-bit numbers in memory. A method that began with the 8-bit 8080 and has been used on all upgrades from the 8085 to the Pentium is a technique called **byte-swapping**. This technique is sometimes confusing for those unfamiliar with it, but becomes clear after a little exposure. When a 16-bit number must be written into the system's byte-wide memory, the low-order 8 bits are written into the first memory location and the high-order 8 bits are written into the second location. Figure 2.5 shows how the 2 bytes that make up the 16-bit hexadecimal number 2055 are written into locations 18000 and 18001, with the low-order 8 bits (55) going into the first location (18000). This is what is known as byte-swapping. The lower byte is always written first, followed by the high byte. Byte-swapping is one of the most significant differences between Intel processors and other machines, such as Motorola's 68000™ (which does not swap bytes). Intel CPUs are also called "**little-endian**" computers, while Motorola processors are called "**big-endian**" computers.

Reading the 16-bit number out of memory is performed automatically by the processor with the aid of certain instructions. The processor knows that it is reading the lower byte first and puts it in the correct place. Programmers who manipulate data in memory must remember to use the proper practice of byte-swapping, or they will discover that their programs do not give the correct, or expected, results. It should be easy to remember the mechanics of byte-swapping because the lower byte is always read/written to the lower memory address.

**FIGURE 2.5** Intel byte-swapping

### Assembler Directives DB, DW, DUP, and EQU

Representing data types in a source file requires the use of special *assembler directives* designed to create the required data and perform all appropriate memory allocation and data formatting for little-endian values. Consider the following sample portion of a list file:

```

0000                                .DATA
0000 03                            NUM1    DB    3
0001 64                            NUM2    DB    100
0002 00                            NUM3    DB    ?
0003 0F 30 C8 3A CE                NUMS    DB    15,48,200,3AH,0CEH
0008 48 65 6C 6C 6F 24            MSG     DB    'Hello$'

000E 0006                          WX      DW    6
0010 03E8                          WY      DW    1000
0012 1234 ABCD                      WZ      DW    1234H,0ABCDH
0016 0000                          TEMP    DW    ?

0018 000A [00]                      SCORES  DB    10 DUP(0)
0022 0007 [0000]                    TIMES   DW    7 DUP(?)

= 000D                              TOP     EQU    13
= 157C                              MORE    EQU    5500

```

In this example, byte and word data types are defined through the use of the **DB (define byte)** and **DW (define word)** assembler directives. As you can see, DB and DW both allow one or more numbers in their data field or no numbers at all, as is the case with the ? in the data field. DB and DW will convert any numbers in their data fields into hexadecimal and store the numbers in the appropriate place within the object file. If an ASCII character string is found in the data field (as in "Hello\$") the ASCII byte associated with each character is generated.

Because each DB or DW statement in the source file may generate different lengths of data, the assembler keeps track of relative memory locations using a program counter to indicate the starting address of each data group. For example, the bytes for the fourth DB statement begin at address 0003 within the data segment, and the word for the second DW statement is located at address 0010 within the data segment. In many cases, we supply a label with a particular DB or DW statement to use with instructions found elsewhere in the source file. The labels are assigned the value of the starting address in each DB or DW statement. Examine the sample data again. Do you see why the address associated with MSG is 0008?

DB and DW statements can be used to simply reserve byte or word space by using ? in the data field. When many reserved bytes or words are needed, the DUP (duplicate) assembler directive is used. Although it is legal to use DB ?,?,?,? in a source file, a simple DB 5 DUP(?) should be used instead. This saves the programmer the effort of having to

count the number of question marks entered. DUP has an additional advantage. Suppose that 2000 reserved bytes are needed. The statement `DB 2000 DUP(?)` is clearly more desirable than two hundred `DB ?,?,?,?,?,?,? ?` statements. In our example, DUP is used to create both byte (SCORES) and word (TIMES) data tables.

One last assembler directive is very useful because it defines a value that can be used in other source statements but does *not* generate any code. This is the EQU (equate) directive. Notice how EQU is used to assign the value 13 to TOP and the value 5500 to MORE. Any instruction in the source program that uses TOP and MORE will automatically use the values 13 and 5500, respectively. From a practical viewpoint, suppose you have written a 5000-line source program that contains the instruction

```
MOV AL, 77
```

in many different places. If, for some reason, you had to change all the 77s to 88s, you might be in for a long editing session. A simple solution would be to define the value 77 with an EQU statement, as in

```
VAL EQU 77
```

then use the EQU label in each instruction, like so

```
MOV AL, VAL
```

If it is necessary to change 77 to 88, only the EQU statement has to be edited.

This brief introduction to data types should help us when we examine the 80x86 instruction set.

---

## 2.6 INSTRUCTION TYPES

The 80x86 instruction set is composed of six main groups of instructions. A discussion of instruction specifics will be postponed until Chapters 3 and 4. Examining the instructions briefly here, however, will give a good overall picture of the capabilities of the processor. All instructions are part of the original 8086 instruction set, unless otherwise indicated.

### Data Transfer Instructions

Data transfer instructions are used to move data among registers, memory, and the outside world. Also, some instructions directly manipulate the stack (a special area in memory used for temporary values), while others may be used to alter the flags.

The data transfer instructions are:

IN	Input byte or word from I/O port
LAHF	Load AH from flags
LDS	Load pointer using data segment
LEA	Load effective address
LES	Load pointer using extra segment
MOV	Move to/from register/memory
OUT	Output byte or word to I/O port
POP	Pop word off stack
POPF	Pop flags off stack
PUSH	Push word onto stack

<b>PUSHF</b>	Push flags onto stack
<b>SAHF</b>	Store AH into flags
<b>XCHG</b>	Exchange byte or word
<b>XLAT</b>	Translate byte

Additional 80286 instructions are:

<b>INS</b>	Input string from port
<b>OUTS</b>	Output string to port
<b>POPA</b>	Pop all registers
<b>PUSHA</b>	Push all registers

Additional 80386 instructions are:

<b>LFS</b>	Load pointer using FS
<b>LGS</b>	Load pointer using GS
<b>LSS</b>	Load pointer using SS
<b>MOVSX</b>	Move with sign extended
<b>MOVZX</b>	Move with zero extended
<b>POPAD</b>	Pop all double (32-bit) registers
<b>POPD</b>	Pop double register
<b>POPFD</b>	Pop double flag register
<b>PUSHAD</b>	Push all double registers
<b>PUSHD</b>	Push double register
<b>PUSHFD</b>	Push double flag register

Additional 80486 instruction is:

<b>BSWAP</b>	Byte swap
--------------	-----------

New Pentium instruction is:

<b>MOV</b>	Move to/from control register
------------	-------------------------------

## Arithmetic Instructions

These instructions make up the arithmetic group. Byte and word operations are available on almost all instructions. A nice addition are the instructions that multiply and divide. Previous 8-bit microprocessors did not include these instructions, forcing the programmer to write subroutines to perform multiplication and division when needed. Addition and subtraction of both binary and BCD operands are also allowed.

The arithmetic instructions are:

<b>AAA</b>	ASCII adjust for addition
<b>AAD</b>	ASCII adjust for division
<b>AAM</b>	ASCII adjust for multiply
<b>AAS</b>	ASCII adjust for subtraction
<b>ADC</b>	Add byte or word plus carry
<b>ADD</b>	Add byte or word
<b>CBW</b>	Convert byte or word
<b>CMP</b>	Compare byte or word

CWD	Convert word to double-word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC	Decrement byte or word by one
DIV	Divide byte or word (unsigned)
IDIV	Integer divide byte or word
IMUL	Integer multiply byte or word
INC	Increment byte or word by one
MUL	Multiply byte or word (unsigned)
NEG	Negate byte or word
SBB	Subtract byte or word and carry
SUB	Subtract byte or word

Additional 80386 instructions are:

CDQ	Convert double-word to quadword
CWDE	Convert word to double-word

Additional 80486 instructions are:

CMPXCHG	Compare and exchange
XADD	Exchange and add

New Pentium instruction is:

CMPXCHG8B	Compare and exchange 8 bytes
-----------	------------------------------

## Bit Manipulation Instructions

Instructions capable of performing logical, shift, and rotate operations are contained in this group. Many common Boolean operations (AND, OR, NOT) are available in the logical instructions. These, as well as the shift and rotate instructions, operate on bytes or words. Single-bit operations are available on processors from the 80386 and up.

The bit manipulation instructions are:

AND	Logical AND of byte or word
NOT	Logical NOT of byte or word
OR	Logical OR of byte or word
RCL	Rotate left through carry byte or word
RCR	Rotate right through carry byte or word
ROL	Rotate left byte or word
ROR	Rotate right byte or word
SAL	Arithmetic shift left byte or word
SAR	Arithmetic shift right byte or word
SHL	Logical shift left byte or word
SHR	Logical shift right byte or word
TEST	Test byte or word
XOR	Logical exclusive-OR of byte or word



Additional 80386 instructions are:

BSF	Bit scan forward
BSR	Bit scan reverse
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset
BTS	Bit test and set
SETcc	Set byte on condition
SHLD	Shift left double precision
SHRD	Shift right double precision

## String Instructions

String operations simplify programming whenever a program must interact with a user. User commands and responses are usually saved as ASCII strings of characters, which may be processed by the proper choice of string instruction.

The string instructions are:

CMPS	Compare byte or word string
LODS	Load byte or word string
MOVS	Move byte or word string
MOVSB (MOVSW)	Move byte string (word string)
REP	Repeat
REPE (REPZ)	Repeat while equal (zero)
REPNE (REPNZ)	Repeat while not equal (not zero)
SCAS	Scan byte or word string
STOS	Store byte or word string

## Program Transfer Instructions

This group of instructions contains all jumps, loops, and subroutine (called procedure) and interrupt operations. The great majority of jumps are conditional, testing the processor flags before execution.

The program transfer instructions are:

CALL	Call procedure (subroutine)
INT	Interrupt
INTO	Interrupt if overflow
IRET	Return from interrupt
JA (JNBE)	Jump if above (not below or equal)
JAE (JNB)	Jump if above or equal (not below)
JB (JNAE)	Jump if below (not above or equal)
JBE (JNA)	Jump if below or equal (not above)
JC	Jump if carry set
JCXZ	Jump if CX equals zero

JE (JZ)	Jump if equal (zero)
JG (JNLE)	Jump if greater (not less or equal)
JGE (JNL)	Jump if greater or equal (not less)
JL (JNGE)	Jump if less (not greater or equal)
JLE (JNG)	Jump if less or equal (not greater)
JMP	Unconditional jump
JNC	Jump if no carry
JNE (JNZ)	Jump if not equal (not zero)
JNO	Jump if no overflow
JNP (JPO)	Jump if no parity (parity odd)
JNS	Jump if no sign
JO	Jump if overflow
JP (JPE)	Jump if parity (parity even)
JS	Jump if sign
LOOP	Loop unconditional
LOOPE (LOOPZ)	Loop if equal (zero)
LOOPNE (LOOPNZ)	Loop if not equal (not zero)
RET	Return from procedure (subroutine)

Additional 80286 instructions are:

BOUND	Check index against array bounds
ENTER	Enter a procedure
LEAVE	Leave a procedure

Additional 80386 instructions are:

IRETD	Interrupt return
JECXZ	Jump if ECX is zero

## Processor Control Instructions

This last group of instructions performs small tasks that sometimes have profound effects on the operation of the processor. Many of the instructions manipulate the flags.

The processor control instructions are:

CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt enable flag
CMC	Complement carry flag
ESC	Escape to external processor
HLT	Halt processor
LOCK	Lock bus during next instruction
NOP	No operation
STC	Set carry flag
STD	Set direction flag

STI	Set interrupt enable flag
WAIT	Wait for TEST pin activity

Additional 80286 instructions (protected mode only) are:

ARPL	Adjust requested privilege level
CLTS	Clear task switched flag
LAR	Load access rights
LGDT	Load global descriptor table
LIDT	Load interrupt descriptor table
LLDT	Load local descriptor table
LMSW	Load machine status word
LSL	Load segment limit
LTR	Load task register
SGDT	Store global descriptor table
SIDT	Store interrupt descriptor table
SLDT	Store local descriptor table
SMSW	Store machine status word
STR	Store task register
VERR	Verify segment for reading
VERW	Verify segment for writing

Additional 80486 instructions are:

INVD	Invalidate cache
INVLPG	Invalidate TLB entry
WBINVD	Write back and invalidate cache

New Pentium instructions are:

CPUID	CPU identification
RDMSR	Read from model-specific register
RDTSC	Read from time stamp counter
RSM	Resume from system management mode
WRMSR	Write to model-specific register

In Chapter 3 we will begin examining each of the 80x86 instructions in detail and see many examples of how they are used.

---

## 2.7 ADDRESSING MODES

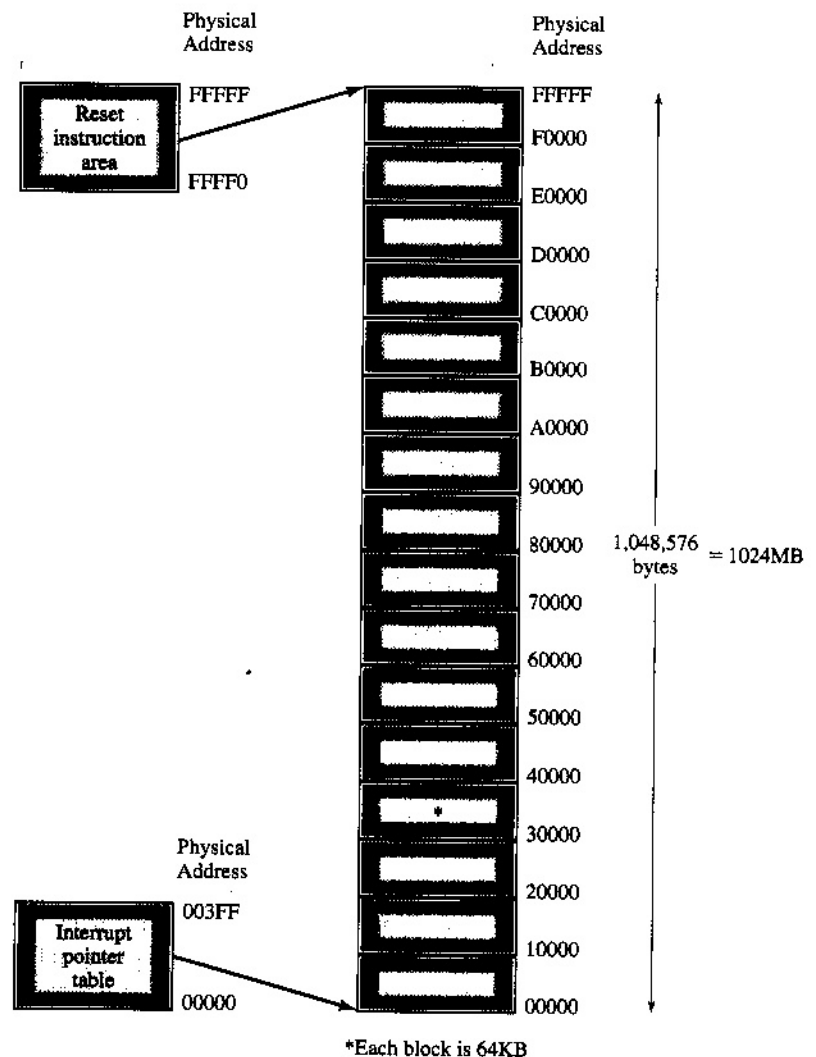
The 80x86 offers the programmer a wide number of choices when referring to a memory location or immediate data value. Many people believe that the number of **addressing modes** contained in a microprocessor is a measure of its power. If that is so, the 80x86 should be counted among the most powerful processors. Many of the addressing modes are used to generate a **physical address** in memory. Recall from Figure 2.2 that a 20-bit

real-mode address is formed by the sum of two 16-bit address values. One of the four segment registers will always supply the first 16-bit address. The second 16-bit address is formed by a specific addressing mode operation. The resulting 20-bit address points to one specific location in the processor's 1MB real-mode addressing space. We will see that there are a number of different ways the second part of the address may be generated. Protected-mode addressing will be covered in Chapter 14.

## Real-Mode Addressing Space

All addressing modes eventually create a physical address that resides somewhere in the 00000 to FFFFFF addressing space of the processor. Figure 2.6 shows a brief memory map of the real-mode addressing space, which is broken up into 16 blocks of 64KB each. Each 64KB block is called a **segment**. A segment contains all the memory locations that can be

**FIGURE 2.6** Addressing space of the processor in real mode



reached when a particular number in a certain segment register is used. For example, if the data segment contains 0000, then addresses 00000 through 0FFFF can be generated when using the data segment. If, instead, register DS contains 1800, then the range of addresses becomes 18000 through 27FFF. It is important to see that a segment can begin on *any* 16-byte boundary. So, 00000, 00010, 00020, 035A0, 10800, and CCE90 are all acceptable starting addresses for a segment.

Altogether, 1,048,576 bytes can be accessed by the processor. This is commonly referred to as 1 **megabyte**. Small areas of the addressing space are reserved for special operations. At the very high end of memory, locations FFFF0 through FFFFF are assigned the role of storing the initial instruction used after a RESET operation. At the low end of memory, locations 00000 through 003FF are used to store the addresses for all 256 interrupts (although not all are commonly used in actual practice). This dedication of addressing space is common among processor manufacturers and may force designers to conform to specific methods or techniques when building systems around the 80x86. For instance, EPROM is usually mapped into high memory, so that the starting execution instructions will always be there at power-on.

## Addressing Modes

The simplest addressing mode is known as immediate. Data needed by the processor is actually included in the instruction. For example:

```
MOV CX, 1024
```

contains the immediate data value 1024. This value is converted into binary and included in the code of the instruction. The data comes immediately after the opcode.

When data must be moved between registers, register addressing is used. This form of addressing is very fast, because the processor does not have to access external memory (except for the instruction fetch). An example of register addressing is

```
ADD AL, BL
```

where the contents of registers AL and BL are added together, with the result stored in register AL. Notice that both operands are names of internal registers and must be the same size.

The programmer may refer to a memory location by its specific address by using direct addressing. Two examples of direct addressing are:

```
MOV AX, [3000]
```

and

```
MOV BL, COUNTER
```

In each case, the contents of memory are loaded into the specified registers. The first instruction uses square brackets to indicate that a memory address is being supplied. Thus, 3000 and [3000] are allowed to have two different meanings. 3000 means the number 3000, whereas [3000] means the number stored at memory location 3000. The second instruction uses the symbol name COUNTER to refer to memory. COUNTER must be defined somewhere else in the program for it to be used this way.

When a register is used within square brackets, the processor uses register indirect addressing. For example:

```
MOV BX, [SI]
```

instructs the processor to use the 16-bit quantity stored in the SI (source index) register as a memory address. A slight variation produces indexed addressing, which allows a small offset value to be included in the memory operand. Consider this example:

```
MOV BX, [SI + 10]
```

The location accessed by the instruction is the sum of the SI register and the offset value 10.

When the register used is the base pointer (BP) or the BX register, **based** addressing is employed. This addressing mode is especially useful when manipulating data in large tables or arrays. An example of based addressing is:

```
MOV CL, [BP + 4]
```

Including an index register (SI or DI) in the operand produces **based-indexed** addressing. The address is now the sum of the base pointer and the index register. An example might be:

```
MOV [BP + DI], AX
```

When an offset value is also included in the operand, the processor uses **based-indexed with displacement** addressing. An example is:

```
MOV DL, [BP + SI + 2]
```

Obviously, the 80x86 allows the base pointer to be used in many different ways.

Other addressing modes are used when string operations must be performed.

The processor is designed to access I/O ports, as well as memory locations. When **port** addressing is used, the address bus contains the address of an I/O port instead of a memory location. I/O ports may be accessed two different ways. The port may be specified in the operand field, as in:

```
IN AL, 80H
```

or indirectly, via the address contained in register DX:

```
OUT DX, AL
```

Using DX allows a port range from 0000 to FFFF, or 65,536 individual I/O port locations. Only 256 (00 to FF) ports are allowed when the port address is included as an immediate operand.

All of these addressing modes will be covered again in detail in Chapter 3.

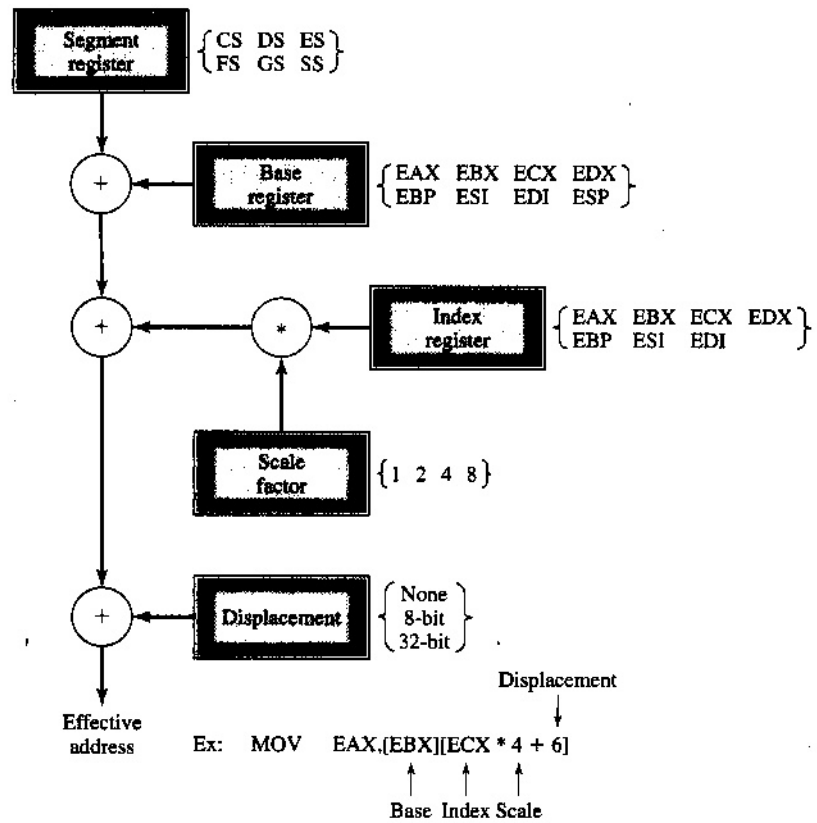
## 32-Bit Addressing Modes

The Pentium architecture supports an additional method of generating addresses, especially designed to take advantage of protected-mode operation. In protected mode, addresses are 32 bits wide, spanning a 4-gigabyte range. The 32-bit addresses are generated as indicated in Figure 2.7.

As usual, a segment register is used as part of the address calculation. Unlike real-mode addressing, any general purpose register may be used as a base register or index register. The only exception is ESP, which can be used as a base register but not an index register.

A scale factor is also included to multiply the contents of the index register by 1, 2, 4, or 8. This is very useful when dealing with arrays of data composed of bytes, words, double-words, or quad-words. The MOV instruction shown in Figure 2.7 multiplies index register ECX by a scale factor of four.

Though designed for protected-mode applications, the 32-bit addressing modes may be used while running in real mode by using an **address size prefix** byte before the instruction that uses the 32-bit addressing mode. This is illustrated in Example 2.2.

**FIGURE 2.7** Generating a 32-bit address**EXAMPLE 2.2**

Examine the following portion of a list file for a real-mode program with instructions using 32-bit registers or addressing modes:

0010 B4 09	MOV	AH, 9
0012 BD 16 0000 R	LEA	DX, TABLE
0016 CD 21	INT	21H
0018 B4 09	MOV	AH, 9
001A 66 8D 1E 0000 R	LEA	EBX, TABLE
001F 66 BA 00000002	MOV	EDX, 2
0025 67 8D 14 93	LEA	DX, [EBX][EDX*4]
0029 CD 21	INT	21H

The second LEA instruction and the following MOV instruction both contain extended registers in their operand fields. Because real-mode code is being generated, the default register size is 16 bits. Thus, the operand size prefix byte 66H is used prior to the code for the LEA and MOV instructions to switch to 32-bit mode.

In a similar fashion, the address size prefix byte 67H is used before the third LEA instruction to allow processing of the 32-bit addressing mode specified by EBX, EDX, and the scale factor. As with the operand size prefix, the address size prefix works for a single instruction only and must precede all instructions that utilize 32-bit addressing. ■



## 2.8 INTERRUPTS

A hardware interrupt is an event that occurs while the processor is executing an instruction. The instruction might be part of a group of instructions in a *main* program, such as a word processing application. The interrupt temporarily suspends execution of the main program in favor of a special routine that services the interrupt. When interrupt processing is complete, the processor is returned back to the exact place in the main program where it left off. For example, a timer interrupt might occur while the word processing application is in the middle of a spell-checking procedure. Spell checking is suspended and the timer interrupt service routine takes over. The routine might simply increment the seconds counter on the time-of-day clock. When the interrupt is finished, spell checking resumes.

Let us take a look at how interrupts are implemented by the 80x86 and how one particular interrupt is used to control the computer when DOS is running.

### Hardware and Software Interrupts

The processor is capable of responding to 256 different types of interrupts. These interrupts are generated in a number of different ways. External hardware interrupts are caused by activating the processor's NMI and INTR signals. NMI is a **nonmaskable interrupt** and cannot be ignored by the CPU. INTR is a maskable interrupt that the processor may choose to ignore depending on the state of an internal interrupt enable flag.

Internal software interrupts are caused by execution of an INT instruction. INT is followed by an interrupt number from 0 to 255, giving the programmer the option of generating any number of specific interrupts during program execution. Machines based on the 80x86 that contain a software disk operating system (DOS) have very specific functions assigned to certain interrupts (INT 21H, for example) that allow the user to read the keyboard, write text to the screen, control disk drives, and so forth. Note that the INT instruction does not interrupt anything like a hardware interrupt does; it is simply an instruction executed during normal program flow.

Some interrupts are generated internally by the processor itself. Divide-error is one example. This interrupt is caused when division by zero is detected in the execution unit (during execution of the IDIV or DIV instructions). The processor can also generate single-step interrupts at the end of every instruction if a certain flag called the trap flag is set. Another internal interrupt is INTO (interrupt on overflow).

### The Interrupt Vector Table

All interrupts use a dedicated table in memory for storage of their interrupt service routine (ISR) starting addresses. The table is called an **interrupt vector table** (or interrupt pointer table) and is 1,024 bytes long, enough storage space for 256 4-byte entries. Because an ISR address occupies 4 bytes of storage, the table holds addresses for all 256 interrupts. Each ISR address is composed of a 2-byte CS value and a 2-byte instruction pointer address. Thus, if the table entry for a type-0 interrupt (divide error) were CS:0100 and IP:0400, the divide error ISR code would have to be located at physical address 01400.

Interrupt processing will be covered in detail in Chapter 5. For now, let us examine one particularly useful interrupt.

### A Brief Look at DOS Interrupt 21H

One of the most useful DOS interrupts is number 21H. This interrupt was chosen as the entry point into DOS for programmers writing their own DOS applications. Although there

are many other interrupts assigned to specific functions by DOS, INT 21H is loaded with so many different functions we rarely need to use others.

It is possible to use DEBUG to determine where the code for DOS's INT 21H routine (or any other interrupt) is located. Because each interrupt vector (CS:IP) requires 4 bytes, the offset into memory for INT 21H's vector is four times 21H, or address 00084H. The following DEBUG session uses this address to find out where the ISR for INT 21H is located:

```
C> debug
-d 0:80 L 10
0000:0080 94 10 16 01 B4 16 26 07-4F 03 FB 0A 8A 03 FB 0A
-q
```

The CS:IP addresses for INT 21H are shown in bold. In this example, CS = 0726H and IP = 16B4H (remember that words are byte-swapped). This gives a 20-bit memory address of 08914H. To examine the actual instructions within the INT 21H service routine, use DEBUG's unassemble command with the CS:IP addresses shown (e.g., -u 726:16b4).

The way INT 21H is used is simply a matter of loading specific registers with data and issuing an INT 21H. For example, to read the computer's time we use the following two instructions:

```
MOV AH,2CH ;get system time function number
INT 21H ;DOS call
```

DOS will return the time as follows:

```
CH = hours
CL = minutes
DH = seconds
DL = 100ths of seconds
```

Many of the programs we will examine in later chapters use INT 21H and other interrupts.

## 2.9 THE 8086: THE FIRST 80x86 MACHINE

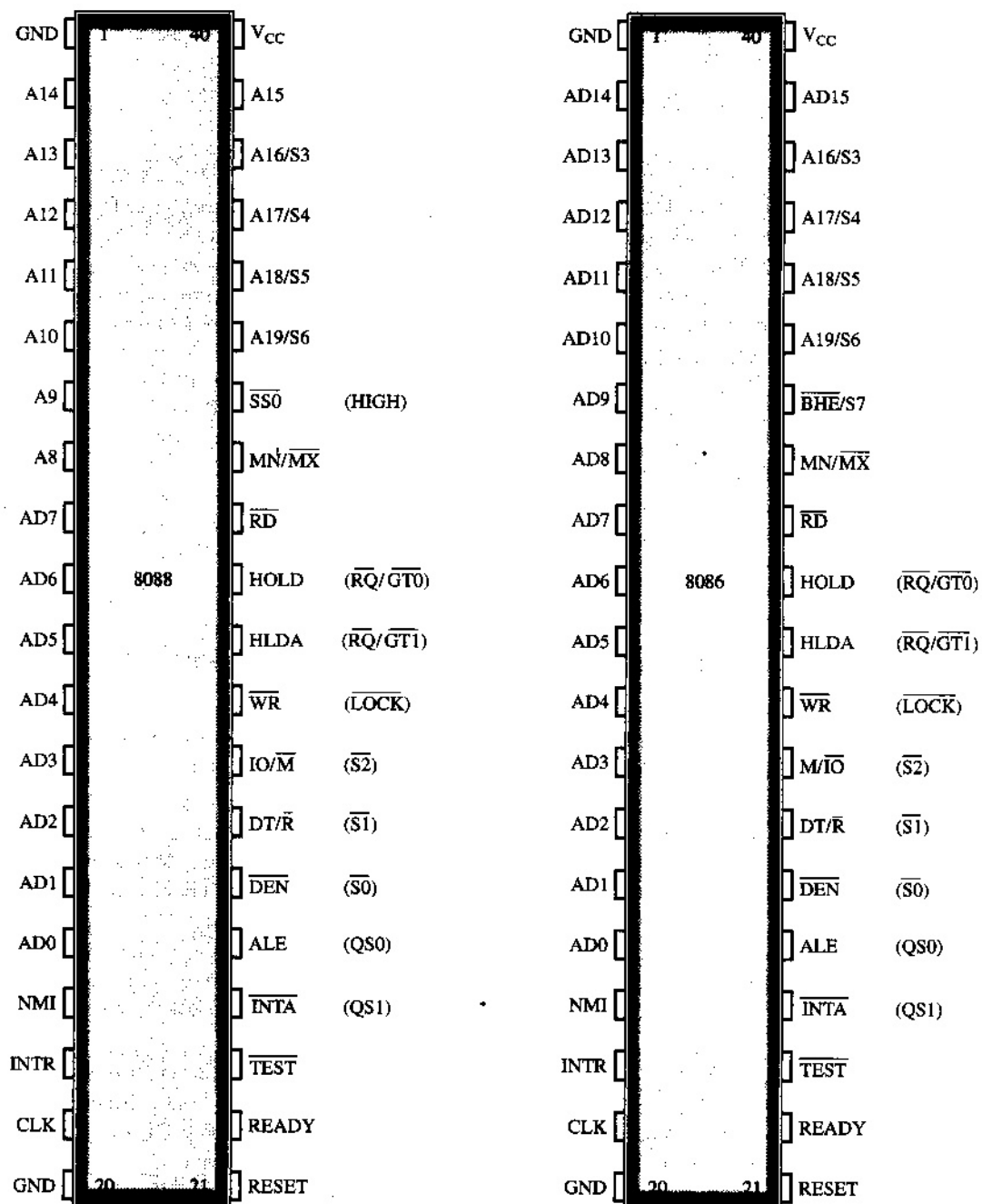
This section and those that follow describe the historical evolution of the 80x86 family. This is done to gain an appreciation for the improvements that have been made to the 80x86 architecture prior to the design of the Pentium. We begin with the 8086.

The 8086 microprocessor is a 16-bit machine with a 16-bit data bus and a 20-bit address bus. This allows for  $2^{20}$ , or 1MB of addressing space. The instruction set and addressing modes presented in this chapter first became available with the 8086, which operates in one mode only: real mode. Every processor in the 80x86 family to come after the 8086 supports the initial instruction set.

When the 8086 is reset (or first turned on), the processor fetches its first instruction from address FFFF0H. On the PC, this address points to boot instructions in the motherboard's system ROM, which begins the process of booting DOS. All of the 80x86 machines, even the Pentium, follow this mechanism when reset.

A slightly reengineered version of the 8086 was the 8088 microprocessor, which is similar to the 8086 except for the use of an external data bus that is only 8 bits wide. This forces the 8088 to access memory twice as often as the 8086, resulting in a significant performance penalty in terms of execution speed. Both the 8086 and the 8088 were used in the first PCs to hit the market.

Figure 2.8 compares the pin assignments of the 8088 with those of the 8086. As you can see, the majority of the pins are the same for both processors. Differences exist on address lines  $A_8$  through  $A_{15}$ . On the 8088, these lines are merely address lines. On the 8086, these are multiplexed address/data lines  $AD_8$  through  $AD_{15}$ . These lines carry  $A_8$  through



Note: ( ) denotes a MAX mode signal

FIGURE 2.8 8088 and 8086 pin assignments

A<sub>15</sub> during certain times, and D<sub>8</sub> through D<sub>15</sub> during others. This is where the other half of the 16-bit data bus comes in. The 8086 is capable of reading or writing 16-bits of data at once and for that reason has a slight speed advantage over the 8088.

Internally, both processors are almost identical. Any program written for the 8088 will run without change on the 8086, and vice versa. This means that the instruction sets of both machines are identical. The 8-bit version was a popular choice among manufacturers of personal computers and is therefore featured in this text, but keep in mind that much of what will be said about the 8088 also applies to the 8086.

Both processors have the capability of operating in **minimum mode** or **maximum mode**. Both modes have different ways of dealing with the external buses; thus we see (again in Figure 2.8) that some pins on the processors have two functions. For example, pin 29 on both machines is WR in minimum mode, and LOCK in maximum mode. The use of maximum mode requires some additional hardware outside the CPU to generate bus control signals. A minimum mode system requires a minimum of external hardware to implement a system.

It is useful to understand that the 8088 and 8086 are not two entirely different microprocessors. You may wish to think of the 8088 as an 8086 on the inside, with an 8-bit data bus on the outside.

A souped-up version of the 8086, called the 80186, contained special hardware such as programmable timers, counters, interrupt controllers, and address decoders. The 80186 was never used in the PC, but was ideal for custom systems that required a minimum of hardware.

## 2.10 A SUMMARY OF THE 80286

The next major improvement in Intel's line of microprocessors was the 80286 High-Performance Microprocessor with Memory Management and Protection™. The 80286 does not contain the internal DMA controllers, timers, and other enhancements of the 80186. Instead, the 80286 concentrates on the features needed to implement **multitasking**, an operating system environment that allows many programs or tasks to run seemingly simultaneously by allocating each program or task a small slice of time to execute, taking turns with time slices from all the other tasks. In fact, the 80286 was designed with multitasking in mind. A 24-bit address bus gives the processor the capability of accessing 16MB of storage. The internal memory management feature increases the storage space to 1 **gigabyte** of virtual address space. That's over 1 billion locations of virtual memory! Virtual addressing is a concept that has gained much popularity in the computing industry. Virtual memory allows a large program to execute in a smaller physical memory. For example, if a system using the 80286 contained 8MB of RAM, memory management and virtual addressing permits the system to run a program containing 12MB of code and data, or even multiple programs in a multitasking environment, *all of which* may be larger than 8MB.

To implement the complicated addressing functions required by virtual addressing, the 80286 has an entire functional unit dedicated to address generation. This unit is called the address unit. It provides two modes of addressing: 8086 real address mode and protected virtual address mode. The 8086 real address mode is used whenever an 8086 program executes on the 80286. The 1MB addressing space of the 8086 is simulated on the 80286 by the use of the lower 20 address lines. Processor registers and instructions are totally backward-compatible with the 8086.

Protected virtual address mode uses the full power of the 80286, providing memory management, additional instructions, and protection features, while at the same time retaining the ability to execute 8086 code. The processor switches from 8086 real address mode to protected mode when a special instruction sets the protection enable bit in the machine's status word. Addressing is more complicated in protected mode and is accomplished through the use of **segment descriptors** stored in memory. The segment descriptor is the device that really makes it possible for an operating system to control and protect memory. Certain bits within the segment descriptor are used to grant or deny access to memory in certain ways. A section of memory may be write protected or made to execute only by the setting of proper bits in the access rights byte of the descriptor. Other bits are used to control how the segment is mapped into virtual memory space and whether the descriptor is for a code segment or a data segment. Special descriptors, called **gate descriptors**, are used for other functions. Four types of gate descriptors are call gates, task gates, interrupt gates, and trap gates. They are used to change privilege levels (there are four), switch tasks, and specify interrupt service routines.

The instruction set of the 80286 is identical to that of the 8086, with additional instructions thrown in to handle the new features. Many of the instructions are used to load and store the different types of descriptors found in the 80286. Other instructions are used to manipulate task registers, change privilege levels, adjust the machine status word, and verify read/write accesses. Clearly, the 80286 differs greatly from the 8086 in the services it offers, while at the same time filling a great need for designers of operating systems.

---

## 2.11 A SUMMARY OF THE 80386

Intel continued its upward-compatible trend with the introduction of the 386 High Performance 32-bit CHMOS Microprocessor with Integrated Memory Management™. Software written for the 8088, 8086, 80186, and 80286 will also run on the 386. A 132-pin Grid Array™ package houses the 386, which offers a full 32-bit data bus and 32-bit address bus. The address bus is capable of accessing over 4 gigabytes of physical memory. Virtual addressing pushes this to over 64 *trillion* bytes of storage.

The register set of the 386 is compatible with earlier models, including all general purpose registers plus the segment registers. Although the general purpose registers are 16 bits wide on all earlier machines, they were extended to 32 bits on the 386. Their new names are EAX, EBX, ECX, and so on. Two additional 16-bit data segment registers are included, FS and GS. Like the 80286, the 386 has two modes of operation: real mode and protected mode. When in real mode, segments have a maximum size of 64KB. When in protected mode, a segment may be as large as the entire physical addressing space of 4 gigabytes. The new extended flags register contains status information concerning privilege levels, virtual mode operation, and other flags concerned with protected mode. The 386 also contains three 32-bit control registers. The first, machine control register, contains the machine status word and additional bits dealing with the coprocessor, paging, and protected mode. The second, page fault linear address, is used to store the 32-bit address that caused the last page fault. In a virtual memory environment, physical memory is divided up into a number of fixed size pages. Each page will, at some time, be loaded with a portion of an executing program or other type of data. When the processor determines that a page it needs to use has not been loaded into memory, a **page fault** is generated. The page fault instructs the processor to load the missing page into memory. Ideally, a low page-fault rate is desired.

The third control register, page directory base address, stores the physical memory address of the beginning of the page directory table. This table is up to 4KB in length and may contain up to 1024 page directory entries, each of which points to another page table area, whose information is used to generate a physical address.

The segment descriptors used in the 80286 are also used in the 386, as are the gate descriptors and the four levels of privilege. Thus, the 386 functions much the same as the 80286, except for the increase in physical memory space and the enhancements involving page handling in the virtual environment.

The computing power of each of the processors that have been presented can be augmented with the addition of a floating-point coprocessor. All sorts of mathematical operations can be performed with the coprocessors with 80-bit binary precision. The 8087 coprocessor is designed for use with the 8088 and 8086, the 80287 with the 80286, and the 80387 with the 386.

## 2.12 A SUMMARY OF THE 80486

This processor is the next in Intel's upward-compatible 80x86 architecture. Surprisingly, there are only a few differences between the 80486 and the 80386, but these differences create a significant performance improvement.

Like the 80386, the 80486 is a 32-bit machine containing the same register set as the 80386 and all of the 80386's instruction set with a few additional instructions. The 80486 has a similar 4-gigabyte addressing space using the same addressing features.

The first improvement over the 80386 is the addition of an 8KB **cache** memory. A cache is a very high-speed memory, with an access time usually ten times faster than that of conventional RAM used for external processor memory. The 80486's internal cache is used to store both instructions and data. Whenever the processor needs to access memory, it will first look in the cache for it. If the data is found in the cache, they are read out much faster than if they had to come from external RAM or EPROM. This is known as a **cache hit**. If the data is not found in the cache, the processor must then access the slower external memory. This is called a **cache miss**. The processor tries to keep the cache's hit ratio as high as possible. Consider the following example:

$$\text{RAM Access Time} = 70 \text{ ns}$$

$$\text{Cache Access Time} = 10 \text{ ns}$$

$$\text{Hit Ratio} = 0.85$$

$$\begin{aligned} \text{Average Memory Access Time} &= 0.85 \times (10 \text{ ns}) \text{ Hit} \\ &\quad + (1 - 0.85) \times (10 \text{ ns} + 70 \text{ ns}) \text{ Miss} \\ &= 20.5 \text{ ns} \end{aligned}$$

The average memory access time for a hit ratio of 0.85 is less than 21 ns! This is due to the following reasoning: If data is found in the cache (85% of the time), the access time is only 10 ns. If data is not found (15% of the time), the access time equals 80 ns (the cache access time plus the RAM access time), because the processor had to read the cache to find out the data was *not* there.

If you consider that a large portion of a program (or even an *entire* program) might fit within the 8KB cache, you will agree that the program will execute very quickly, because most instruction fetches will be for code already in the cache. This architectural improvement significantly increases the processing speed of the 80486. Some of the new 80486 instructions are included to help maintain the cache.

The 80486 has two other improvements. Although it executes the same instruction set as the 80386, the 80486 does so with a redesigned internal architecture. This new design allows many 80486 instructions to execute with fewer clock cycles than those required by the 80386. This reduction in clock cycles adds additional speed to the 80486's execution. Also, the 80486 comes with an on-chip math coprocessor or FPU (floating-point unit). You might recall that the 80386 can be connected to an external 80387 math coprocessor to enhance performance. The 80486 has the equivalent of an 80387 built right into it! And because the FPU is closer to the CPU, data is transferred quicker, which leads to another performance boost. The two versions of the 80486 are the 80486 SX and the 80486 DX. The 80486 SX has its internal FPU disabled. The 80486 DX has its FPU enabled.

Although the 80386 and 80486 share many similarities, the 80486's differences create a much more powerful processor.

---

## 2.13 A SUMMARY OF THE PENTIUM

The Pentium will run all programs written for any machine in the 80x86 line, though it does so at speeds many times that of the fastest 80486 (from 60 MHz up to 233 MHz). And the Pentium does so with a radically new architecture!

There are two major computer architectures in use: CISC and RISC. CISC stands for Complex Instruction Set Computer. RISC stands for Reduced Instruction Set Computer. All of the 80x86 machines prior to the Pentium can be considered CISC machines. The Pentium itself is a mixture of both CISC and RISC technologies. The CISC aspect of the Pentium provides for upward compatibility with the other 80x86 architectures. The RISC aspects lead to additional performance improvements. Some of these improvements are separate 8KB data and instruction caches, dual integer pipelines, and branch prediction.

As Figure 2.9 shows, the Pentium processor is a complex machine with many interlocking parts. At the heart of the processor are the two integer pipelines, the U pipeline and the V pipeline. These pipelines are responsible for executing 80x86 instructions. A floating-point unit is included on the chip to execute instructions previously handled by the external 80x87 math coprocessors. During execution, the U and V pipelines are capable of executing two integer instructions at the same time, under special conditions, or one floating-point instruction.

The Pentium communicates with the outside world via a 32-bit address bus and a 64-bit data bus. The bus unit is capable of performing burst read and writes of 32 bytes to memory, and through bus cycle pipelining, allows two bus cycles to be in progress simultaneously.

An 8KB instruction cache is used to provide quick access to frequently used instructions. When an instruction is not found in the instruction cache, it is read from the external data bus and a copy placed into the instruction cache for future references. The branch target buffer and prefetch buffers work together with the instruction cache to fetch instructions as fast as possible. The prefetch buffers maintain a copy of the next 32 bytes of prefetched instruction code and can be loaded from the cache in a single clock cycle, due to the 256-bit-wide data output of the instruction cache.

The Pentium uses a technique called branch prediction to maintain a steady flow of instructions into the pipelines. To support branch prediction, the branch target buffer maintains a copy of instructions in a different part of the program located at an address called the branch target. For example, the branch target of a CALL XYZ instruction is the address

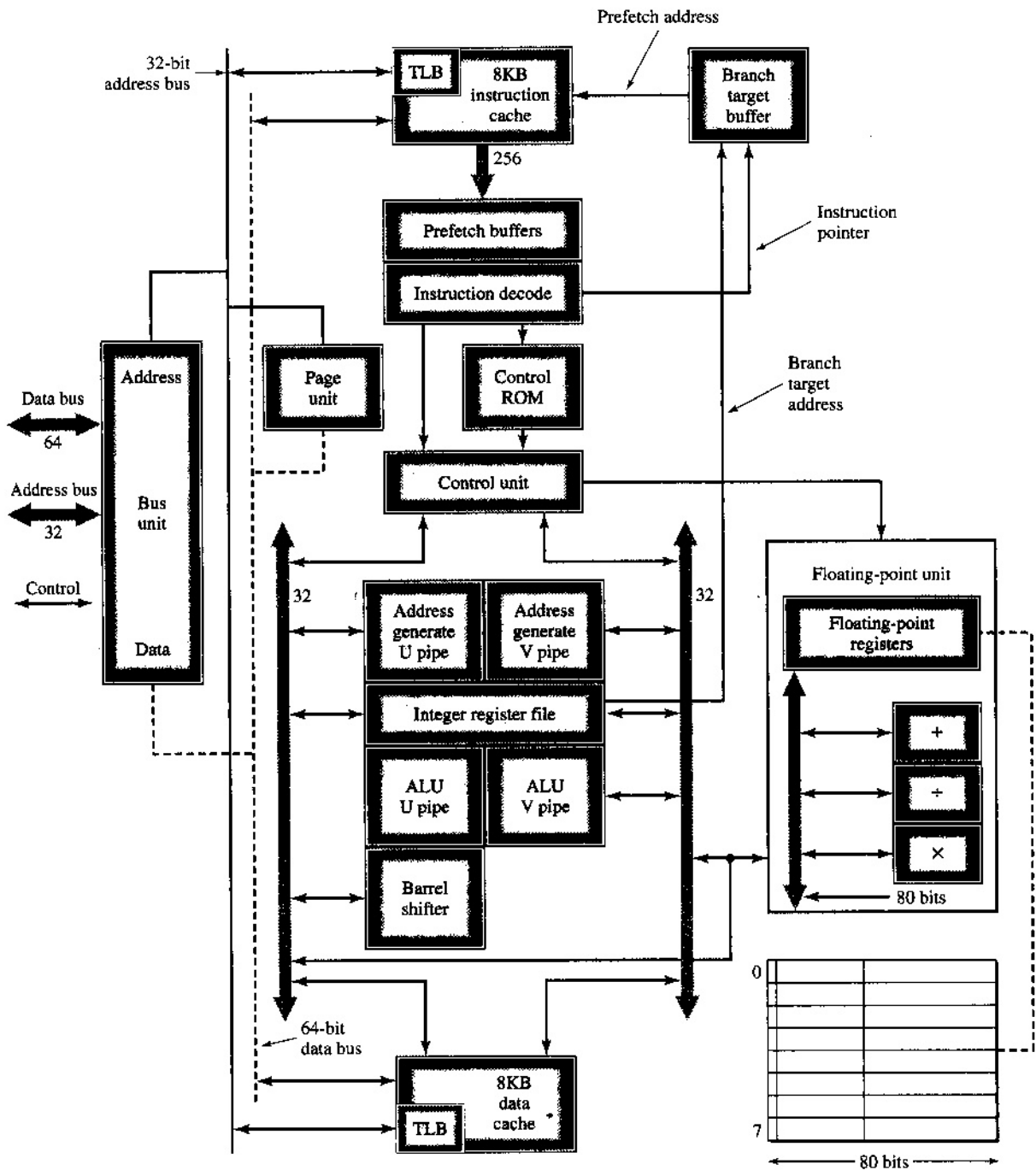


FIGURE 2.9 Pentium architecture block program

of the subroutine XYZ. Certain instructions, such as CALL, may cause the processor to jump to an entirely different program location, instead of simply proceeding to the instruction in the next location. So, just in case the code from the target address is needed, the branch target buffer maintains a copy of it and feeds it to the instruction cache.



A separate 8KB data cache stores a copy of the most frequently accessed memory data. Because memory accesses are significantly longer than processor clock cycles, it pays to keep a copy of memory data in a fast-reading cache. The data and instruction caches may both be enabled/disabled with hardware or software. Both also employ the use of a translation lookaside buffer, which converts logical addresses into physical addresses when virtual memory is employed.

The floating-point unit of the Pentium maintains a set of floating-point registers and provides 80-bit precision when performing high-speed math operations. This unit has been completely redesigned from the one used inside the 80486 and is also pipelined. The floating-point unit uses hardware in the U and V pipelines to perform the initial work during a floating-point instruction (such as fetching a 64-bit operand), and then uses its own pipeline to complete the operation. Because both integer pipelines are used, only one floating-point instruction may be executed at a time.

Altogether, the Pentium processor includes many features designed to increase performance over earlier 80x86 machines. This was possible by blending CISC and RISC technology together. The benefit to programmers is that all Intel processors from the 8086 up, including the Pentium, run the same basic instruction set, which we will learn about beginning in the next chapter, but they do it faster and faster. In Chapter 15, we will examine the improvements made to the Pentium's architecture with looks at the Pentium II, III, and IV CPUs, among others.

---

## 2.14 TROUBLESHOOTING TECHNIQUES

A great amount of material was presented in this chapter regarding the Intel 80x86 architecture. It would be helpful to really commit some of the most basic material about the 80x86 to memory. At a minimum, you should be able to do the following without much thought:

- Name all of the processor registers, their bit sizes, and whether they can be split into 8-bit halves.
- Be familiar with several architectural features, such as the processor's addressing space (1MB in real mode), interrupt mechanism, and I/O mechanism.
- Show what is meant by Intel byte-swapping (little-endian format).
- List the names and meanings of the most common flags, such as zero, carry, and sign.
- Describe the method used to form a 20-bit address in real mode (combining a segment register with an offset).
- Show how an instruction is composed of an operation, a set of operands, and a particular addressing mode.

Knowing these basics thoroughly will assist you in mastering the 80x86 instruction set that we will examine in Chapters 3 and 4.

---

## SUMMARY

This chapter has taken an introductory look at the 80x86 family of upward-compatible microprocessors. The software model of the 80x86 was examined first, showing all the 16-bit general purpose registers (AX, BX, CX, DX, SI, DI, and BP) and the four 16-bit segment

registers (CS, DS, SS, and ES), as well as the 32-bit extended registers beginning with the 80386.

Although 80x86 real mode contains only 16-bit registers, its architecture allows the generation of 20-bit physical addresses, giving the processor a 1-megabyte addressing space. One of the segment registers is always involved in a memory access. The general purpose registers were shown to have specific tasks assigned to them by default, such as the use of AX in multiply and divide operations. Techniques to use 32-bit registers and addressing modes were also demonstrated.

A technique called Intel byte-swapping was also introduced, which accounts for the way a 16-bit number is stored in memory (low byte first). This technique is rarely seen on other microprocessors.

The entire instruction set was presented to give you a feel for the type of operations the 80x86 is capable of performing. This discussion was followed by a brief explanation of what a segment is, and what addressing modes are available. Some examples were shown to illustrate the use of different addressing modes. This was followed by an explanation of the processor's interrupt structure, and a summary of the entire 80x86 family.

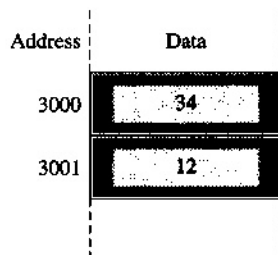
In the next chapter we will take a detailed look at the software operation of the 80x86.

---

## STUDY QUESTIONS

1. Name all of the general purpose registers and some of their special functions.
2. How are the segment registers used to form a 20-bit address?
3. (a) If CS contains 03E0H and IP contains 1F20H, from what address is the next instruction fetched?  
(b) If SS contains 0400H and SP contains 3FFEh, where is the top of the stack located?  
(c) If a data segment begins at address 24000H, what is the address of the last location in the segment?
4. Explain what the instruction and data caches are used for.
5. Are the U and V pipelines identical in operation?
6. (a) Show the DB statement needed to define a list of numbers called FACTORS that contains all the integer factors of the decimal value 50.  
(b) Show how a DW statement can be written to reserve 250 words of the value 7.
7. What is a segment?
8. Two memory locations, beginning at address 3000H, contain the bytes 34H and 12H. What is the word stored at location 3000H? See Figure 2.10 for details.

**FIGURE 2.10** For Question 8



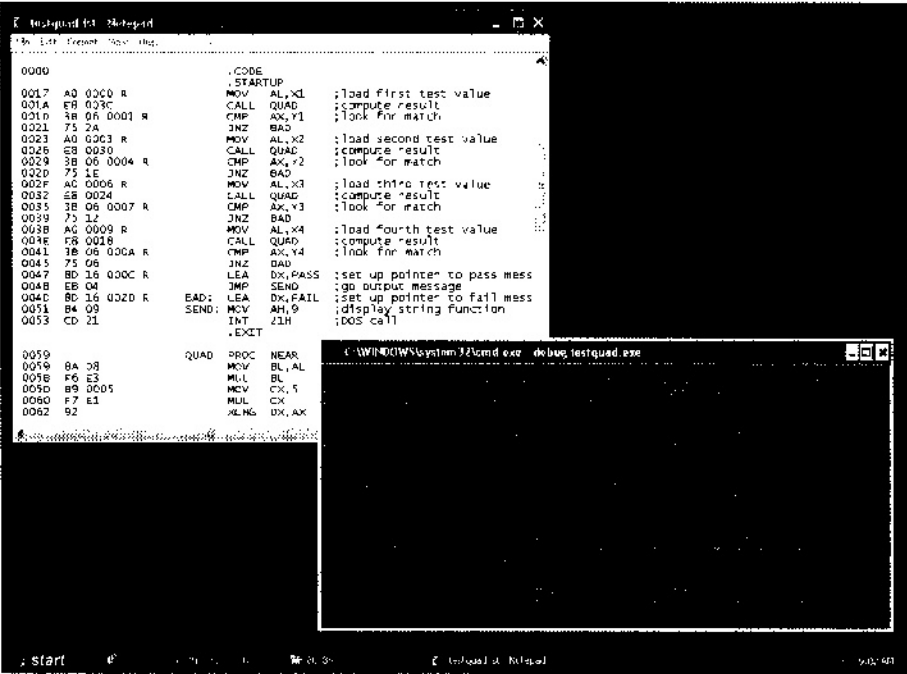
9. What is Intel byte-swapping?
10. Count the number of different instructions available. How many are there?
11. How many addressing modes are provided?
12. What is a physical address?
13. Why is register addressing so fast?
14. What do square brackets mean when they appear in an operand (e.g., MOV AX, [3000])?
15. What is the difference between MOV AX,1000H and MOV AX,[1000H]?
16. How does port addressing differ from memory addressing?
17. What is an interrupt?
18. Name one instruction that can cause an interrupt.
19. How many interrupts does the 80x86 support?
20. What are some of the differences between real mode and protected mode?
21. List the important features of the 80286.
22. What is one advantage of virtual memory?
23. What is a page fault?
24. Compare two 386 systems, one containing 512KB of RAM, the second containing 4 megabytes. How would the number of page faults compare when:
  - (a) a 220KB application is executed on both machines?
  - (b) a 6-megabyte application is executed on both machines?
25. Which has the greater effect on the number of page faults, physical memory size or the size of the program being executed?
26. Why would we resist building a complete physical memory for the 386? Does the reason apply to the 8086?
27. Why would anyone possibly need 4.3 billion bytes for a program? Can you think of any applications that may require this much memory?
28. List three differences between the 80286 and the 80386.
29. List three differences between the 80386 and the 80486.
30. Compute the average memory access time from the following information:

RAM Access Time = 80 ns  
Cache Access Time = 10 ns  
Hit Ratio = 0.92
31. What makes the Pentium so different from other 80x86 CPUs?
32. Use DEBUG to find the address of INT 21H on your DOS machine.

# PART 2

## Software Architecture

- 3 80x86 Instructions, Part 1: Addressing Modes, Flags, Data Transfer, and String Instructions
- 4 80x86 Instructions, Part 2: Arithmetic, Logical, Bit Manipulation, Program Transfer, and Processor Control Instructions
- 5 Interrupt Processing



Microsoft® Windows® desktop showing a program under development

---

## CHAPTER 3

---

# 80x86 Instructions, Part 1: Addressing Modes, Flags, Data Transfer, and String Instructions

---

### OBJECTIVES

In this chapter you will learn about:

- The style of source files written in 80x86 assembly language
- The different addressing modes of the 80x86
- The operation and use of the processor flags
- Data transfer and string instructions

### KEY TERMS

Assembler	Linker	Segment
Condition code	List file	Segment override prefix
Cross-assembler	Loader	Sign bit
Direction flag	Machine code	Source file
Effective address	Module	Stack
Flag	Object file	Zero flag
Immediate data	Pseudo-opcode	
Library	Repeat prefix	

---

## 3.1 INTRODUCTION

This chapter introduces you to the first part of the instruction set of the 80x86 microprocessor family, and the ways that different addressing modes and data types can be used to make the instructions do the most work for you. The combination of instructions and addressing modes found in the 80x86 makes the job of writing code much easier and more efficient than before. In this chapter we will examine the various addressing modes, flags,

and conventions used when representing data. We will take a detailed look at the data transfer and string instructions, leaving the remainder of the instruction set for Chapter 4.

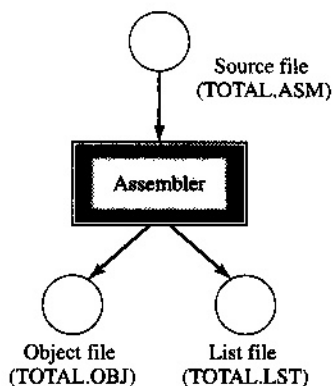
Section 3.2 introduces the conventions followed when writing 80x86 assembly language source code. Section 3.3 explains the different instruction types available; this is followed by coverage of the 80x86's addressing modes in Section 3.4. Processor flags are detailed in Section 3.5 to set the stage for the first two instruction groups, data transfer and string instructions, which are presented in Sections 3.6 and 3.7, respectively. Troubleshooting techniques are presented in Section 3.8.

## 3.2 ASSEMBLY LANGUAGE PROGRAMMING

Program execution in any microprocessor system consists of fetching binary information from memory and decoding that information to determine the instruction represented. The information in memory may have been programmed into an EPROM or downloaded from a separate system. But where did the program come from and how was it written? As humans, we have trouble handling many pieces of information simultaneously and thus have difficulty writing programs directly in **machine code**, the binary language understood by the microprocessor. It is much easier for us to remember the mnemonic `SUB AX,AX` than the corresponding machine code `2BC0`. For this reason, we write **source files** containing all the instruction mnemonics needed to execute a program. The source file is converted into an **object file**, containing the actual binary information the machine will understand, by a special program called an **assembler**. Some assemblers allow the entire source file to be written and assembled at one time. Other assemblers, called single-line assemblers, work with one source line at a time and are restricted in operation. The `DEBUG` utility that comes with DOS and Windows contains a built-in, single-line assembler.

The assembler discussed here is not a single-line assembler. Instead it is a **cross-assembler**. Cross-assemblers are programs written in one language, such as C, that translate source statements into a second language: the machine code of the desired processor. Figure 3.1 shows this translation process. The source file in the example, `TOTAL.ASM`, is presented as input to the assembler. The assembler will convert all source statements into the correct binary codes and place these into the object file `TOTAL.OBJ`. Usually, the object file contains additional information concerning program relocation and external

**FIGURE 3.1** Source program assembly



references, and thus is not yet ready to be loaded into memory and executed. A second file created by the assembler is the **list file**, TOTAL.LST, which contains all the original source file text plus the additional code generated by the assembler. The list file may be displayed on the screen, or printed. The object file may not be printed or displayed, since it is just code.

## A Sample Source File

Let us look at a sample source file, a subroutine designed to find the sum of 16 bytes stored in memory. It is not important at this time that you understand what each instruction does. We are simply trying to get a feel for what a source file might look like and what conventions to follow when we write our own programs.

```

                                ORG 8000H
TOTAL:    MOV AX,7000H      ;load address of data area
                                MOV DS,AX      ;init data segment register
                                MOV AL,0        ;clear result
                                MOV BL,16       ;init loop counter
                                MOV SI,0        ;init data pointer
ADDUP:    ADD AL,[SI]       ;add data value to result
                                INC SI          ;increment data pointer
                                DEC BL          ;decrement loop counter
                                JNZ ADDUP        ;jump if counter not zero
                                MOV [SI],AL     ;save sum
                                RET              ;and return
                                END

```

The first line of source code contains a command that instructs the assembler to load its program counter with 8000H. The ORG (for origin) command is known as an assembler **pseudo-opcode**, a fancy name for a mnemonic that is understood by the assembler but not by the microprocessor. ORG does not generate any source code; it merely sets the value of the assembler's program counter. This is important when a section of code must be loaded at a particular place in memory. The ORG statement is a good way to generate instructions that will access the proper memory locations when the program is loaded into memory.

Hexadecimal numbers are followed by the letter H to distinguish them from decimal numbers. This is necessary since 8000 decimal and 8000 hexadecimal differ greatly in magnitude. For the assembler to tell them apart, we need a symbol that shows the difference. Some assemblers use \$8000; others use &H8000. It is really a matter of which software you purchase. All examples in this book will use the 8000H form.

The second source line contains the major components normally used in a source statement. The label TOTAL is used to point to the address of the first instruction in the subroutine. ADDUP is also a label. Single-line assemblers do not allow the use of labels.

The opcode is represented by MOV and the operand field by AX,7000H. The order of the operands is <destination>, <source>. This indicates that 7000H is being MOVED into AX. So far we have three fields: label, opcode, and operand. The fourth field, if it is used, usually contains a comment explaining what the instruction is doing. Comments are preceded by a semicolon (;) to separate them from the operand field. In writing source code, you should follow the four-column approach. This will result in a more understandable source file.

The final pseudo-opcode in most source files is END. The END statement informs the assembler that it has reached the end of the source file. This is important, because many

assemblers usually perform two passes over the source file. The first pass is used to determine the lengths of all instructions and data areas and to assign values to all symbols (labels) encountered. The second pass completes the assembly process by generating the machine code for all instructions, usually with the help of the symbol table created in the first pass. The second pass also creates and writes information to the list and object files. The list file for our example subroutine looks like this:

1	8000				ORG	8000H
2	8000	B8	0070	TOTAL:	MOV	AX,7000H
3	8003	8E	D8		MOV	DS,AX
4	8005	B0	00		MOV	AL,0
5	8007	B3	10		MOV	BL,16
6	8009	BE	0000		MOV	SI,0
7	800C	02	04	ADDUP:	ADD	AL,[SI]
8	800E	46			INC	SI
9	800F	FE	CB		DEC	BL
10	8011	75	F9		JNZ	ADDUP
11	8013	88	04		MOV	[SI],AL
12	8015	CB			RET	
13					END	

Normally the comments would follow the instructions, but they have been removed for the purposes of this discussion.

The first column of numbers represents the original source line number.

The second column of numbers represents the memory addresses of each instruction. Notice that the first address matches the one specified by the ORG statement. Also notice that the ORG statement does not generate any code.

The third column of numbers is the machine code generated by the assembler. The machine codes are intermixed with data and address values. For example, B8 00 70 in line 2 represents the instruction MOV AX,7000H, with the MOV instruction coded as B8 and the immediate word 7000H coded in byte-swapped form as 00 70. In line 3, the machine codes 8E D8 represent MOV DS,AX. Neither of these 2 bytes are data or address values, as they were in line 2. Look for other data values in the instructions on lines 4 through 6. Line 5 makes an important point: the assembler will convert decimal numbers into hexadecimal (the 16 in the operand field has been converted into 10 in the machine code column).

Finally, another look at the list file shows that there are 1-, 2-, and 3-byte instructions present in the machine code. When an address or data value is used in an instruction, chances are good that you will end up with a 2- or 3-byte instruction (or possibly even more).

Following the code on each line is the text of the original source line. Having all of this information available is very helpful during the debugging process.

### Assembler Directives ORG, SEGMENT, ENDS, ASSUME, and END

The actual form of an 80x86 source file is much more complicated than the simple example we have just examined. When writing 80x86 source files, we separate code areas from data areas. We may even have a separate area reserved for the stack. The new source file for TOTAL, which includes separate areas called **segments**, contains many new pseudocodes to help the assembler generate the correct machine code for the object file.



```

DATA      SEGMENT  PARA 'DATA'
          ORG      7000H
POINTS    DB      16 DUP(?)      ;save room for 16 data bytes
SUM       DB      ?              ;save room for result
DATA      ENDS

CODE      SEGMENT  PARA 'CODE'
          ASSUME   CS:CODE,DS:DATA

          ORG      8000H
TOTAL:    MOV      AX,DATA        ;load address of data segment
          MOV      DS,AX          ;init data segment register
          MOV      AL,0           ;clear result
          MOV      BL,16          ;init loop counter
          LEA      SI,POINTS      ;init data pointer
ADDUP:    ADD      AL,[SI]        ;add data value to result
          INC      SI             ;increment data pointer
          DEC      BL             ;decrement loop counter
          JNZ      ADDUP          ;jump if counter not zero
          MOV      SUM,AL         ;save sum
          RET                    ;and return

CODE      ENDS
          END      TOTAL

```

In this new source file we use a DATA segment and a CODE segment. These areas are defined by the SEGMENT pseudocode and terminated by ENDS (for end segment). Note that the SEGMENT assembler directive does *not* affect the value in any segment register. The TOTAL subroutine is placed inside the CODE segment. In the DATA segment, 16 bytes of storage are reserved with the DB 16 DUP(?) instruction. DB stands for define byte, and DUP for duplicate. The ? means we do not know what value to put into memory and is a way of telling the assembler that we do not care what byte values are used in the reserved data area. We could easily use DB 16 DUP(0) to place 16 zeros into the reserved area. Words can also be defined/reserved in a similar fashion, using, for example, DW 8 DUP(0) or DW 8 DUP(?). The ORG 7000H statement tells the assembler where to put the data areas. It is not necessary for this ORG value to be smaller than the ORG of the subroutine. ORG 87C0H would have also worked in place of ORG 7000H. It is all a function of where RAM exists in your system.

The addition of the TOTAL label in the END statement informs the assembler that TOTAL, not SUM, is the starting execution address. This information is also included in the object file.

Most assemblers now accept simplified segment directives and automatically generate the necessary code to manage segments. The TOTAL source file, rewritten with simplified segment directives, now looks like this:

```

          .MODEL    SMALL
          .DATA
          ORG      7000H
POINTS    DB      16 DUP(?)      ;save room for 16 data bytes
SUM       DB      ?              ;save room for result

          .CODE
          ORG      8000H
TOTAL:    MOV      AX,7000H      ;load address of data area
          MOV      DS,AX        ;init data segment register

```

```

                MOV     AL,0           ;clear result
                MOV     BL,16          ;init loop counter
                LEA      SI,POINTS      ;init data pointer
ADDUP:          ADD     AL,[SI]        ;add data value to result
                INC     SI             ;increment data pointer
                DEC     BL             ;decrement loop counter
                JNZ      ADDUP          ;jump if counter not zero
                MOV     SUM,AL         ;save sum
                RET                     ;and return
                END      TOTAL

```

The first directive, `.MODEL`, instructs the assembler that the type of program being created falls into a category called `SMALL`. Small programs are programs that contain one code segment and one data segment. All of the programs in this book are small programs. Other models allow for multiple code segments, multiple data segments, or both, as indicated in Table 3.1.

The `.DATA` directive indicates the beginning of a data segment. There is no need to indicate the end of the data segment. This is automatically assumed when the `.CODE` directive is encountered, which begins the code segment. Clearly, it is easier for the programmer to use these simplified segment directives, rather than get bogged down with the minute details of assembler syntax. The remaining programs in the book will utilize simplified segment directives.

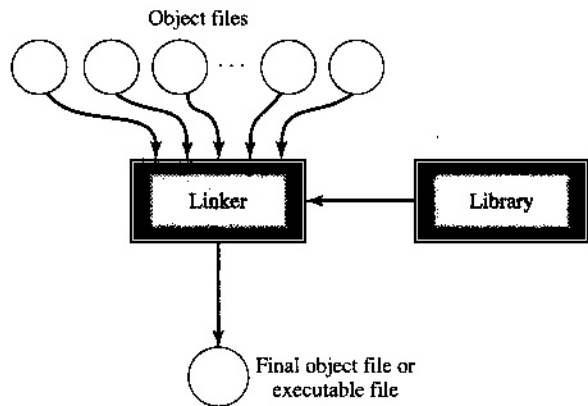
When a large program must be written by a team of people, each person will be assigned a few subroutines to write. They must all assemble and test their individual sections to ensure the code executes correctly. When all portions of the program (called **modules**, after a technique called modular programming) are assembled and tested, their object files are combined into one large object file via a program called a **linker**. Figure 3.2 represents this process. The linker examines each object file, determining its length in bytes, its proper place in the final object file, and what modifications should be made to it.

In addition, a special collection of object files is sometimes available in a **library** file. The library may contain often-used subroutines or patches of code. Instead of continuously reproducing these code segments in a source file, a special pseudocode is used to instruct the assembler that the code must be brought in at a later time (by the linker). This helps keep the size of the source file down and promotes quicker writing of programs.

When the linker is through, the final code is written to a file called the load module. Another program called a **loader** takes care of loading the program into memory. Usually the linker and loader are combined into a single program called a link-loader.

**TABLE 3.1** Predefined `.MODEL` types

<i>Memory Model</i>	<i>Data Segments</i>	<i>Code Segments</i>	<i>Special Features</i>
TINY	one	one	Only used for <code>.COM</code> file. CS and DS combined.
SMALL	one	one	Smallest <code>.EXE</code> file model.
MEDIUM	one	multiple	
COMPACT	multiple	one	
LARGE	multiple	multiple	
HUGE	multiple	multiple	Uses normalized addresses.
FLAT	one	one	32-bit addressing.

**FIGURE 3.2** Linking multiple object files together

So, writing the source file is actually only the first step in a long process. But even before a source file can be written, the programmer must understand the instructions that will be used in the source file. The remaining sections will begin coverage of this important topic.

### 3.3 INSTRUCTION TYPES

For purposes of discussion in this section, the instruction set of the 80x86 microprocessor is divided into seven different groups:

1. Data transfer
2. Strings
3. Arithmetic
4. Bit manipulation
5. Loops and jumps
6. Subroutine and interrupt
7. Processor control

The data transfer group contains instructions that transfer data from memory to register, register to register, and register to memory. Instructions that perform I/O are also included in this group. Data may be 8, 16, or 32 bits in length, and all of the processors registers may be used (including the stack pointer and flag register).

The next group deals with strings. A string is simply a collection of bytes stored sequentially in memory, whose length can be up to 64KB. Instructions are included in this group to move, scan, and compare strings.

The arithmetic group provides addition and subtraction of 8-, 16-, and 32-bit values, signed and unsigned multiplication and division of 8-, 16-, 32-, and 64-bit numbers, and special instructions for working with BCD and ASCII data.

The bit manipulation group is used to perform AND, OR, XOR (exclusive -OR), and NOT (1's complement) operations on 8-, 16-, and 32-bit data contained in registers or memory. Shift and rotate operations on bytes and words are also included, with single or multi-bit shifts or rotates possible.

Loops and jumps are contained in the next group. Many different forms of instructions are available, with each one testing a different condition based on the state of the processor's flags. The loop instructions are designed to repeat automatically, terminating when certain conditions are met.

The subroutine and interrupt group contains instructions required to call and return from subroutines and handle interrupts. The processor stack can be manipulated by a special form of the return instruction, and two classes of subroutines, called near and far procedures, are handled by these instructions.

The final group of instructions is used to directly control the state of some of the flags, enable/disable the 80x86 interrupt mechanism, and synchronize the processor with external peripherals.

Many different addressing modes can be used with most instructions, and, in the next section, we will examine these addressing modes in detail.

---

## 3.4 ADDRESSING MODES

The power of any instruction set is a function of both the types of instructions implemented and the number of addressing modes available. Suppose that a microprocessor could not directly manipulate data in a memory location. The data would have to be loaded into a processor register, manipulated, and written back into memory. If an addressing mode were available that could directly operate on data in memory, the task could be done more effectively. In this section, we will examine the many different addressing modes available in the 80x86 and how each is used. The examples presented make use of the MOV instruction, which has the following syntax: MOV <destination>,<source>. It will be obvious in the examples what is being accomplished with each MOV instruction, so a detailed description is not included here. Also, whenever the contents of a memory location or an address or data register is referred to, assume that the value or address is hexadecimal.

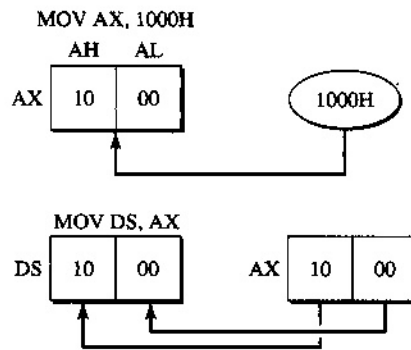
### Creating an Effective Address

Chapter 2 introduced the concept of segmented memory, where a segment is a 64KB block of memory accessed with the aid of one of the six segment registers. Whenever *any* real-mode address is generated by the 80x86, the 20-bit value that appears on the processor's address bus is formed by some combination of segment register and an additional numerical offset. This address is referred to as the **effective address**. Many of the data transfer instructions use the data segment register by default when forming an effective address. As we saw in Chapter 2, instruction fetches generate effective addresses via the addition of the code segment register and the instruction pointer. Stack operations automatically use the stack segment register and the stack pointer. Thus, from a programming standpoint, almost all instructions require proper initialization of a segment register for correct execution and generation of effective addresses.

One way to initialize a segment register is to first place the desired segment address into the AX register and then copy the contents of AX into the corresponding segment register. For example, to initialize the data segment register to 1000H, we would use the following instructions:

```
MOV  AX,1000H
MOV  DS,AX
```

Remember that the direction of data transfer in the operand field is from right to left, so 1000H is MOVED into AX, and then the contents of AX are MOVED into DS, as you can see in Figure 3.3. (Unfortunately, MOV DS,1000H is an illegal instruction. We are *not allowed* to move a numerical value directly into a segment register!)

**FIGURE 3.3** Initializing the data segment register

All of the examples used to explain the processor's addressing modes assume that the corresponding segment register has already been initialized.

### Immediate Addressing

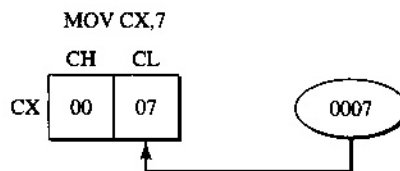
We often use **immediate data** in the operand field of an instruction. Immediate data represents a constant that we wish to place somewhere, by MOVing it into a register or a memory location. In `MOV AX, 1000H`, we are placing the immediate value 1000H into register AX. Immediate data is represented by 8-, 16-, or 32-bit numbers that are coded directly into the instruction. For example, `MOV AX, 1000H` is coded as B8 00 10, and `MOV AL, 7` is coded as B0 07. Notice in the first instruction that the 16-bit value 1000H has its lower byte (00) and its upper byte (10) reversed in the actual machine code. As we saw in Chapter 2, this technique is commonly referred to as Intel byte-swapping, and it is the way all 16-bit numbers are stored on Intel CPUs. This technique is also called little-endian format, because the little end (lower byte) of the two-byte value is stored first.

An important question at this time is "How did the assembler know that the 7 in `MOV AL, 7` should be coded as the single byte value 07, and not the 2-byte value 07 00 (in byte-swapped form)?" The answer is that the assembler looked at the size of the other operand in the instruction (AL). Because AL is the lower half of the AX register, the assembler knows that it may contain only 8 bits of data. Knowing this, do you see why `MOV AL, 9C8H` would be an illegal instruction? If you cannot answer this question, think about how many bits are needed to represent the hexadecimal number 9C8. Because 12 bits are needed, we cannot possibly store 9C8H in the 8-bit AL register.

We will now apply this in an example.

#### ■ EXAMPLE 3.1 What is the result of `MOV CX, 7`?

**Solution:** Because register CX is specified, the immediate value 7 is coded as the 2-byte number 00 07. Figure 3.4 shows how this 2-byte number is placed into CX.

**FIGURE 3.4** An example of immediate addressing

The machine code for `MOV CX, 7` is B9 07 00. ■

Here are some additional examples of immediate addressing:

```
MOV  AL,20      ;put 20 decimal into AL
ADD  BH,20H     ;add 20 hexadecimal to BH
SHL  DX,1       ;shift DX left one bit
AND  AH,40H     ;and AH with 40 hexadecimal
SUB  EAX,1      ;subtract 1 from EAX
```

## Register Addressing

The operand field of an instruction in many cases will contain one or more of the internal registers. Register addressing is the name we use to refer to operands that contain one of these registers. The `MOV DS,AX` instruction from the previous section is an example of register addressing, because we are using only processor registers in the operand field. Instructions of this form often execute very quickly, because they do not require any memory accesses beyond the initial instruction fetch cycles. The machine code corresponding to `MOV DS,AX` is 8E D8, a 2-byte instruction containing all the information necessary for the processor to perform the desired operation. `DEC DX` (decrement the DX register) is another example of register addressing, and it has 4A as its corresponding machine code.

### ■ EXAMPLE 3.2

If register AX contains the value 1000H and register BL contains 80H, what is the result of `MOV AL,BL`?

**Solution:** The contents of the BL register are copied into the AL register, leaving AH undisturbed. The final value in AX is 1080H. Notice that the contents are *copied* during the `MOV`. `MOV` can also be interpreted as “make a copy of.” So, when we move data from one place to another, we actually are just making a second copy of the source data.

The machine code for `MOV AL,BL` is 88 D8. ■

It is important to note that only the code segment register is needed to execute the instructions used in Examples 3.1 and 3.2.

Here are some additional examples of register addressing:

```
PUSH  AX        ;save copy of AX on stack
ADD   BH,CL     ;add CL to BH, result in BH
XCHG  BX,CX     ;exchange BX and CX
MUL   DL        ;multiply AL by DL
DIV   EBX       ;divide EDX:EAX by EBX
```

## Direct Addressing

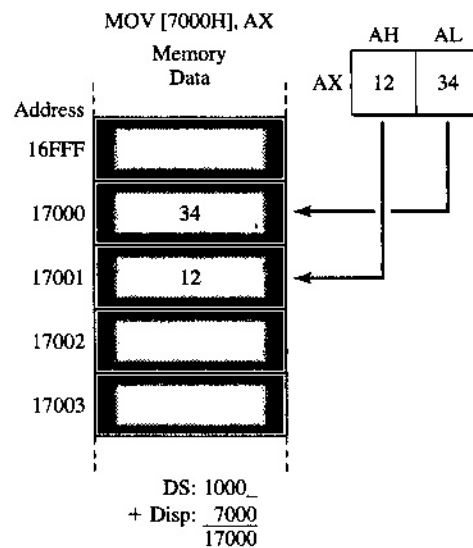
In this addressing mode, the effective address is formed by the addition of a segment register and a displacement value that is coded directly into the instruction. The displacement is usually the address of a memory location *within a specific segment*. One way to refer to a memory location in the operand field is to surround the memory address with square brackets. The instruction `MOV [7000H],AX` illustrates this concept. The 7000H is not thought of as immediate data here, but rather as address 7000H within the segment pointed to by the DS register. DS is the segment register used by the processor whenever brackets ([ ]) are used in the operand field (unless we override the use of DS by specifying a different segment register). A detailed example should serve to explain this addressing mode more clearly.

### ■ EXAMPLE 3.3

What is the result of `MOV [7000H],AX`? Assume that the DS register contains 1000H, and AX contains 1234H.

**Solution:** Examine Figure 3.5 very carefully. Remember that when the 80x86 uses a segment register to form an effective address, it shifts the segment register 4 bits to the left (turning 1000H into 10000H) and then adds the specified 16-bit displacement (or offset). Thus, the effective address generated by the processor is 17000H. Because register AX is used in the operand field, 2 bytes will be written into memory, with the lower byte (34) going into the first location (17000H) and the upper byte (12) going into the second location (17001H). Once again, we can see that the processor has byte-swapped the data as they were written into memory.

**FIGURE 3.5** An example of direct addressing



The machine code for `MOV [7000H],AX` is A3 00 70. ■

Another form of direct addressing requires the use of a label in the operand field. The sample program `TOTAL` that we examined in Section 3.2 used direct addressing in the `MOV SUM,AL` instruction. Another look at the `TOTAL` source file should convince you that the `SUM` label represents a displacement value within the segment pointed to by the DS register.

The automatic use of the DS register for memory accesses can be overridden by the programmer by specifying a different segment register within the operand field. If we wish to use the extra segment register in the instruction of Example 3.3, we would write `MOV ES:[7000H],AX`. The machine code required to allow the use of the ES register in this instruction is 26 A3 00 70. Note the similarity to the machine code in Example 3.3.

The first byte in the instruction, 26H, is called a **segment override prefix**. This byte instructs the processor to use the extra segment instead of the default data segment. Table 3.2 lists the override prefixes for each segment.

Here are some additional examples of direct addressing:

```
ADD AL,[4000]    ;add contents of location 4000 to AL
OR  [440H],BL    ;or contents of location 440H with BL
DEC COUNT       ;decrement value stored at COUNT
ADD SPIN,6       ;add 6 to value stored at SPIN
```

**TABLE 3.2** Segment override prefix bytes

Segment	Prefix
CS	2EH
DS	3EH
ES	26H
FS	64H
GS	65H
SS	36H

### Register Indirect Addressing

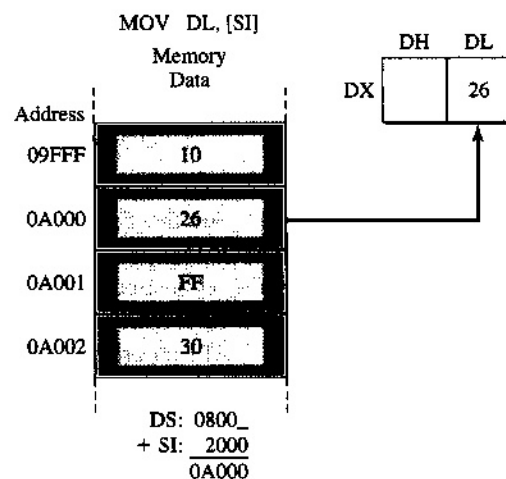
In this addressing mode we have a choice of four registers to use within the square brackets to specify a memory location. The four registers to choose from are the two base registers (BX and BP) and the two index registers (SI and DI). The 16-bit contents of the specified register are added to the DS register in the usual fashion (or SS if BP is used).

#### ■ EXAMPLE 3.4

What is the effective address generated by the instruction `MOV DL,[SI]`, if register SI contains 2000H, and the DS register contains 800H?

**Solution:** Shifting the DS register 4 bits to the left and adding the contents of register SI produces an effective address of 0A000H. Figure 3.6 illustrates this process.

**FIGURE 3.6** An example of register indirect addressing



The machine code for `MOV DL,[SI]` is 8A 14. ■

Using a register in the operand field to point to a memory location is a very common programming technique. It has the advantage of being able to generate any address within a specific segment simply by changing the value of the specified register.



Here are some additional examples of register indirect addressing:

```
MOV  ES:[DI],AL    ;put copy of AL into location pointed to
                   ;by DI in the extra segment
ADD  CX,[BP]       ;add word at location pointed to by BP (in
                   ;stack segment) to CX
XOR   [BX],DH      ;exclusive-or value at location pointed to
                   ;by BX with DH
JMP   [BP]         ;jump indirect to address stored at
                   ;location pointed to by BP
```

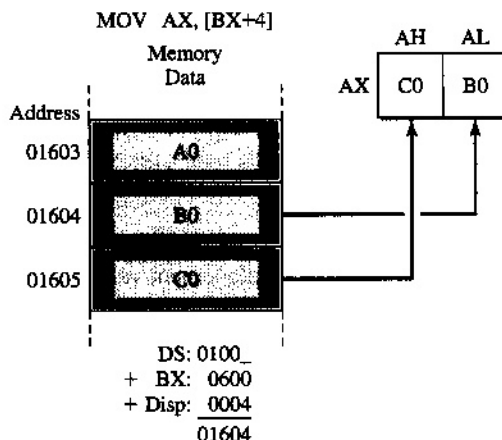
### Based Addressing

This addressing mode uses one of the two base registers (BX or BP) as the pointer to the desired memory location. It is very similar to register indirect addressing, the difference being that an 8- or 16-bit displacement may be included in the operand field. This displacement is also added to the appropriate segment register, along with the base register, to generate the effective address. The displacement is interpreted as a signed, 2's complement number. The 8-bit displacement gives a range of  $-128$  to  $+127$ . The 16-bit displacement gives a range of  $-32,768$  to  $32,767$ . So, the signed displacement allows us to point forward and backward in memory, a handy tool when accessing data tables stored in memory.

#### ■ EXAMPLE 3.5

What is the effective address generated by the instruction `MOV AX,[BX+4]`? Assume that the DS register contains 100H and register BX contains 600H.

**Solution:** Figure 3.7 shows how the DS register, the BX register, and the displacement value are added together to create the effective address 1604H. This address is then used to read the word out of locations 1604H and 1605H into the AX register.



**FIGURE 3.7** An example of based addressing

The machine code for `MOV AX,[BX+4]` is 8B 47 04. ■

When the BP register is used, the stack segment is used in place of the data segment. Here are some additional examples of based addressing:

```

SUB  [BP+10],BH      ;subtract BH from value stored at location
                      ;pointed to by BP plus 10 in the stack segment
MOV  DX,CS:[BX-2]     ;put a copy of the value stored at
                      ;location pointed to by BX minus 2 (in the
                      ;code segment) into DX
AND  [BP+4000H],CH    ;and CH with value stored at location
                      ;pointed to by BP plus 4000H
CMP  AL,[BX+100]      ;compare AL with value stored at
                      ;location pointed to by BX plus 100

```

### Indexed Addressing

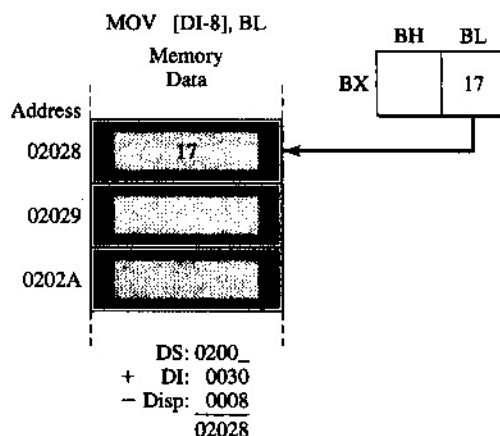
Like based addressing, indexed addressing allows the use of a signed displacement. The difference is that one of the index registers (SI or DI) must be used in the operand field.

#### ■ EXAMPLE 3.6

What is the effective address generated by the instruction `MOV [DI-8],BL`? Assume that the DS register contains 200H and that register DI contains 30H.

**Solution:** Figure 3.8 shows how the negative displacement is combined with the DI register. Notice that although the DI register points to address 30H within the data segment, the actual location accessed is 8 bytes behind.

**FIGURE 3.8** An example of indexed addressing



The machine code for `MOV [DI-8],BL` is 88 5D F8. As in the previous example, the third byte in the machine code is the displacement value. In this case, the displacement of -8 is coded as F8 hexadecimal, which is the 2's complement of 8. ■

Here are some additional examples of indexed addressing:

```

MOV  AL,ES:[SI]      ;put a copy of the value stored in location
                      ;pointed to by SI (in the extra segment) into AL
CALL [SI+2]          ;call subroutine whose address is
                      ;stored at location pointed to by SI plus 2
XOR  [DI-80H],DX     ;exclusive-OR value stored at location
                      ;pointed to by DI minus 80H with DX
ADD  [DI],BX         ;add BX to value stored at location
                      ;pointed to by DI

```

## Based Indexed Addressing

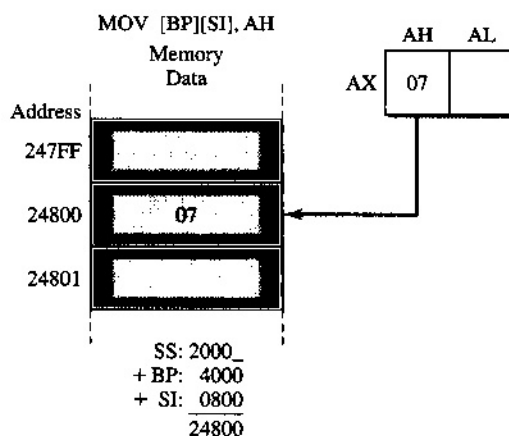
This addressing mode combines the features of based and indexed addressing but does not allow the use of a displacement. Two registers must be used as the source or destination operand, one from BX/BP and the other from SI/DI. The contents of both registers are *not* interpreted as signed numbers; therefore, each register may have a range of values from 0 to 65535. Once again, use SS instead of DS when BP is used.

### ■ EXAMPLE 3.7

What is the effective address generated by `MOV [BP+SI],AH`? Assume the SS register contains 2000H, register BP contains 4000H, and register SI contains 800H.

**Solution:** Shifting the SS register 4 bits to the left and adding the contents of BP and SI gives an effective address of 24800H. See Figure 3.9 for an illustration of this. Also note that a different form of the instruction is used in the figure. `MOV [BP][SI],AH` is a different way of saying `MOV [BP+SI],AH`. Both methods of specifying the operand are acceptable.

**FIGURE 3.9** An example of based indexed addressing



The machine code for `MOV [BP+SI],AH` is 88 22. ■

Here are some additional examples of based indexed addressing:

MOV	AX, (BX+SI)	;put copy of value stored in location pointed to by BX plus SI into AX
ADD	[BP+DI], BL	;add BL to value stored at location pointed to by BP plus DI in the stack segment
OR	CL, ES: [BX][DI]	;or CL with value stored at location pointed to by BX plus DI (in the extra segment)
PUSH	[BP][SI]	;push value stored at location pointed to by BP plus SI in the stack segment

Note that `[BP+SI]` and `[BP][SI]` are both acceptable operand formats.

## Based Indexed with Displacement Addressing

This addressing mode combines all of the features of the addressing modes we have been examining. As with all the others, it still accesses memory locations only within one 64KB segment. It does, however, give the programmer the option of using two registers to access

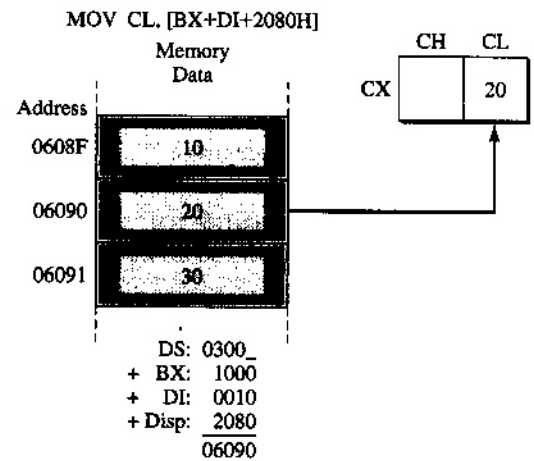
stored information, and the addition of the signed displacement makes this addressing mode the most flexible of all.

### ■ EXAMPLE 3.8

What memory location is accessed by `MOV CL,[BX+DI+2080H]`? Assume that the DS register contains 300H, register BX contains 1000H, and register DI contains 10H.

**Solution:** Figure 3.10 shows how the three registers and the displacement are added to create the memory address 06090H. The byte stored at this location is copied into register CL.

**FIGURE 3.10** An example of based indexed with displacement addressing



The machine code for `MOV CL,[BX+DI+2080H]` is 8A 89 80 20. In this case the last 2 bytes are the byte-swapped displacement. ■

Here are some additional examples of based indexed with displacement addressing:

ADD	AL,[BP+DI-50]	;add the value stored in location pointed to by BP plus DI minus 50 (in the stack segment) to AL
XOR	[BX+SI-4E00H],DX	;exclusive-OR DX with the value stored in location pointed to by BX plus SI minus 4E00H
JMP	[BP+SI+1000]	;jump to address stored in location pointed to by BP+SI+1000 in the stack segment
MOV	ES:[BX+DI+4],AX	;put copy of AX into location pointed to by BX plus DI plus 4 (in the extra segment)

## String Addressing

The instructions that handle strings do not use the standard addressing modes covered in the previous examples. So, when we look at a string instruction we will not see any registers listed in the operand field. For example, consider the string instruction `MOVSB` (move byte string). No processor registers are shown in the instruction, but the processor knows that it should use both SI and DI during execution of the instruction (as well as DS and ES). All of the string-based instructions assume that the SI register points to the first element in the source string (which might be either a byte value or a word value) and that the DI

register points to the first element of the destination string. The 80x86 will automatically adjust the contents of SI and DI during execution of the string instruction.

We will examine string operations in more detail in Section 3.7.

## Port Addressing

Intel brand microprocessors differ from other processors on the market in their implementation of the use of I/O ports for data communication between the processor and the outside world. One way to get information into the processor is to read it from memory. Another way is to read an input port. When sending data out of the CPU, we can direct it into a memory location or send it to an output port. The 80x86 provides the programmer with up to 65,536 input and output ports (although many useful designs rarely use more than a handful of I/O ports). An I/O port is accessed in much the same way that memory is accessed, by placing the address of the I/O port onto the address bus and enabling certain control signals. The address of the I/O port can be coded directly into an instruction, as in:

```
IN  AL, 40H
```

or

```
OUT 80H, AL
```

In this case, the port address must be between 00 and FFH, a total of 256 different I/O ports. Notice that the AL register is used to receive the port information. We could also use AX to receive 16 bits of data from an input port (as in `IN AX, 38H`).

A second method of addressing I/O ports requires that the port address be placed into the DX register. The corresponding instructions are:

```
IN  AL, DX
```

and

```
OUT DX, AL
```

Because we are now using register DX to store the port address, our choices range from 0000 to FFFFH, a total of 65,536 I/O locations. Note that `IN AX, DX` and `OUT DX, AX` are also allowed.

I/O ports are very useful for communication with peripherals connected to the processor, such as serial and parallel I/O devices, video display chips, clock/calendar chips, and many others.

## Using 32-Bit Addressing in Real Mode

Recall from Chapter 2 that the 80x86 provides an additional method of generating addresses that produces 32-bit effective addresses. In the real mode, these 32-bit addresses must fall within the familiar 64KB range (0000H to FFFFH) used within a segment. The advantage is that any extended register may be used as a base register or as an index register (or both in the same instruction). The only exception is the ESP register, which may only be used as a base register.

A typical instruction using 32-bit addressing might look like this:

```
MOV  AX, [EBX] [ECX*4]
```

In this example, EBX is the base register and ECX is the index register. Index registers may be *scaled* (multiplied) by a factor of 1, 2, 4, or 8. This allows easy access of 1-, 2-, 4-, and 8-byte quantities.

When a single register is used to specify the 32-bit address, it is automatically treated as a base register unless a scale factor is included. The following two instructions illustrate this important difference:

```
MOV SI, [EDX]      ;EDX is a base register
MOV DI, [EDX*8]    ;EDX is an index register
```

The same register may also be used as both a base and index register, as in:

```
MOV BX, [EAX] [EAX*2]
```

An optional displacement of 8 or 32 bits may also be included. This is useful for specifying the starting address of a block of data, such as an array.

When the base register is EBP or ESP, the stack segment is used for the memory access. Otherwise, the data segment is used by default.

Keep in mind that it is not possible to mix 16- and 32-bit registers in the same address operand. So, an instruction like:

```
MOV AL, [EBX] [CX]
```

causes an error during assembly, since EBX and CX may not be used together.

Take the time now to review the addressing modes we have just examined, because they are crucial to understanding the operation of the instructions we will begin looking at in Section 3.6.

---

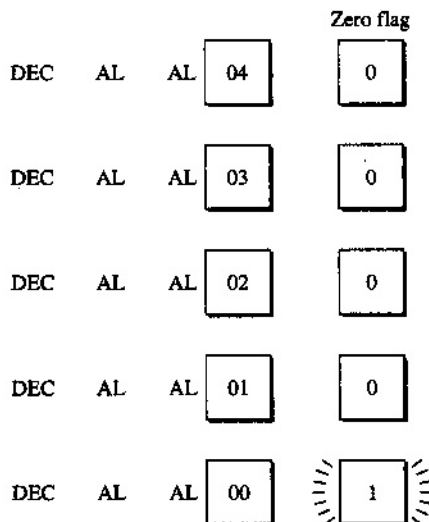
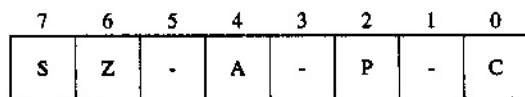
## 3.5 THE PROCESSOR FLAGS (CONDITION CODES)

The 80x86 has a number of status indicators that are referred to as **condition codes** or **flags**. Both terms convey useful information. The “condition” part of condition codes refers to information about the most recently executed instruction. Did the last DEC AL instruction produce a 0 in AL? The “zero condition” is an example of a condition code. From another point of view, when we see a flag waving, we often know that some new event has just occurred. In this sense, the **zero flag** is a way for the processor to wave at the programmer when a zero condition has occurred. Figure 3.11 shows how the zero flag changes from a 0 to a 1 when the AL register is finally decremented to zero. The flag is really a single bit in the flag register. This bit can be only a 0 or a 1. There are many instructions that look at the zero flag and make a decision based on its contents (e.g., jump if the zero flag is not 0).

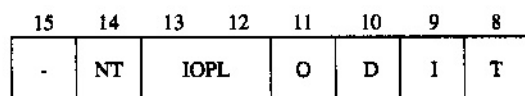
The processor has five main flags that are commonly tested during the execution of a program and others that we will examine later. The five main flags and their bit position within the processor’s flag register are shown in Figure 3.12(a). The flags and their meaning are as follows.

### Sign (Bit 7)

When we make use of signed binary data in a program, there are times when we wish to know if the last addition or subtraction produced a positive or negative result. Remember that when we use 2’s complement format, the most significant bit in the data is used as the **sign bit**. This would be bit 7 for a byte value and bit 15 for a word value. The processor examines this bit and adjusts its sign flag accordingly. When the sign flag is 0, the processor is saying that the number produced by the last arithmetic or logical instruction is positive. When the sign flag is a 1, the number can be interpreted as a negative number in 2’s complement notation.

**FIGURE 3.11** Operation of the zero flag**FIGURE 3.12** Flag register, lower word

(a) Condition code half



(b) Additional flags

**EXAMPLE 3.9**

Consider the following two pairs of instructions:

```
MOV AL, 3FH    MOV AL, 7FH
INC AL         INC AL
```

In each case, the final value in the accumulator (AL) will be interpreted as a *signed* binary number.

In the first pair of instructions, the accumulator value 3FH is incremented to 40H, which is 01000000 in binary. Notice that the MSB (bit 7) is 0, indicating that 40H is a *positive* number. The sign flag will be cleared in this case.

In the second pair of instructions, the accumulator value 7FH is incremented to 80H, which is 10000000 in binary. The MSB is now 1, indicating that 80H is a *negative* number. The sign flag will be set by this result. ■

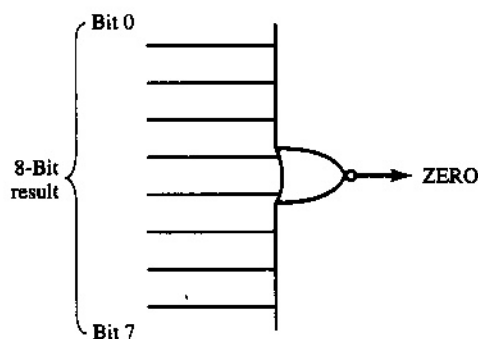
**Zero (Bit 6)**

We were briefly introduced to the zero flag in the beginning of this section. The zero flag is set (made equal to 1) or cleared (made equal to 0) after execution of many arithmetic and logical instructions. In Figure 3.11 we saw that the zero flag was set when the AL register

was finally decremented to 0. The zero flag is often used in programs to determine when a match has been found in a compare operation, when a register or memory location contains 0, and when a loop should be terminated. It is not difficult to see that the zero flag is set when a “zero” is created by an instruction.

■ **EXAMPLE 3.10** From the hardware perspective, it is not difficult to determine if a group of bits are all 0. As Figure 3.13 shows, a NOR gate is used to generate an output signal called ZERO. A NOR gate outputs a 1 when all of its inputs are 0. Think of the eight NOR inputs as if they were connected to the individual bits of AL. Thus, when AL becomes 00000000, ZERO will go

**FIGURE 3.13** Hardware generation of zero condition

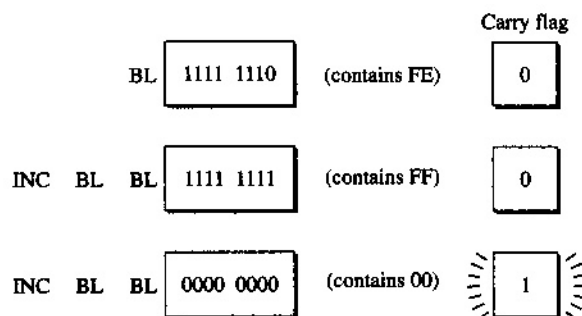


high. Any other 8-bit pattern in AL will cause ZERO to be low (indicating a *not-zero* condition). Can you imagine what is needed to check for 0 in a 16- or 32-bit register? ■

### Carry (Bit 0)

Suppose that register BL contains FEH. If we increment BL, we will get FFH. This represents an 8-bit number containing all 1s (or 255 as an unsigned decimal). If we increment BL again, what do we get? The correct answer is not 100H, but 00 *with a 1 in the carry flag*. Figure 3.14 shows this concept in graphical form. Because 100H requires 9 bits for representation, we cannot store 100H in register BL. We can store only the lower 8 bits, which are all low. The carry flag is used to store the ninth bit (or the 17th bit in a 16-bit operand and a 33rd bit in a 32-bit operand).

**FIGURE 3.14** Operation of the carry flag





■ **EXAMPLE 3.11** The carry flag is also used to indicate a *borrow* as a result of a subtraction. Consider these two pairs of instructions:

```
MOV AL, 6      MOV AL, 6
SUB AL, 1      SUB AL, 9
```

Remember that subtraction in binary is found by adding the 2's complement of one number to another.

In the first pair of instructions, subtracting 1 from 6 leaves 5. The carry flag is cleared in this case, because we subtracted a smaller number from a larger one.

In the second pair of instructions, subtracting 9 from 6 gives FDH (or 11111101 in binary, the 2's complement representation of  $-3$ ). Because we subtracted a larger number from a smaller one, the carry flag will be set, indicating a borrow. ■

### Auxiliary Carry (Bit 4)

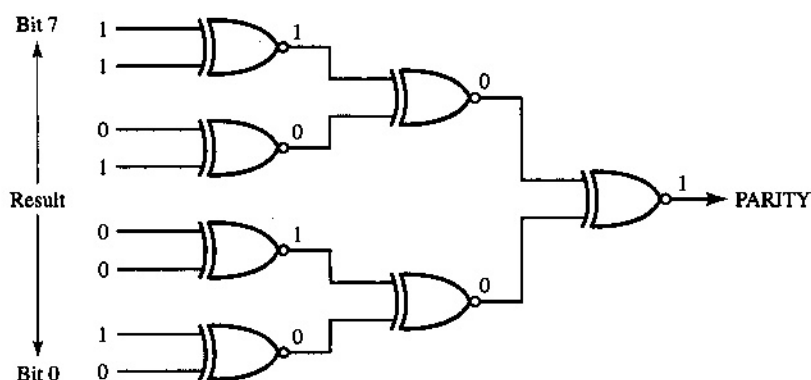
The operation of this flag is very similar to that of the carry flag except that the carry is out of bit 3 instead of bit 7 (or 15 or 31). In other words, the auxiliary carry flag indicates a carry out of the lower 4 bits. The main reason for including this flag is to aid in the execution of the processor's decimal adjust instructions that allow the programmer to work with BCD numbers. There are no instructions that directly test the state of this flag as there are for the sign, zero, and carry flags.

### Parity (Bit 2)

The parity bit is used to indicate the presence of an even number of 1s in the lower 8 bits of a result. For example, if a logical instruction produced the bit pattern 11010010 in register AL, the parity flag would be set, because there are an even number of 1s (four actually) in AL. If the pattern had been 11110010, the parity flag would be cleared, because five 1s is not even.

■ **EXAMPLE 3.12** The state of the parity flag is easily determined through the use of exclusive-NOR gates. As shown in Figure 3.15, seven exclusive NOR gates are used to determine the parity of an 8-bit result. An exclusive NOR gate outputs a 1 when its inputs are identical (00 or 11) and

**FIGURE 3.15** Parity generation with exclusive NOR gates



a 0 when its inputs are different. From Figure 3.15 it is clear that the input number 11010010 generates a 1 on the PARITY output, indicating even parity.

Of the five flags just examined, the first three (sign, zero, and carry) are the most often used. We must be familiar with the processor flags to understand the results produced by many of the instructions we will begin examining in the next section. ■

---

### Other Flags

A number of flags are found in bits 8–15 of the flag register (see Figure 3.12(b) for these). The trace, direction, interrupt-enable, overflow, I/O priority level, and nested task flags will all become important to us later, as will the protected mode flags in bits 16–31 of the flag register. For now, simply keep in mind that they are also part of the flag register.

The flags, or condition codes, contain valuable information concerning the operation of a program on an instruction-by-instruction basis. Thus, we should make good use of the flags when writing programs. Employ the conditional jump instructions where possible, and pay attention to how the flags are affected by all instructions in the program. Sometimes a well-written program that appears completely logical in its method will still yield incorrect results because a flag condition was overlooked.

Keep the condition codes in mind as you study the remaining sections and use Appendix B as you examine each new instruction.

---

## 3.6 DATA TRANSFER INSTRUCTIONS

As with any microprocessor, a detailed presentation of available instructions is important. Unless you have a firm grasp of what can be accomplished with the instructions you may use, your programming will not be efficient. Indeed, you may even create problems for yourself.

Still, there is no better teacher than experience. You should experiment with these instructions and examine the results. Compare what you see with the manufacturer's data. A difficult concept often becomes clear in practice.

Each of the following sections deals with a separate group of instructions. Information about the instruction, how it works, how it is used, what its mnemonic looks like, how it affects the condition codes, and more, will be presented for each instruction. Even so, it is strongly suggested that you constantly refer to Appendix B as you read about each new instruction. Most of the material in this appendix, such as allowable addressing modes and condition code effects, is not reproduced here.

The instructional groups are organized in such a way that the more commonly used instructions are presented first, followed by less frequently used instructions. For example, in the group dealing with data movement instructions, the MOV instruction is presented first due to its wide range of applications. Although the LDS and LES instructions come first alphabetically, they have restricted functions and are not needed in many programming applications. So, they do not get to steal the spotlight from MOV by appearing first.

Hopefully, covering the instructions in this fashion will allow you to study the important instructions first (in each group), and the other instructions as the need arises.

In most cases the machine code for the instruction will be included. This is done simply to compare instruction lengths and explain new features about the 80x86.

Examples will also be given for each instruction. Let us begin our coverage of the instruction set with the data transfer instructions.

## Moving Data

This group of instructions makes it possible to move (copy) data around inside the processor and between the processor and its memory and I/O systems. Remember that the more popular instructions are covered first.

**MOV Destination,Source (Move Data).** We have already seen many examples of the MOV instruction in its role of explaining the addressing modes. So, instead of repeating those examples here, we will examine other aspects of the MOV instruction. When we use MOV to load immediate data into a register, the assembler will look at the size of the specified register in the operand field to determine if the immediate data is a 1-, 2-, or 4-byte number. For example:

```
MOV AL,30H
```

and

```
MOV AX,30H
```

are two different instructions and the immediate data 30H is interpreted differently in each. In the first instruction, the 30H is coded as a byte value because it is being MOVED into AL. In the second instruction, the 30H is coded as a word value because it is being MOVED into AX. This is clearly shown by the resulting code for both instructions. MOV AL,30H is coded as B0 30. MOV AX,30H is coded as B8 30 00. Note that the second two bytes represent the byte-swapped value 0030H.

This much we have already seen. But what happens when the assembler does not have any way of determining how large the operands should be? For example, in MOV [SI],0 the processor does not know if the 0 should be coded as a byte value, word value, or as a double-word value. This instruction would then produce an error during assembly. For cases like this, include some additional information in the instruction's operand field. If you wish to MOV a byte value into memory, use MOV BYTE PTR [SI],0. Word values require MOV WORD PTR [SI],0 and double-word values require MOV DWORD PTR [SI],0. The byte ptr, word ptr, and dword ptr assembler directives stand for "byte pointer," "word pointer," and "double-word pointer." The corresponding code for MOV BYTE PTR [SI],0 is C6 04 00. For MOV WORD PTR [SI],0 it is C7 04 00. For MOV DWORD PTR [SI],0, we get the machine code 66 C7 04 00000000. Notice the operand-size prefix byte. This pointer feature of the assembler can be applied to many 80x86 instructions.

**MOVSX Destination,Source (Move with Sign Extended).** When working with signed binary values, it is common to convert 8- or 16-bit numbers into 16- or 32-bit *sign-extended* numbers. Recall that the MSB of a signed number is used to represent the sign (+/-) of the number. It is commonly called the sign bit. Negative numbers have a sign bit equal to one. The MOVSX instruction examines the state of the sign bit when extending the source value. When the source is an 8-bit operand, the upper byte of the destination will be set to 00H for a positive source value or FFH for a negative source value. Thus, the sign of the source is extended through the upper 8 bits of the destination.

When the source is a 16-bit operand, the sign is extended through the upper 16 bits of the destination, resulting in 0000H or FFFFH in the upper two bytes. Example 3.13 demonstrates the operation of MOVSX.

- **EXAMPLE 3.13** Suppose that register AL contains 36H and register BX contains C3EEH. What are the results of `MOVSX AX,AL` and `MOVSX EBX,BX`?

**Solution:** When `MOVSX AX,AL` executes, the result in AX equals 0036H. The sign bit of AL is zero, causing the upper 8 bits of AX to be set to zero.

`MOVSX EBX,BX` causes the sign bit of BX (a one) to be extended through the upper 16 bits of EBX, giving FFFFC3EEH as the 32-bit result. Each destination register has the same sign as its corresponding source register.

The machine code for `MOVSX AX,AL` is 0F BE C0. The machine code for `MOVSX EBX,BX` is 66 0F BF DB. ■

---

***MOVZX Destination,Source (Move with Zero Extended).*** This instruction is used to convert 8- and 16-bit values into 16- and 32-bit values by adding leading zeros to the source operand. Thus, a value like 89H becomes 0089H, and 3700H becomes 00003700H. This is similar to what `MOVSX` does, except that the sign bit is ignored. `MOVZX` should not be used when working with signed numbers.

- 
- **EXAMPLE 3.14** Repeat Example 3.13 for the same register values, except use `MOVZX` instead of `MOVSX`. What are the final results?

**Solution:** Register AX contains 0036H, and EBX contains 0000C3EEH. Note the difference in EBX from the FFFFC3EEH value of Example 3.13. The upper 16 bits of EBX are not sign-extended by `MOVZX` as they are by `MOVSX`.

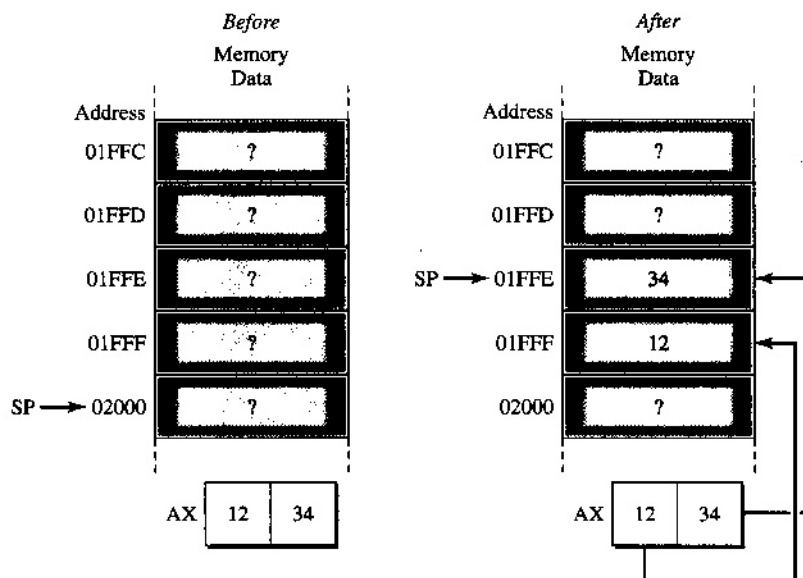
The machine code for `MOVZX AX,AL` is 0F B6 C0. The machine code for `MOVZX EBX,BX` is 66 0F B7 DB. ■

---

***PUSH Source (Push Data onto Stack).*** It is often necessary to save the contents of a register so that it can be used for other purposes. The saved data may be copied into another register or written into memory. If we are interested only in saving the contents of one or two registers, we could simply reserve a few memory locations and then directly `MOV` the contents of each register into them. This practice is limiting, however, because we cannot easily modify our program in the future (say, to save additional registers in memory) without making a significant number of changes. Instead, we will use a special area of memory called the **stack**. The stack is a collection of memory locations pointed to by the stack pointer register and the stack segment register. When we wish to write data into the stack area, we use the `PUSH` instruction. We commonly refer to this as pushing data onto the stack. The processor will automatically adjust the stack pointer during the push in such a way that the next item pushed will not interfere with what is already there. Example 3.15 shows the operation of `PUSH` in more detail.

- 
- **EXAMPLE 3.15** The stack segment register has been loaded with 0000 and the stack pointer register with 2000H. If register AX contains 1234H, what is the result of `PUSH AX`?

**Solution:** The stack pointer (presently containing 2000H) points to a location referred to as the top of the stack. Whenever we push an item onto the stack, the stack pointer is decremented by 2. This is necessary because all pushes involve 2 bytes of data (usually the contents of a register). Decrementing the stack pointer during a push is a standard way of



**FIGURE 3.16** Execution of `PUSH AX`

implementing stacks in hardware. Figure 3.16 shows the new contents of memory after `PUSH AX` has executed. The data contained in memory locations 01FFF and 01FFE is replaced by the contents of register AX. Notice that the contents have been byte-swapped during the write (34 comes before 12) and that the new stack pointer value is 1FFE. Remember that the stack *builds* toward 0. As more items are pushed, the address within the stack pointer gets smaller by 2 each time. Also notice that the contents of register AX remain unchanged.

The machine code for `PUSH AX` is 50. ■

When the SP register is pushed, the value written to the stack is the value of SP before the push.

***PUSHW/PUSHD Source (Push Word/Double-Word Data onto Stack).*** The operation of `PUSHW` and `PUSHD` is similar to `PUSH`. `PUSHW` is used to push immediate word values onto the stack, as in:

```
PUSHW 1000H
```

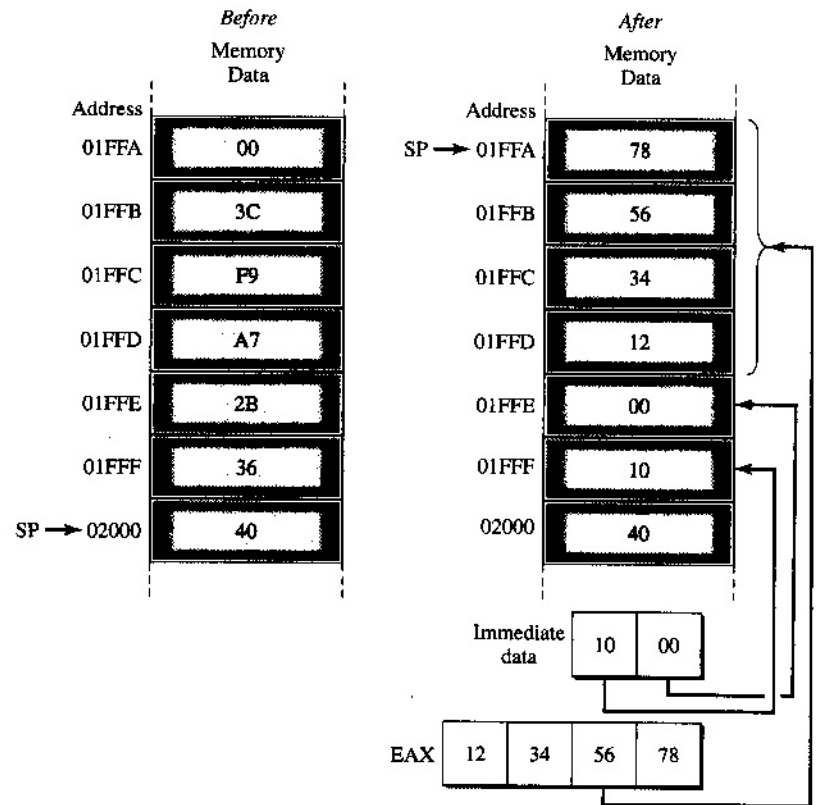
`PUSHD` is used to push double-word values onto the stack. `PUSHD` may be used to push the 32-bit processor registers onto the stack, as in:

```
PUSHD EAX
```

`PUSHW` decrements the stack pointer by 2 and `PUSHD` decrements SP by 4. Note that `PUSH EAX` and `PUSHD EAX` are equivalent.

■ **EXAMPLE 3.16** Figure 3.17 illustrates the changes made to stack memory and the SP during execution of these two instructions:

```
PUSHW 1000H
PUSHD EAX
```

**FIGURE 3.17** Execution of PUSHW 1000H and PUSHD EAX

Register EAX contains 12345678H. The stack pointer is decremented by 6 during execution of both instructions.

The machine code for `PUSHW 1000H` is 68 00 10. The machine code for `PUSHD EAX` is 66 50. ■

**PUSHA/PUSHAD (Push All Registers/Push All Double-Registers).** These instructions automatically push all general purpose registers and the stack pointer onto the stack. `PUSHA` pushes all 16-bit registers and `PUSHAD` pushes all 32-bit registers. Table 3.3 indicates the order in which the registers are pushed.

**TABLE 3.3** Order of registers pushed with `PUSHA/PUSHAD`

When Pushed	PUSHA	PUSHAD
First	AX	EAX
	CX	ECX
	DX	EDX
	BX	EBX
	SP	ESP
	BP	EBP
	SI	ESI
Last	DI	EDI

■ **EXAMPLE 3.17** If the stack pointer initially contains 2000H, what is its value after PUSHAD executes?

**Solution:** Since eight 32-bit registers are pushed, the SP is decremented by 4 a total of eight times. This gives a final SP value of 1FE0H.

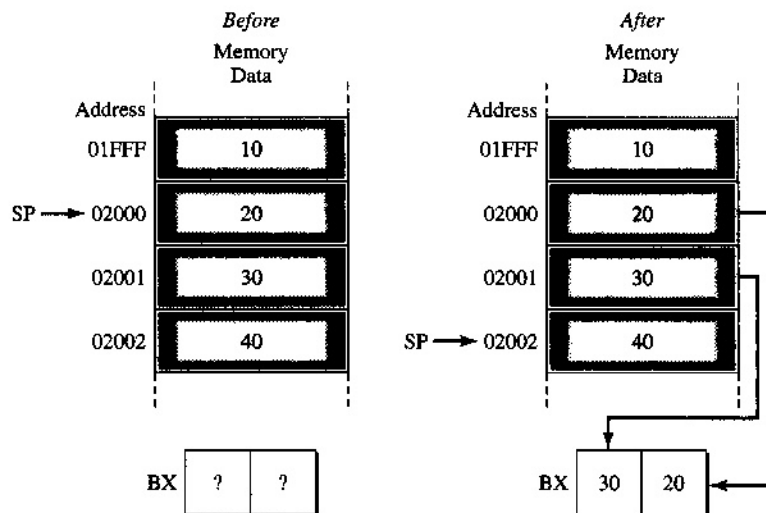
The machine code for PUSHAD is 66 60. ■

**POP Destination (Pop Word Off Stack).** The POP instruction is used to perform the reverse of a PUSH. The stack pointer is used to read 2 bytes of data and copy them into the location specified in the operand field. The stack pointer is automatically incremented by 2. All registers except CS and IP may be popped. The destination operand may be a memory location.

■ **EXAMPLE 3.18** Assume the contents of the stack segment register and the stack pointer are 0000 and 2000H, respectively. What is the result of POP BX?

**Solution:** Figure 3.18 shows a snapshot of memory contents in the stack area. The contents of location 2000 (20) are copied into the lower byte of BX. The stack pointer is incremented and the contents of location 2001 (30) are copied into the upper half of BX. The old contents of BX are lost. The stack pointer is then incremented a second time. Compare the operation of PUSH and POP and you will see that they complement one another.

**FIGURE 3.18** Execution of POP BX



The machine code for POP BX is 5B. ■

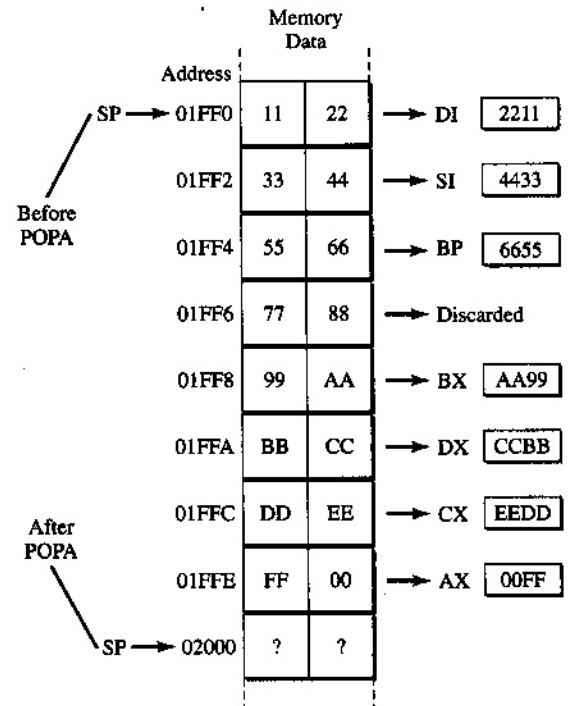
**POPA/POPAD Destination (Pop All Registers/Pop All Double-Registers).** These instructions complement the PUSHA/PUSHAD instructions. All general purpose registers (16-bit for POPA, 32-bit for POPAD) are popped from the stack in the order indicated in Table 3.4. It is important to note that the contents of the SP (or ESP) are not loaded with the data popped off the stack. This is necessary to prevent the stack from changing locations halfway through the execution.

**TABLE 3.4** Order of registers popped with POPA/POPAD

When Popped	POPA	POPAD
First	DI	EDI
	SI	ESI
	BP	EBP
	SP	ESP
	BX	EBX
	DX	EDX
	CX	ECX
Last	AX	EAX

Note: The value popped for SP/ESP is discarded.

- **EXAMPLE 3.19** Figure 3.19 shows the result of executing the POPA instruction. Initially, the SP contains 1FF0H. After POPA executes, the SP contains 2000H, *not* the value of 8877H popped off the stack. The memory data is shown in byte-swapped format, which accounts for the difference of 2 between each address.

**FIGURE 3.19** Loading all general purpose registers with POPA

The machine code for POPA is 61. ■

**IN Accumulator,Port (Input Byte or Word from Port).** This is another instruction that was briefly introduced during our examination of addressing modes. The processor has an I/O space that is separate from its memory space. There are 65,536 possible I/O ports available



for use by the programmer. In almost all cases a machine designed around the 80x86 would use only a handful of these ports.

The input port is actually a hardware device connected to the processor's data bus. When executing the IN instruction, the processor will output the address of the input port on the address bus. The selected input port will then place its data onto the data bus to be read by the processor. Data read from an input port *always ends up in the accumulator*.

The processor allows two different forms of the IN instruction. If a full 16-bit port address must be specified, the port address is loaded into register DX, and IN AL,DX or IN AX,DX is used to read the input port. If the port number is between 00 and FFH, a different form of IN may be used. In this case, to input from port 80H we would use IN AL,80H or IN AX,80H. Using AL in the operand field causes 8 bits of data to be read. Two bytes can be input by using AX in the operand field.

■ **EXAMPLE 3.20** What is the result of IN AL,80H if the data at input port 80 is 22?

**Solution:** The byte value 22 is copied into register AL.

The machine code for IN AL,80H is E4 80. ■

It may be helpful for you to remember the expression "All I/O is through the accumulator" when working with the I/O instructions. Input ports can be used to read data from keyboards, A/D converters, DIP switches, clock chips, UARTs, and other peripherals that may be connected to the CPU.

**INS Destination,DX (Input String from Port).** This instruction is very similar to IN, except that the destination is a memory location instead of the accumulator. The memory location is pointed to by the DI register and is located in the extra segment. It is common to use the notation ES:DI to represent a segment: offset pair. So, ES:DI is the memory location where the input port (address in DX) data will be written to. The destination operand specified in the instruction is only used to indicate the size of the data transfer: byte, word, or double-word.

After the input port data has been written into memory, the DI register is automatically incremented or decremented by 1, 2, or 4, depending on the state of the processor's direction flag. This is a feature of string operations, which we will examine more closely in Section 3.7.

Three simplified instructions are equivalent to INS, one for each data size. INSB inputs bytes, INSW inputs words, and INSD inputs double-words. All three use DX as the port address and ES:DI as the destination, but do not require any operands, as INS does.

■ **EXAMPLE 3.21** The following instructions (and associated machine code) perform the same job:

```
6C      INSB or INS  BYTE PTR ES:[DI],DX
6D      INSW or INS  WORD PTR ES:[DI],DX
66 6D   INSD or INS  DWORD PTR ES:[DI],DX ■
```

**OUT Port,Accumulator (Output Byte or Word to Port).** This instruction is a complement to the IN instruction. With OUT, we can send 8 or 16 bits of data to an output port. The port address may be loaded into DX for use with OUT DX,AL or OUT DX,AX, or specified within the instruction, as in OUT 80H,AL or OUT 80H,AX.

- **EXAMPLE 3.22** What happens during execution of `OUT DX,AL` if AL contains 7C and DX contains 3000?

**Solution:** The port address stored in register DX is output on the address bus, along with the 7C from AL on the data bus. The output port circuitry must recognize address 3000 and store the data.

The machine code for `OUT DX,AL` is EE. ■

**OUTS DX,Source (Output String to Port).** This instruction reads data from string memory located at DS:SI and outputs it to the port specified by register DX. The size of the source operand controls the size of the data output to the port. The SI register is automatically incremented or decremented (depending on the state of the direction flag) after the data transfer is made.

Three simplified forms of OUTS are recognized by the assembler. They are OUTSB, OUTSW, and OUTSD. Each outputs a byte, word, or double-word, respectively, to the port specified by register DX. No operands are necessary, since the data will automatically be read using the SI register and the data segment.

- **EXAMPLE 3.23** Using a source operand to specify the size of the data transfer may be done like this:

```
6E          OUTS    DX,BYTE PTR [SI]
6F          OUTS    DX,WORD PTR [SI]
66 6F      OUTS    DX,DWORD PTR [SI]
```

The same result may be accomplished by using one of the simplified forms:

```
6E          OUTSB
6F          OUTSW
66 6F      OUTSD
```

Remember that the SI register is automatically incremented or decremented by 1, 2, or 4 after each operation. ■

String operations, including another look at OUTS, will be covered in detail in Section 3.7.

**LEA Destination,Source (Load Effective Address).** This instruction is used to load the offset of the source memory operand into one of the processor's registers. The memory operand may be specified by any number of addressing modes. The destination may not be a segment register.

- **EXAMPLE 3.24** What is the difference between `MOV AX,[40H]` and `LEA AX,[40H]`?

**Solution:** In `MOV AX,[40H]` the processor is directed to read 2 bytes of data from locations 40H and 41H and place the data into register AX. In `LEA AX,[40H]` the processor simply places 40H into register AX. Modifying these instructions slightly should help to further define the difference just presented. Suppose that a label called TIME has been defined in a program and has the memory address value of 40H associated with it. Using `MOV AX,TIME` will cause the data at locations TIME and TIME+1 (40H and 41H) to be read from memory and stored in AX. Using `LEA AX,TIME` will cause the effective address of the label (which is 40H) to be copied into AX. The memory location TIME is

never accessed in `LEA AX,TIME`. All operands dealing with memory locations are assumed to be found in a 64KB block pointed to by the data segment register.

What does `LEA AX,[SI]` do? It *does not* read the contents of the memory locations pointed to by SI. Instead, the value of SI at execution time is loaded into AX.

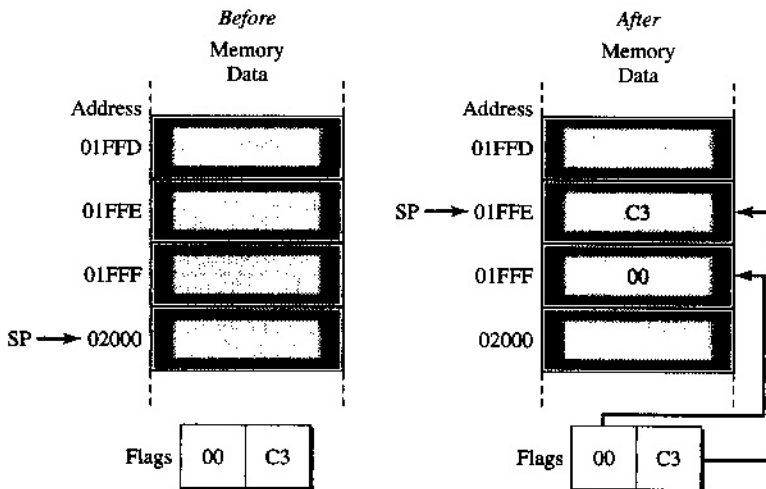
The machine code for `LEA AX,[40H]` is 8D 06 40 00. The machine code for `LEA AX,[SI]` is 8D 04. ■

**PUSHF/PUSHFD (Push Flags onto Stack).** There are times when it is necessary to save the state of each flag in the processor's flag register. Usually this is done whenever the processor is interrupted. Saving the flags and restoring them at a later time, along with the processor registers, is a proven technique for resuming program execution after an interrupt. `PUSHF` pushes the lower 16 bits of the flag register onto the stack. Use `PUSHFD` to push the entire 32-bit flag register.

### ■ EXAMPLE 3.25

Assume that the stack segment register and the stack pointer have been loaded with addresses 0000 and 2000H, respectively, and that the flag register contains 00C3H. What is the result of `PUSHF`?

**Solution:** `PUSHF` writes the contents of the flag register into stack memory. The operation of `PUSHF` is similar to `PUSH`, as you can see in Figure 3.20. The stack pointer is decremented by 2. Then the lower byte of the flag register (C3) is written into stack memory, followed by the upper byte (00). As usual, we see that the 16-bit flag register has been byte-swapped as it was written into memory.



**FIGURE 3.20** Operation of `PUSHF`

The machine code for `PUSHF` is 9C. The machine code for `PUSHFD` is 66 9C. ■

**POPF/POPFD (Pop Flags off Stack).** This instruction reverses the operation of `PUSHF`/`PUSHFD`, popping 2 or 4 bytes off the stack and storing them in the flag register. The operation is similar to `POP`, with the stack pointer increased by 2 or 4 at completion.

The machine code for POPF is 9D. The machine code for POPFD is 66 9D.

**XCHG Destination,Source (Exchange Data).** This instruction is used to swap the contents of two 8-, 16-, or 32-bit operands. One operand must be a processor register (excluding the segment registers). The other operand may be a register or a memory location. If a memory location is used as an operand it is assumed to be within a data segment.

■ **EXAMPLE 3.26** Registers AL and BL contain 30 and 40, respectively. What is the result of XCHG AL,BL?

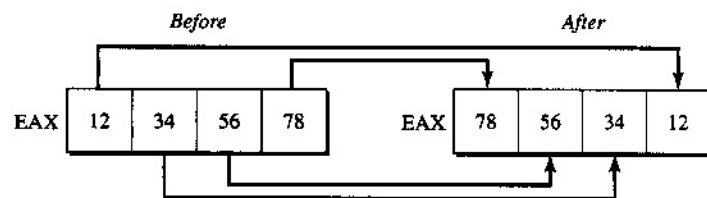
**Solution:** After execution, AL contains 40 and BL contains 30.

The machine code for XCHG AL,BL is 86 C3. It may be interesting to note that the machine code for XCHG BL,AL (which performs the same operation as XCHG AL,BL) is 86 D8. ■

**BSWAP Destination (Byte-Swap).** This instruction swaps bytes in a 32-bit general purpose register. The upper and lower bytes switch places, as do the middle 2 bytes. BSWAP is useful for converting 32-bit numbers from *little-endian* format into *big-endian* format, and vice versa. A number stored in little-endian format has its lower byte in the lowest memory location. Big-endian format places the upper byte in the lowest memory location. Big-endian numbers are found in Motorola processors, such as the 680x0 series. Intel machines use little-endian numbers (Intel byte-swapping).

■ **EXAMPLE 3.27** The contents of register EAX are swapped with BSWAP as indicated in Figure 3.21. Note that executing two BSWAP instructions in a row with the same register restores the register to its original value.

**FIGURE 3.21** Result of BSWAP EAX



The machine code for BSWAP EAX is 66 0F C8. ■

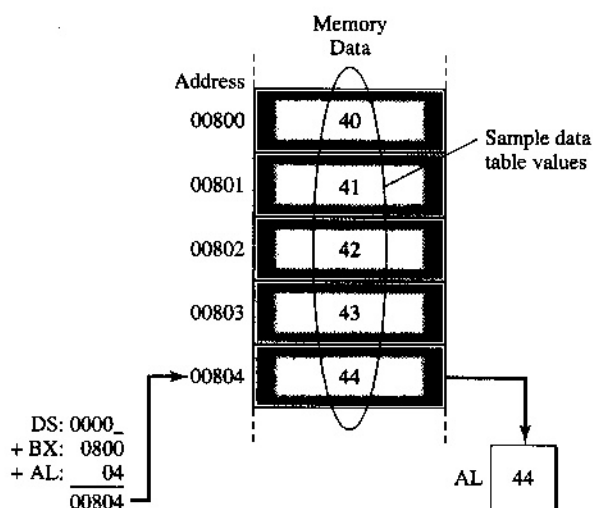
**XLAT Translate-Table (Translate Byte).** Some programming applications require quick translation from one binary value to another. For example, in an image processing system, binary video information defining brightness in a gray-level image can be falsely colored by translating each binary pixel value into a corresponding color combination. The process is easily accomplished with the aid of a color look-up table. XLAT is one instruction that is useful for implementing such a look-up table. XLAT assumes that a 256-byte data table has been written into memory at the starting address contained in register BX. The number in register AL at the beginning of execution is used as an index into the translation table. The

byte stored at the address formed by the addition of BX and AL is then copied into AL. The translation table is assumed to be in the data segment.

■ **EXAMPLE 3.28** A translation table resides in memory with a starting address of 0800H. How does XLAT know where the table is? If register AL contains 04, what is the result of XLAT?

**Solution:** XLAT uses register BX as the pointer to the beginning of the translation table, so it is necessary to place the address 0800H into BX before executing XLAT (assume here that the DS register contains 0000). This can be easily done with `MOV BX,0800H`. Figure 3.22 shows the result of executing XLAT with AL equal to 04. The byte at address 0804H is copied into register AL, giving it a final value of 44.

**FIGURE 3.22** Execution of XLAT



The machine code for XLAT is D7. ■

A very useful application of XLAT is encryption. Suppose that the 26 letters of the alphabet are scrambled and then written into memory at a starting address found in register BX. If register AL is restricted to numbers from 1 to 26 (1 representing A, 26 representing Z), executing XLAT will map one letter of the alphabet into a different letter. In this fashion, we can encrypt text messages one character at a time. A second table would then be needed to translate the encrypted message back into correct form.

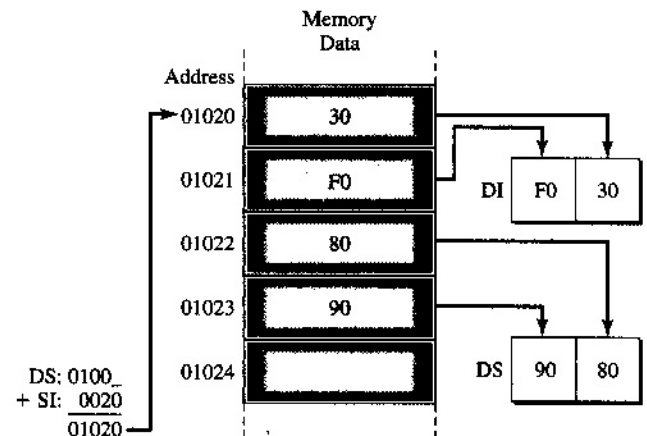
Another application involves PC keyboards. Because each key on a keyboard generates a unique 8-bit code, it is possible to create different keyboard "layouts," where the layout is actually a translation table that contains user-defined keyboard codes. The original keyboard codes are used as addresses within the translation table to generate the correct code.

**LDS Destination,Source (Load Pointer Using DS).** This instruction is used to load two 16-bit registers from a 4-byte block of memory. The first 2 bytes are copied into the register specified in the destination operand of the instruction. The second 2 bytes are copied into the DS register. This instruction will come in handy when working with source strings, which we will look at in the string instructions section.

■ **EXAMPLE 3.29** Assume that the DS register contains 0100 and register SI contains 0020. What is the result of `LDS DL,[SI]`?

**Solution:** Figure 3.23 shows how the SI register is used to access the data that will be copied into DI and DS. Note that the addresses read out of memory are byte-swapped. After execution, the DS register contains 9080 and register DI contains F030.

**FIGURE 3.23** Execution of `LDS DI,[SI]`



The machine code for `LDS DI,[SI]` is C5 3C. ■

**LES/LFS/LGS/LSS Destination,Source (Load Pointer Using ES/FS/GS/SS).** This instruction is nearly identical to `LDS`. The difference is that the second address read out of memory is written into the indicated segment register instead of the DS register.

■ **EXAMPLE 3.30** All of the load segment/register instructions may be used with 32-bit extended registers as well. Here are a few examples with accompanying machine code:

```
67 66 C5 06      LDS    EAX, FWORD PTR [ESI]
66 C4 1E 1000    LES    EBX, FWORD PTR TEMP
67 66 0F B4 0C 97 LFS    ECX, FWORD PTR [EDI] [EDX*4]
66 0F B2 26 2000 LSS    ESP, FWORD PTR NEWSTACK
```

The labels `TEMP` and `NEWSTACK` are located at addresses 1000H and 2000H, respectively. The `FWORD PTR` directive is used to indicate the 32-bit size of the offset in the address operand.

The last instruction is particularly useful because it is able to completely change the working stack address during its execution. ■

**LAHF (Load AH Register from Flags).** One way to determine the state of the flags is to load a copy of them into a register. Then individual bits within the register can be manipulated or tested by the programmer. `LAHF` can be used to copy the lower byte of the flag register into register AH.

■ **EXAMPLE 3.31** The lower byte of the flag register contains 83H. What is the result of LAHF and what is the state of each flag?

**Solution:** LAHF copies 83 into register AH. Refer to Figure 3.12 for information on the flag positions. Because the binary equivalent of 83H is 10000011, we see that the sign and carry flags are currently set, and the zero, auxiliary carry, and parity flags are cleared.

The machine code for LAHF is 9F. ■

**SAHF (Store AH Register into Flags).** This instruction is used to load a new set of flags into the flag register. The contents of register AH are copied into the lower byte of the flag register, giving new values to all five main processor flags.

The machine code for SAHF is 9E.

### Assembler Directives OFFSET, BYTE PTR, WORD PTR, DWORD PTR, FWORD PTR, and SEG

There are times when we need to provide a small amount of assistance to the assembler so that it can figure out how to code an instruction. For example, the instruction:

```
MOV AL, [SI]
```

contains an operand reference to 8-bit register AL. This indicates to the assembler that the memory location pointed to by [SI] is a *byte* location. By a similar method, the instruction:

```
MOV [SI], AX
```

contains a reference to the 16-bit register AX. What happens when an instruction contains no reference to size, as in:

```
MOV [SI], 5
```

In this case, the assembler will give an error saying that the operand must have the size specified. The assembler does not know if we are trying to write the byte 05, the word 0005, or even the double-word value 00000005 into memory.

To get around instances such as this, we make use of the **BYTE PTR** (byte pointer), **WORD PTR** (word pointer), and **DWORD PTR** (double-word pointer) assembler directives. In terms of our example, to move the byte value 05 into memory we would use:

```
MOV BYTE PTR [SI], 5
```

and to write the word 0005 we would use:

```
MOV WORD PTR [SI], 5
```

Double-word values are specified like this:

```
MOV DWORD PTR [SI], 5
```

Situations like these arise only when we have not used the **DB**, **DW**, or **DD** directives to define the size of a data area.

It is often necessary to load the address of a label into a register. In particular, DOS **INT 21H**, Function 09H, requires that the address of an ASCII text string be loaded into

register DX prior to its call. There are two ways to accomplish this. In the first method, we use the LEA instruction:

```
B4 09          MOV  AH,9
8D 16 000E     LEA  DX,MESSAGE
CD 21          INT  21H
```

In the second method, we use the assembler directive OFFSET to perform the same chore:

```
B4 09          MOV  AH,9
BA 000E       MOV  DX,OFFSET MESSAGE
CD 21          INT  21H
```

Can you spot the advantage of using the OFFSET directive? Notice that the LEA DX instruction requires 4 bytes of machine code. The equivalent MOV DX instruction needs only 3 bytes of code and fewer clock cycles as well, which results in faster execution. This might not seem like a big savings, but consider that a source file might be hundreds or even thousands of lines long. An instruction like LEA DX,MESSAGE might appear quite often in the source file. If it appears 100 times, then 100 bytes are saved using MOV DX with OFFSET. Writing efficient programs, programs that both run quickly and require little space in memory, is important. Using programming techniques like the ones just shown can help shorten the length of an executable program.

The previous instructions that set up register DX for INT 21H's function 09H assumed that the DS register had already been initialized to its proper value. But what if this has not been done? Then it is the programmer's responsibility to initialize the data segment to the address assigned by DOS when the program was loaded into memory. For example, how can we determine the segment address for the MESSAGE string? This is accomplished with the SEG directive. Two statements are needed to initialize a segment register with SEG, such as:

```
MOV  AX, SEG MESSAGE
MOV  DS, AX
```

The SEG directive determines the segment where the MESSAGE data is located and places the segment address into AX. This address is then loaded into the DS register with the second MOV instruction. Normally, when only one data area is used in a program, it is not necessary to use the SEG operator, because the single data area is automatically assigned to the data segment. When multiple data areas are required, it may be necessary to initialize the ES, FS, and GS registers. The SEG directive will come in handy in this case.

Last, if 32-bit addressing modes are used, it may be necessary to declare a 48-bit pointer variable. Pointer variables contain a 16-bit segment portion and a 16- or 32-bit offset portion. So, pointers are either 32 bits or 48 bits in length. A 48-bit pointer can be declared like this (in the .DATA section of the source file):

```
BIGPTR    DF    ?
```

where the DF (define far word) directive automatically reserves 6 bytes of storage for the pointer. Any instruction that uses BIGPTR as an operand, such as:

```
LDS      EDX, BIGPTR
```

will access the 6 bytes correctly.

When it is not possible to use a far label (such as BIGPTR), the FWORD PTR (far word pointer) directive can be used to specify the proper operand size. Example 3.30 shows examples of this directive.

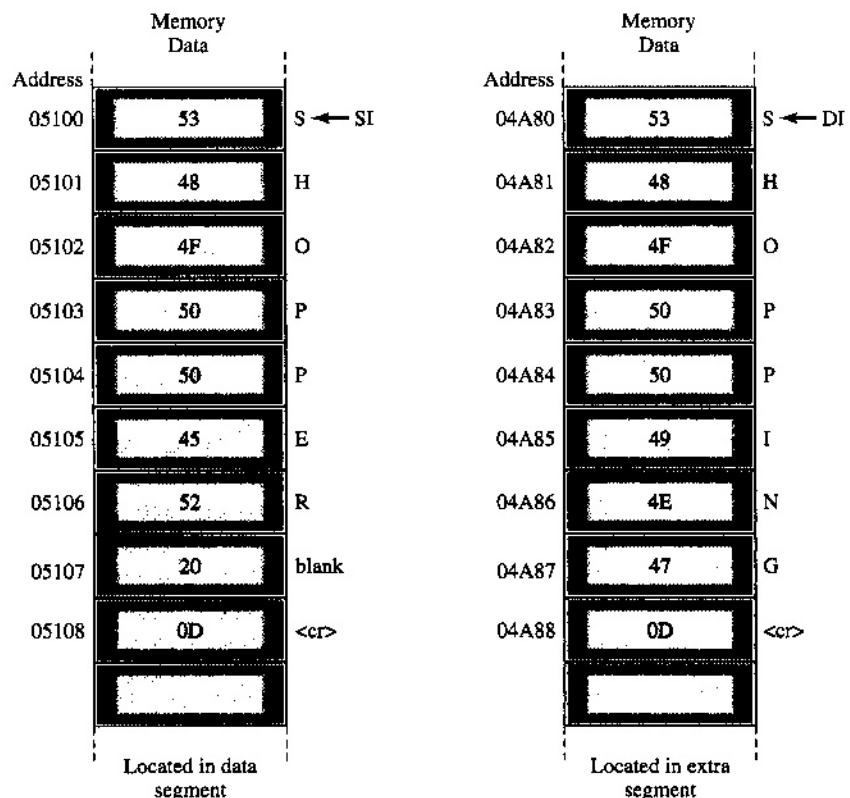


## 3.7 STRING INSTRUCTIONS

A particularly nice feature of the 80x86 is its ability to handle strings. A string is a collection of bytes, words, or long-words that can be up to 64KB in length. An example of a string might be a sequence of ASCII character codes that constitute a password, or the ASCII codes for "YES." A string could also be the 7 bytes containing your local telephone number. No matter what kind of information is stored in a string, there are a number of common operations we find useful to perform on them. During the course of executing a program, it may become necessary to make a copy of a string, compare one string with another to see if they are identical, or scan a string to see if it contains a particular byte or word. The processor has instructions designed to do this automatically. A special instruction called the **repeat prefix** can be used to repeat the copy, compare, or scan operations. Register CX has an assigned role in this process. It contains the repeat count necessary for the repeat prefix. CX is decremented during the string operation, which terminates when CX reaches 0. The SI and DI registers are also vital parts of all string operations. The processor assumes that the SI register points to the first element of the source string, which must be located in the data segment. The destination string is located in a similar way via the DI register and must reside in the extra segment. A special flag called the **direction flag** is used to control the way SI and DI are adjusted during a string instruction. They are automatically incremented or decremented by 1, 2, or 4, based on the value of the direction flag and the size of the string elements.

Figure 3.24 gives an example of two text strings stored in memory. The first string spells out SHOPPER, and is followed by a blank (20H) and a carriage return code (0DH).

**FIGURE 3.24** Two sample text strings



The second string spells out SHOPPING and is followed by only a carriage return code. Although the strings are the same length (9 bytes) they are different after the fifth character. We will be able to use these strings in our examples to understand the operation of the string instructions.

### Initializing the String Pointers

Before we can use any string instruction, we have to set up the SI, DS, DI, and ES registers. There are a number of ways this can be done. The source string (SHOPPER) in Figure 3.24 could be pointed to by these instructions:

```
MOV  AX,510H    ;string segment-address
MOV  DS,AX
MOV  SI,0        ;string offset within segment
```

When the contents of SI and DS are combined to form an effective address, 05100H will be the first byte accessed in the data segment. A similar technique is used to initialize the destination string (SHOPPING):

```
MOV  AX,4A8H    ;string segment-address
MOV  ES,AX
MOV  DI,0        ;string offset within segment
```

Remember that we cannot move immediate data directly into a segment register, hence our use of the accumulator in the first instruction.

Another way to initialize the string pointers is through the LDS and LES instructions. In this way, both strings can be initialized with only two instructions:

```
LDS  SI, SRCSTR
LES  DI, DSTSTR
```

where SRCSTR and DSTSTR are the labels of two 4-byte pointer fields that contain the string offset and segment values. Refer to Example 3.29 for a review of LDS. (LES works in a similar fashion.)

### Using String Instructions

**REP/REPE/REPZ/REPNE/REPNZ.** These five mnemonics are available for use by the programmer to control the way a string operation is repeated, if at all. REP (*repeat*), REPE (*repeat while equal*), REPZ (*repeat while zero*), REPNE (*repeat while not equal*), and REPNZ (*repeat while not zero*) are all recognized by the assembler as prefix instructions for string operations. MOVS (*move string*) and STOS (*store string*) make use of the REP prefix. When preceded by REP, these string operations repeat until CX decrements to 0. REPE and REPZ operate the same way, but are used for SCAS (*scan string*) and CMPS (*compare string*). Here, an additional condition is needed to continue the string operation. Each time SCAS or CMPS completes its operation, the zero flag is tested and execution continues (repeats) as long as the zero flag is set. This makes sense, because a compare operation involves an internal subtraction, and the subtraction produces a 0 result when both operands match. The zero flag is set in the case of matching operands and cleared for different ones.

REPNE and REPNZ also repeat as long as CX does not equal 0 but require that the zero flag be cleared to continue. So, we have three ways to repeat string operations:

1. Repeat while CX does not equal 0.
2. Repeat while CX does not equal 0 and the zero flag is set.
3. Repeat while CX does not equal 0 and the zero flag is cleared.

If necessary, combinations of string operations can be used to perform a specific kind of string function.

**MOVS *Destination-String,Source-String (Move String)*.** This instruction is used to make a copy of the source string in the destination string. The names of the strings must be used in the operand fields so that the processor knows whether they are byte strings or word strings. A byte string might be defined like this:

```
STRINGX DB 'SHOPPER',0DH
```

and a word string like this:

```
STRINGY DW 1000H,2000H,3000H,4000H
```

The assembler will associate the DB or DW directive used in the source line to set the type of string being defined. Thus, the assembler will know what kind of strings it is working with when it encounters the operands in MOVS. The operands in MOVS are only used to set the string size. The DS:SI and ES:DI registers must already be initialized to the starting address of each string.

When MOVS executes with the REP prefix, CX must be initialized to the proper count. The state of the direction flag will determine which way the strings are copied. If it is cleared, SI and DI will auto-increment. If the direction flag is set, SI and DI will auto-decrement. The direction flag can be cleared with the CLD instruction and set with the STD instruction.

**MOVSB/MOVSW/MOVSQ (*Move String*).** These three mnemonics can be used in place of MOVS and cause identical execution. Because they explicitly inform the assembler of the string size, there is no need to include the string operands in the instruction.

### ■ EXAMPLE 3.32

What instructions are necessary to make a copy of the SHOPPER string from Figure 3.24? We want the destination string to have a starting address of 3000H, and the index registers should auto-increment during the string operation.

**Solution:** Because the SHOPPER string is 9 bytes long, we must initialize CX to 9. The direction flag must be cleared to get the copy performed in the correct manner, and REP must be used to copy bytes from the source string until CX is decremented to 0. One way to do all this would be:

```
B8 10 05    MOV     AX,510H      ;source string segment-address
8E D8       MOV     DS,AX
29 F6       SUB     SI,SI        ;source string offset
B8 00 03    MOV     AX,300H      ;destination string segment-address
8E C0       MOV     ES,AX
29 FF       SUB     DI,DI        ;destination string offset
FC         CLD                ;auto-increment
F3         REP                     ;repeat while CX <> 0
A4         MOVSB                ;copy string
```

Note the alternate method used to place 0000H in SI and DI. The SUB instructions require 2 bytes of code each. MOV SI,0 and MOV DI,0 would require 3 bytes each. The code is included for each instruction for your interest. ■

**CMPS** *Destination-String,Source-String (Compare String)*. This instruction is used to compare two strings. The compare operation, as we have already seen, is accomplished by an internal subtraction of the destination and source operands. So, in this case, a byte or word from the destination string is subtracted from the corresponding element of the source string. If the two elements are equal, the zero flag will be set. Different elements cause the zero flag to be cleared. The REPZ prefix will allow strings to be checked to see if they are identical. This is a very handy tool when writing interactive programs (programs that require a response from the user). For example, if a user enters "ZOOM IN" when asked for a command, the program can check this string against all legal command strings to see if it matches any of them. String comparisons are also employed in spell checkers, programs that automatically find misspelled words in a text file. (The text for this book was run through a spell checker in a relatively short period of time.)

Because the flags are adjusted during execution of CMPS, we know if the two strings matched by examining the zero flag at the end of execution. JZ MATCH (jump to MATCH if the zero flag is set) can be used to detect matching strings.

### ■ EXAMPLE 3.33

Assume that DS:SI and ES:DI have been initialized to the starting addresses of the two strings from Figure 3.24. If REPZ CMPS STRINGA,STRINGB is executed with CX equal to 4, do the strings match? Do they match if CX equals 8? What state must the direction flag be in?

**Solution:** The direction flag must be cleared so that SI and DI auto-increment during the compare. When CX equals 4, the processor compares only the first 4 bytes of each string. Because each string begins with "SHOP," the zero flag remains set throughout the compare and we get a match. When CX equals 8, CMPS will repeat until SI and DI point to the sixth byte in each string. Then the comparison fails due to the "E" in "SHOPPER" and the "I" in "SHOPPING." The zero flag is then cleared and the instruction terminates, even though CX has not yet decremented to 0. This indicates that the strings are different.

The machine code for CMPS is A6 for byte strings, A7 for word strings, and 66 A7 for double-word strings. ■

The assembler allows the use of CMPSB, CMPSW, and CMPSD as alternate forms of the instruction. Once again, no operands are needed with these instructions.

**SCAS** *Destination-String (Scan String)*. This instruction is used to scan a string by comparing each string element with the value saved in AL, AX, or EAX. AL is used for byte strings, AX for word strings, and EAX for double-word strings. The string element pointed to by ES:DI is internally subtracted from the accumulator and the flags adjusted accordingly. Once again the zero flag is set if the string element matches the accumulator, and cleared otherwise. The accumulator and the string element remain unchanged. If REPZ is used as the prefix to SCAS, the string is effectively searched for the item contained in the accumulator. Alternately, if REPZ is used, a string can be scanned until an element *differs* from the accumulator. This is especially handy when working with text strings. Suppose a text string contains a number of leading blanks (ASCII code 20H) before the actual text begins. SCAS can be used with 20H in AL and REPZ as the prefix to skip over the leading blanks. When the first nonblank (non-20H byte) character is encountered, the scan will terminate with DI pointing to the nonblank character.

- **EXAMPLE 3.34** Suppose that the ES and DI registers are initialized to point to the starting address of the “SHOPPING” string in Figure 3.24. What is the result of `REPZ SCAS` if CX contains 9 and AL contains 4EH?

**Solution:** With CX set to 9, the processor will be able to scan each element of the “SHOPPING” string. However, the `REPZ` prefix will allow the scan to continue only as long as the current string element does *not* match the byte stored in AL. There is no match between the accumulator until ES:DI points to address 04A86H. At that point, the accumulator matches the contents of memory and the scan terminates with the zero flag set.

The machine code for `SCAS` is AE for a byte string, AF for a word string, and 66 AF for a double-word string. ■

The assembler recognizes `SCASB`, `SCASW`, and `SCASD` as alternate forms of `SCAS`.

**LDS Source-String (Load String).** This instruction is used to load the current string element into the accumulator and automatically advance the SI register to point to the next element. It is assumed that DS:SI have already been set up prior to execution of `LDS`.

The direction flag determines whether SI is incremented or decremented.

- **EXAMPLE 3.35** Refer to Figure 3.24 once again. Assume that DS:SI currently point to address 05105H and that the direction flag is set. What is the result of executing `LDS` with a byte-size operand?

**Solution:** `LDS` copies the byte from the current address indicated by DS:SI (which is 45H) into AL and then *decrements* SI by 1 to point to the next element. SI is decremented because the direction flag is set. The next string element is at address 05104H.

The machine code for `LDS` is AC for byte strings, AD for word strings, and 66 AD for double-word strings. ■

The assembler accepts `LDSB`, `LDSW`, and `LDS D` as explicit uses of `LDS`.

**STOS Destination-String (Store String).** This instruction is used to write elements of a string into memory. The contents of AL, AX, or EAX are written into memory at the address pointed to by ES:DI and then DI is adjusted accordingly depending on the state of the direction flag and the size of the string elements.

- **EXAMPLE 3.36** We wish to modify the “SHOPPING” string of Figure 3.24 by adding the word “MALL” to it. The carriage return code 0DH will be replaced by a blank, and the character codes for “MALL” and another carriage return will be added. How should `STOS` be used to accomplish this modification?

**Solution:** Because the modification represents 6 bytes to write into memory, we will use the word version of `STOS` to write two codes into memory at a time. First, we will write the blank code and the letter “M.” Then the codes for “A” and “L,” and finally the code for “L” and the carriage return. The direction flag will be cleared to allow auto-incrementing of DI,

and ES:DI must be initialized to address 04A88H to begin. The necessary instructions are as follows:

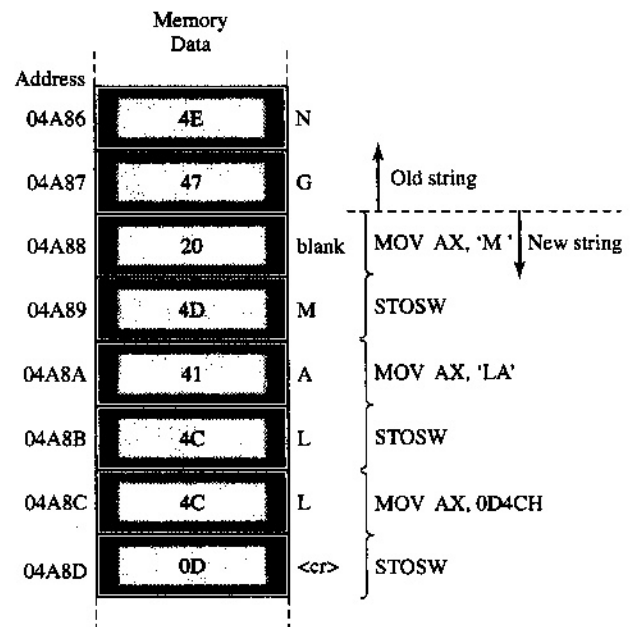
```

B8 A8 04    MOV    AX,4A8H      ;destination string segment-address
8E C0       MOV    ES,AX
BF 08 00    MOV    DI,8        ;destination offset
FC         CLD                ;auto-increment
B8 20 4D    MOV    AX,'M '     ;code for BLANK and 'M'
AB         STOSW
B8 41 4C    MOV    AX,'LA'     ;code for 'A' and 'L'
AB         STOSW
B8 4C 0D    MOV    AX,0D4CH    ;code for 'L' and CR
AB         STOSW

```

The first three instructions are needed to initialize ES:DI to address 04A88H. Because the 04A8H number in ES will become 4A80H when the processor forms the effective address, DI must be initialized to 0008 to get the right starting address for the modification. Because the cpu will byte-swap the string words as they are written into memory, it is necessary to place them into AX in reverse order. For example, in `MOV AX,'M '` the ASCII codes for M and blank become the word 4D20H (when the assembler looks up the ASCII codes). The lower byte of 4D20H is 20H, the first character we wish to write into string memory. The upper byte gets written into memory next, which places the code for "M" *after* the code for blank. Figure 3.25 indicates this operation. Three STOSWs are needed to write the six new string characters.

**FIGURE 3.25** Modifying a string with STOS



The machine code for STOS is AA for byte strings, AB for word strings, and 66 AB for double-word strings. ■

As with the other string instructions, the assembler recognizes STOSB, STOSW, and STOSD as explicit forms of STOS.

## Another Look at INS/OUTS

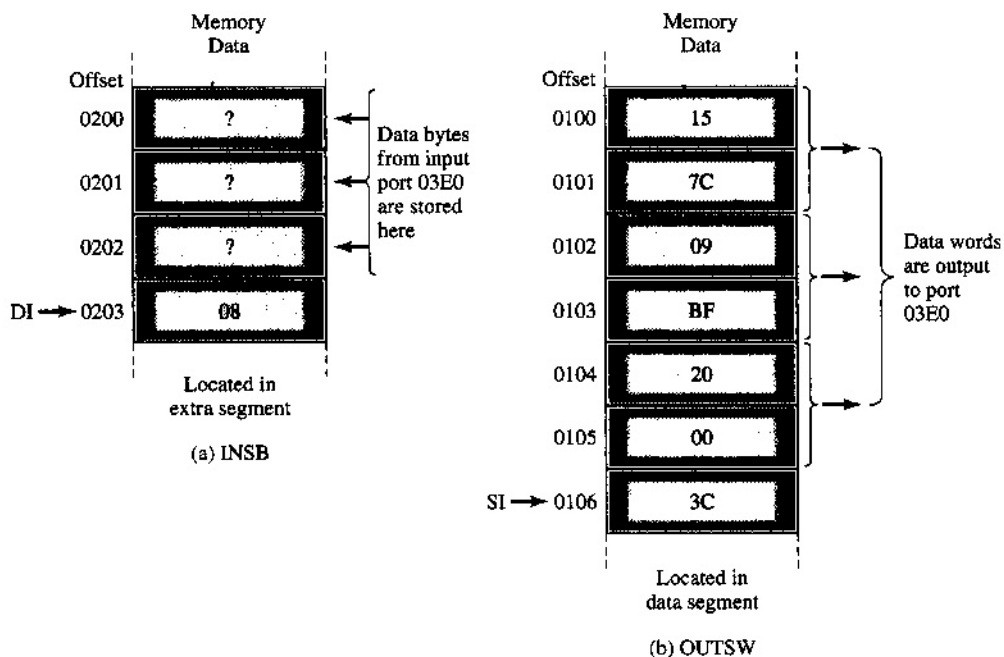
Recall that there are two string instructions, INS and OUTS, that transfer data between an I/O port and string memory. The source string DS:SI supplies data to the OUTS instruction, and INS writes its data to the destination string ES:DI. As usual with string operations, the index register is incremented or decremented by 1, 2, or 4 after each operation.

■ **EXAMPLE 3.37** The following registers contain the indicated hexadecimal values:

DS = 0400      ES = 0600      CX = 0003  
SI = 0100      DI = 0200      DX = 03E0

Furthermore, the direction flag is clear. What are the results of executing these instructions?

REP INSB      REP OUTSW



**FIGURE 3.26** Operation of INSB and OUTSW

**Solution:** The INSB instruction is repeated three times. This causes the processor to read 3 bytes from input port 03E0 and store them in string locations 0200, 0201, and 0202, as indicated by Figure 3.26(a). The final value in DI is 0203.

The OUTSW instruction also executes three times. Each word read from string memory is sent to output port 03E0. The words are located at addresses 0100, 0102, and 0104. The final value in SI is 0106. Figure 3.26(b) illustrates OUTSW's execution. ■



---

## 3.8 TROUBLESHOOTING TECHNIQUES

With a large portion of the instruction set still left to cover, it would be good to stop here and review some key points to remember when working with 80x86 instructions.

- Examine the relationship between the instruction mnemonic and the resulting machine code. You will begin to see patterns. These patterns will help you discover errors such as leaving the H off a hexadecimal number like 16H in the instruction `MOV BL,16`. The assembler will convert the 16 into 10H and generate the machine code B3 10 instead of B3 16. Knowing what to expect is a good place to start.
- Know your data sizes. It is frustrating to look at the instruction `MOV AL,500` and wonder why it does not assemble. It looks okay, does it not? But remember that AL is the lower 8-bit half of AX and can only handle integers as large as 255. Knowing your data sizes will help you avoid problems like this.
- Do not make the assembler guess. For example, the instruction `MOV [SI],5` looks okay, but it actually generates an error because the assembler does not know if the "5" is an 8-bit 5, a 16-bit 5, or a 32-bit 5. This affects how many memory locations are accessed by [SI] and could lead to trouble if the assembler does not use the correct size. To force a particular size, such as 16 bits, use `MOV WORD PTR [SI],5`.
- Learn how to use several of the most basic addressing modes, which segment registers are involved, and how they form the physical address.
- Once again, always be aware of how an instruction uses, or affects, the processor flags. Many programs do not work simply because the flags are ignored by the programmer. This will be especially true in Chapter 4 when we examine the arithmetic and logical instructions.
- Before using any string instructions, make sure the direction flag is set to the appropriate value, so that the index registers update properly.

These points, and others we will see in Chapter 4, should go a long way toward eliminating many of the common errors encountered when working with assembly language.

---

## SUMMARY

In this chapter we began coverage of the 80x86 instruction set. This included detailed looks at each addressing mode and flag available to the programmer in the real mode. Data transfer and string instructions were explained by example, including operations allowed on the 32-bit extended registers. The important concept of a stack was introduced, to illustrate how to save data in memory.

The structure of source and list files was also covered, with attention paid to the assembler details necessary for writing proper code. All of the information presented here will be useful as you complete coverage of the instruction set in Chapter 4.

---

## STUDY QUESTIONS

1. Explain the use of the ORG, DB, DW, and END assembler directives.
2. What happens when a source file is assembled?
3. What two files are created by the assembler?



4. What are the opcode, data type, and operand(s) in this instruction:

```
MOV AH, 7
```

5. What is meant by byte-swapping?
6. Which of these assembler directives produce data: ORG, DB, DW, DD, DF, SEG, END?
7. What does a linker do?
8. List the seven basic instruction groups.
9. Identify the source and destination addressing mode in each of these instructions:
- (a) MOV AX, BX
  - (b) MOV AH, 7
  - (c) MOV [DI], AL
  - (d) MOV AX, [BP]
  - (e) MOV AL, [SI+6]
  - (f) JNZ XYZ
  - (g) CBW
10. Why are the flags so important in a control-type program?
11. What does this two-instruction sequence do?
- ```
XCHG AX, BX
XCHG BX, CX
```
12. What does this instruction accomplish?
- ```
BSWAP EBX
```
13. Memory locations 00490H through 00493H contain, respectively, 0A, 9C, B2, and 78. What does AX contain after each instruction? (Assume that SI contains 00490H and that BP contains 0002.)
- (a) MOV AX, [SI]
  - (b) MOV AX, [SI+1]
  - (c) MOV AX, [SI][BP]
14. Registers AX, BX, CX, and DX contain, respectively, 1111H, 2222H, 3333H, and 4444H. What are the contents of each register after this sequence of instructions?
- ```
PUSH AX
PUSH CX
PUSH BX
POP DX
POP AX
POP BX
```
15. What is the difference between LDS and LES? When should each be used?
16. Show the instructions needed to scan a 200-byte string for the byte 25H.
17. Repeat Question 16 for the word value 9A25H.
18. Redo Example 3.32 for auto-decrement copying.
19. Redo Example 3.36 by using byte operation instead.
20. Explain how the processor switches from 16-bit operands to 32-bit operands, when operating in the real mode.
21. What are the simplified segment directives discussed in this chapter?
22. If EAX contains 00000200H, EBX contains 00000003H, and the data segment contains 1000H, what is the effective address generated by these instructions?
- (a) MOV ECX, [EAX]
  - (b) MOV ECX, [EBX][EAX]
  - (c) MOV ECX, [EAX][EBX\*8]
  - (d) MOV ECX, [ESI][EDI]

23. Which registers in Question 22 are base registers? Which are index registers?
24. Can the stack pointer be used as an index register when using 32-bit addressing?
25. What scale values may be used in 32-bit addressing?
26. What is a segment override prefix? Show an example of when it may be used.
27. If an I/O port address is greater than FFH, what must be done to use IN or OUT (or INS/OUTS)?
28. What is the difference between IN and INS?
29. If BX contains 3000H, what are the results of these two instructions?
  - (a) `MOVSB EBX, BX`
  - (b) `MOVZSB EBX, BX`
30. Repeat Question 29 for BX equal to 9A00H.
31. How can all general purpose registers be pushed or popped from the stack with a single instruction?
32. How is register AL used during execution of XLAT?
33. The segment address of the word variable ABC is not known. Show how the SEG directive can be used to initialize the extra segment so that ABC is accessible via ES. Also show the instruction needed to read ABC into register DX.
34. What are the physical addresses actually used during the string operations shown in Figure 3.26?
35. What are the word values output to the port in Figure 3.26?

---

## CHAPTER 4

---

# 80x86 Instructions, Part 2: Arithmetic, Logical, Bit Manipulation, Program Transfer, and Processor Control Instructions

---

### OBJECTIVES

In this chapter you will learn about:

- The arithmetic, logical, and bit manipulation instructions
- The program transfer (loop/jump and subroutine/interrupt) instructions
- The processor control instructions
- How an assembler generates machine code
- The special properties of relocatable code

### KEY TERMS

|                           |                        |                 |
|---------------------------|------------------------|-----------------|
| Binary coded decimal      | Interrupt-type         | Relative offset |
| Direct jump               | Interrupt vector table | Relocatability  |
| Frame pointer             | Intersegment transfer  | Stack frame     |
| Indirect jump             | Jump                   | Subroutine      |
| Intrasegment transfer     | Overflow flag          | Trace flag      |
| Interrupt-enable flag     | Packed decimal number  |                 |
| Interrupt service routine | Procedure              |                 |

---

## 4.1 INTRODUCTION

Chapter 3 introduced the flags, addressing modes, data transfer, and string instructions. This chapter completes coverage of the 80x86 instruction set. As before, each instruction is presented with an accompanying example and its associated machine code.

In addition, we will examine the properties of the relocatable machine code used by the processor and see how it is generated by an assembler.

Section 4.2 covers the arithmetic instructions. These are followed by the logical and bit manipulation instructions in Sections 4.3 and 4.4, respectively. Program transfer instructions (loops, jumps, subroutines) are explained in Section 4.5. The details of the processor control instructions are examined in Section 4.6. Section 4.7 shows how an assembler generates 80x86 machine code. The relocation properties of the processor's machine code are discussed in Section 4.8. A final set of troubleshooting techniques for the 80x86 instruction set is provided in Section 4.9.

---

## 4.2 ARITHMETIC INSTRUCTIONS

This group of instructions provides the 80x86 with its basic integer math skills. Floating-point math operations are handled by a separate group of instructions, which we will examine in Chapter 11. Addition, subtraction, multiplication, and division can all be performed on different sizes and types of numbers. You will need to refer to Appendix B often to understand fully the effects of each instruction on the processor's flags.

### The Instructions

**ADD** *Destination,Source (Add Bytes, Words, or Double-Words)*. This instruction is used to add 8-, 16-, or 32-bit operands together. The sum of the two operands replaces the destination operand. All flags are affected.

---

#### ■ EXAMPLE 4.1

If AX and BX contain 1234H and 2345H, respectively, what is the result of `ADD AX,BX`? How would this answer compare with `ADD BX,AX`?

**Solution:** Considering `ADD AX,BX`, adding 1234H to 2345H gives 3579H, which replaces the contents of AX. BX still contains 2345H. `ADD BX,AX` generates the same sum, but the contents of BX are changed instead.

The machine code for `ADD AX,BX` is 01 D8. For `ADD BX,AX` we get 01 C3. ■

We can also add constants (immediate data) to registers or memory. For example, `ADD CX,7` would add 7 to the number stored in register CX. The machine code for this instruction is 83 C1 07. The 80x86 will sign-extend the immediate value before it is used.

**ADC** *Destination,Source (Add Bytes, Words, or Double-Words with Carry)*. The operation of ADC is similar to ADD; however, in addition to the source operand the processor also adds the contents of the carry flag. The carry flag is then updated based on the size of the result. Other flags are also affected. ADC is commonly used to add multibyte operands together (such as 128-bit numbers).

---

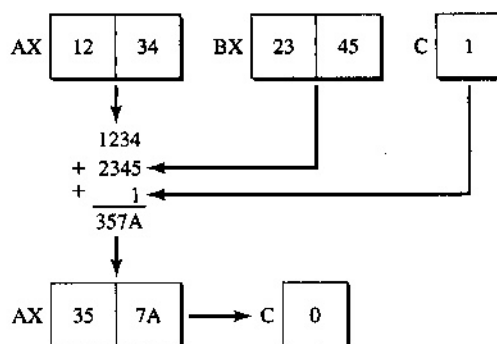
#### ■ EXAMPLE 4.2

Using the same register values from Example 4.1, what is the result of `ADC AX,BX` if the carry flag is set?

**Solution:** Figure 4.1 shows how AX, BX, and the carry flag are added together to get 357AH, which replaces the contents of register AX. The carry flag is then cleared, because 357AH fits into a 16-bit register.

The machine code for `ADC AX,BX` is 11 D8. ■

---

**FIGURE 4.1** Execution of ADC AX,BX

**INC Destination (Increment Byte, Word, or Double-Word by 1).** There are many times when we need to add only 1 to a register or to the contents of a memory location. We could do this using the ADD instruction, but it is simpler to use INC. Also, the use of INC generates less machine code than the corresponding ADD instruction, resulting in faster execution. All flags except the carry flag are affected. Use ADD destination, 1 to update the carry flag.

**EXAMPLE 4.3**

The two instructions `ADD AX,1` and `INC AX` both increase the value of register AX by 1. The machine code for each differs greatly, with `ADD AX,1` assembling into 3 bytes (05 01 00) and `INC AX` only requiring 1 byte (40). Because 3 bytes take longer to fetch from memory than 1 byte, `INC AX` executes faster than `ADD AX,1`. ■

**SUB Destination,Source (Subtract Bytes, Words, or Double-Words).** SUB can be used to subtract 8-, 16-, or 32-bit operands. If the source operand is larger than the destination operand, the resulting borrow is indicated by setting the carry flag.

**EXAMPLE 4.4**

If AL contains 00 and BL contains 01, what is the result of `SUB AL,BL`?

**Solution:** Subtracting 01 from 00 in 8-bit binary results in FFH, which is the 2's complement representation of -1. So, the contents of AL are replaced with FFH, and both the carry and sign flags are set to indicate a borrow and a negative result. ■

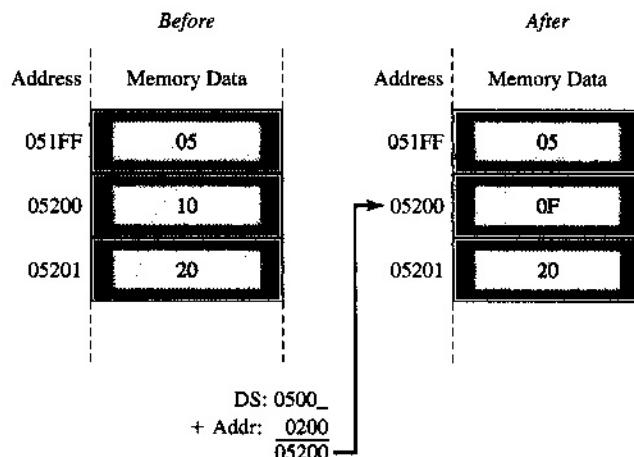
**SBB Destination,Source (Subtract Bytes, Words, or Double-Words with Borrow).** SBB executes in much the same way as SUB, except the contents of the carry flag are also subtracted from the destination operand. The contents of the carry flag are updated at completion of the instruction.

**DEC Destination (Decrement Byte, Word, or Double-Word by 1).** DEC provides a quick way to subtract 1 from any register or the contents of any memory location. All flags except the carry flag are affected.

**EXAMPLE 4.5**

What is the result of `DEC byte ptr [200H]`? Assume that the DS register contains 0500H.

**Solution:** Because we are decrementing the contents of a memory location, the assembler must be informed of the operand size. This is accomplished with the `byte ptr` directive. Figure 4.2 shows the change made to location 05200H when the instruction executes.

**FIGURE 4.2** Execution of  
DEC BYTE PTR [200H]

The machine code for DEC byte ptr [200H] is FE 0E 00 02. Notice that the last 2 bytes are the byte-swapped offset 0200H. ■

**CMP Destination,Source (Compare Bytes, Words, or Double-Words).** This very useful instruction allows the programmer to perform comparisons on byte, word, and double-word values. Comparisons are employed in search algorithms and whenever range checking needs to be done on input data. For example, it may be beneficial to use CMP to check a register for a 0 value prior to multiplication or division. The internal operation of CMP is actually a subtraction of the destination and source operands without any modification to either. The flags are updated based on the results of the subtraction. The zero flag is set after a CMP if the destination and source operands are equal, and cleared otherwise. All other flags are also affected. Immediate data is sign-extended to the size of the destination before the comparison is performed.

#### ■ EXAMPLE 4.6

What CMP instruction is needed to determine if the accumulator (AX) contains the value 3B2EH?

**Solution:** We should use CMP AX,3B2EH to do the checking. As we will see later, we should follow the CMP instruction with some kind of conditional jump (as in JZ MATCH or JNZ NOMATCH to control the flow of a program). We could also test each half of AX individually, using CMP AH,3BH and CMP AL,2EH, but this results in more code and slower execution time.

The machine code for CMP AX,3B2EH is 3D 2E 3B. ■

#### ■ EXAMPLE 4.7

Let us examine the operation of CMP when both signed and unsigned numbers are used as operands. Consider the following instruction sequence:

```
MOV AL, 20
CMP AL, 10
CMP AL, 30
```

When the first `CMP` instruction executes, both the sign and carry flags will be cleared because the accumulator (20) was larger than the immediate data (10). The internal subtraction  $20 - 10$  gives a positive result.

When the second `CMP` instruction executes, the internal subtraction performed is  $20 - 30$ . In this case, the sign and carry flags are both set (due to the negative result).

All three numbers used (10, 20, and 30) represent *positive* numbers. What happens when *negative* numbers are used? The three instructions are now as follows:

```
MOV  AL, 90H      ;represents -112
CMP  AL, 80H      ;represents -128
CMP  AL, 0A0H     ;represents -96
```

Remember that negative numbers are represented using 2's complement notation.

The sign and carry flags are both cleared during execution of the first `CMP` instruction, because  $-112$  minus  $-128$  ( $90H$  minus  $80H$ ) results in a positive value. When the second `CMP` instruction is executed, the sign and carry flags are both set as a result of the internal subtraction  $-112$  minus  $-96$  ( $90H$  minus  $A0H$ ).

When both positive and negative numbers are compared, as in:

```
MOV  AL, 10
CMP  AL, 90H
```

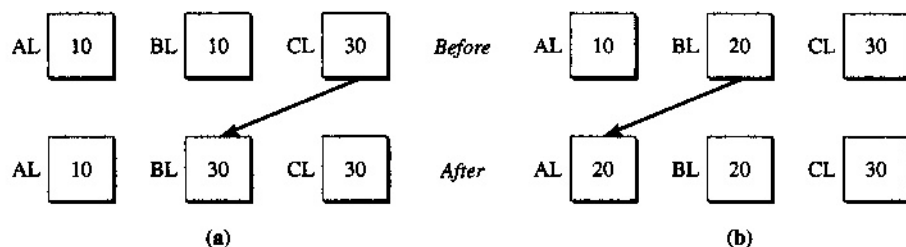
the sign and carry flags are affected differently. In this case, the sign flag is cleared and the carry flag is set. Can you determine why? ■

**CMPXCHG Destination, Source (Compare and Exchange).** This instruction compares the destination operand with the accumulator (AL, AX, or EAX, depending on the size of the destination). The flags are set accordingly. If the accumulator equals the destination, the source operand is copied into the destination. If the accumulator and destination operands are different, the accumulator is replaced by the value in the destination.

#### ■ EXAMPLE 4.8

If AL, BL, and CL contain the respective values 10H, 20H, and 30H, what is the result of `CMPXCHG BL, CL`? What is the result if BL initially equals 10H?

**Solution:** Figure 4.3 shows the results of each `CMPXCHG` execution. ■



**FIGURE 4.3** Execution of `CMPXCHG BL, CL` (a) when AL equals BL and (b) when AL does not equal BL

In both cases, data is rapidly exchanged between two registers. This type of data exchange is especially useful in operating system software that supports multiple processes through the use of semaphores. A semaphore is essentially a counter whose value decides which process is able to execute next. It is often necessary to update the semaphore value with a single instruction. When multiple instructions are used, such as:

```

CMP     AL,BL
JNZ     NOTEQUAL    ;jump to NOTEQUAL if AL <> BL
MOV     BL,CL
JMP     NEXT
NOTEQUAL MOV    AL,BL
NEXT     ---

```

it is possible to incorrectly update the semaphore due to unexpected interrupts that switch between the two or more processes accessing the semaphore at the same time. The **CMPXCHG** instruction solves this problem by performing the equivalent work of the five example instructions.

**CMPXCHG8B Destination (Compare and Exchange 8 Bytes).** This instruction is similar to **CMPXCHG** except that the comparison is fixed at 64 bits. The **EDX** and **EAX** registers (**EDX:EAX**) specify the 64-bit operand (**EDX** being the upper 32 bits) that is compared with the destination. If equal, the destination receives a copy of the 64-bit number in **ECX:EBX** (**ECX** holding the upper 32 bits).

If the comparison is not equal, the destination value is copied into **EDX:EAX**. Only the zero flag is affected by the result of the comparison.

#### ■ EXAMPLE 4.9

What are the results of the **CMPXCHG8B** instruction given this sequence of instructions:

```

MOV     EDX,12345678H
MOV     EAX,9ABCDEF0H
MOV     ECX,01020304H
MOV     EBX,05060708H
CMPXCHG8B  VAL64BIT

```

**VAL64BIT** is a 64-bit (8-byte) memory operand, defined as follows:

```
VAL64BIT    DQ    123456789ABCDEF0H
```

**Solution:** Because the 64-bit memory operand **VAL64BIT** is equal to **EDX:EAX**, the contents of **VAL64BIT** are replaced by **ECX:EBX**. Its new value is 0102030405060708H.

Note the use of the **DQ** (define quadword) directive used to reserve room for **VAL64BIT**. ■

**MUL Source (Multiply Bytes, Words, or Double-Words Unsigned).** This unsigned multiply instruction treats byte, word, and double-word numbers as unsigned binary values. This gives an 8-bit number a range of 0 to 255, a 16-bit number the range 0 to 65,535, and a 32-bit number the range 0 to 4,294,967,296. The source operand specified in the instruction is multiplied by the accumulator. The source operand may not be immediate data. If the source is a byte, **AL** is used as the multiplier, with the 16-bit result replacing the contents of **AX**. If the source is a word, **AX** is used as the multiplier, and the 32-bit result is returned in registers **DX** and **AX**, with **DX** containing the upper 16 bits of the result. For double-word source values, the multiplier is **EAX**. The 64-bit result is stored in **EDX** and **EAX**. All flags are affected.



■ **EXAMPLE 4.10** What is the result of `MUL CL` if `AL` contains `20H` and `CL` contains `80H`? What is the result of `MUL AX` if `AX` contains `A064H`?

**Solution:** The decimal equivalents of `20H` and `80H` are 32 and 128, respectively. The product of these two numbers is 4096, which is `1000H`. Upon completion, `AX` will contain `1000H`. The decimal equivalent of `A064H` is 41,060. `MUL AX` multiplies the accumulator by itself, giving 1,685,923,600 as the result, which is `647D2710H`. The upper half of this hexadecimal number (`647DH`) will be placed into register `DX`. The lower half (`2710H`) will replace the contents of `AX`.

The machine code for `MUL CL` is `F6 E1`. The machine code for `MUL AX` is `F7 E0`. ■

**IMUL Source (Integer Multiply Bytes, Words, or Double-Words).** This multiply instruction is very similar to `MUL` except that the source operand is assumed to be a *signed* binary number. This gives byte operands a range of  $-128$  to  $127$ , word operands a range of  $-32,768$  to  $32,767$ , and double-word operands a range of  $-2,147,483,648$  to  $2,147,483,647$ . Once again the operand size determines whether the result will be placed into `AX`, `DX` and `AX`, or `EDX` and `EAX`.

■ **EXAMPLE 4.11** What is the result of `IMUL CL` if `AL` contains `20H` and `CL` contains `80H`?

**Solution:** Although this example appears to be exactly like Example 4.10, it actually is not, because the `80H` in register `CL` is interpreted as  $-128$ . The product of  $-128$  and 32 is  $-4096$ , which is `F000H` in 2's complement notation. This is the value placed into `AX` upon completion.

The machine code for `IMUL CL` is `F6 E9`. ■

■ **EXAMPLE 4.12** `IMUL` also accepts two or three operands under certain restrictions. When two operands are used, the first operand must be a 16/32-bit register. The second operand may be a register, a memory operand, or immediate data. The immediate data will be sign-extended to 16 or 32 bits. The result must be 16 or 32 bits wide. Examples are:

|                          |                   |                       |                               |
|--------------------------|-------------------|-----------------------|-------------------------------|
| <code>0F AF C3</code>    | <code>IMUL</code> | <code>AX, BX</code>   | <code>;AX = AX * BX</code>    |
| <code>0F AF 0D</code>    | <code>IMUL</code> | <code>CX, [DI]</code> | <code>;CX = CX * [DI]</code>  |
| <code>6B DB 32</code>    | <code>IMUL</code> | <code>BX, 50</code>   | <code>;BX = BX * 50</code>    |
| <code>6B C9 F4</code>    | <code>IMUL</code> | <code>CX, -12</code>  | <code>;CX = CX * -12</code>   |
| <code>66 0F AF C3</code> | <code>IMUL</code> | <code>EAX, EBX</code> | <code>;EAX = EAX * EBX</code> |

When three operands are used, the first operand must be a 16/32-bit register. The second operand may be a register or memory operand. The third operand must be an immediate value. Examples of this form are as follows:

|                                  |                   |                               |                                  |
|----------------------------------|-------------------|-------------------------------|----------------------------------|
| <code>6B C3 05</code>            | <code>IMUL</code> | <code>AX, BX, 5</code>        | <code>;AX = BX * 5</code>        |
| <code>6B CA F6</code>            | <code>IMUL</code> | <code>CX, DX, -10</code>      | <code>;CX = DX * -10</code>      |
| <code>66 69 1C 000007D0</code>   | <code>IMUL</code> | <code>EBX, [SI], 2000</code>  | <code>;EBX = [SI] * 2000</code>  |
| <code>66 69 1C FFFFFFF830</code> | <code>IMUL</code> | <code>EBX, [SI], -2000</code> | <code>;EBX = [SI] * -2000</code> |

Notice that the 32-bit size of `EBX` in the last two instructions cause the immediate data to be sign-extended to 32 bits. ■

**DIV Source (Divide Bytes, Words, or Double-Words Unsigned).** In this instruction, the accumulator is divided by the value represented by the source operand. All numbers are interpreted as *unsigned* binary numbers. If the source is a byte, the quotient is placed into AL and the remainder into AH. If the source is a word, it is divided into the 32-bit number represented by DX and AX (with DX holding the upper 16 bits). The 16-bit quotient is placed into AX and the 16-bit remainder into DX. For double-word operands, the 64-bit value in EDX and EAX is divided by the source. The 32-bit quotient is placed in EAX. The 32-bit remainder is saved in EDX. If the quotient is too large to fit in the destination, a type-0 interrupt is generated. All flags are affected.

---

■ **EXAMPLE 4.13** What is the result of `DIV BL` if AX contains 2710H and BL contains 32H?

**Solution:** The decimal equivalents of 2710H and 32H are 10,000 and 50, respectively. The quotient of these two numbers is 200, which is C8H. This is the byte placed into AL. Because the division had no remainder, AH will contain 00.

The machine code for `DIV BL` is F6 F3. ■

---

**IDIV Source (Integer Divide Bytes, Words, or Double-Words).** As with `MUL` and `IMUL`, we also see similarities between `DIV` and `IDIV`. In `IDIV` the operands are treated as signed binary numbers. Everything else remains the same.

---

■ **EXAMPLE 4.14** What is the result of `IDIV CX` if AX contains 7960H, DX contains FFEH, and CX contains 1388H?

**Solution:** Because a word operand is specified, the 32-bit number represented by DX and AX will be divided by CX. Combining DX and AX gives FFE7960H, which is -100,000. CX represents 5,000. Dividing -100,000 and 5,000 gives -20, which is FFECH. This value is placed into AX, and 0000 is placed into DX because the division has no remainder.

The machine code for `IDIV CX` is F7 F9. ■

---

■ **EXAMPLE 4.15** A simple random-number generator uses a 64-bit pattern to represent its random value. A new random number is generated by dividing the current random number by a 32-bit generator pattern. The quotient and remainder are combined to form the new random pattern, with the remainder making up the upper 32 bits. One way to do this is as follows:

Address and data in data segment:

|      |          |        |    |           |
|------|----------|--------|----|-----------|
| 0000 | 12345678 | RANDHI | DD | 12345678H |
| 0004 | 9ABCDEF0 | RANDLO | DD | 9ABCDEF0H |
| 0008 | 5A5A5A5A | GTERM  | DD | 5A5A5A5AH |

Random-number generator:

|    |            |      |             |
|----|------------|------|-------------|
| 66 | 8B 16 0000 | MOV  | EDX, RANDHI |
| 66 | A1 0004    | MOV  | EAX, RANDLO |
| 66 | 8B 1E 0008 | MOV  | EBX, GTERM  |
| 66 | F7 FB      | IDIV | EBX         |
| 66 | 89 16 0000 | MOV  | RANDHI, EDX |
| 66 | A3 0004    | MOV  | RANDLO, EAX |

When the six instructions are executed, the new values of `RANDHI` and `RANDLO` are 5296DBCCH and 33944A55H, respectively. This may certainly be viewed as a random pattern, although significant testing is required to determine how well the generator works when used repeatedly. ■

---

**NEG Destination (Negate Byte, Word, or Double-Word).** This instruction is used to find the signed 2's complement representation of the number in the destination. This is accomplished by subtracting the destination operand from 0. All flags are affected.

---

■ **EXAMPLE 4.16** What is the result of `NEG AX` if `AX` contains FFECH?

**Solution:** Subtracting FFECH from 0 gives 0014H, which is +20. Take another look at Example 4.14 and you will see that we have just proved the results of the `IDIV` instruction. The machine code for `NEG AX` is F7 D8. ■

---

**CBW (Convert Byte to Word).** This instruction is used to extend a signed 8-bit number in `AL` into a signed 16-bit number in `AX`. This is sometimes necessary before performing an `IDIV` or `IMUL`. No flags are affected.

---

■ **EXAMPLE 4.17** What does `AX` contain after execution of `CBW` if `AL` initially contains 37H? What if `AL` contains B7H?

**Solution:** Because 37H is a positive signed number, the result in `AX` is 0037H. Note that the most significant bit is still a 0. B7H, however, is a negative signed number, resulting in `AX` becoming FFB7H. Note here that the MSB is a 1, indicating a negative result.

The machine code for `CBW` is 98. ■

---

**CWD/CWDE (Convert Word to Double-Word).** `CWD` extends the sign of the number stored in `AX` through all 16 bits of register `DX`. This results in a 32-bit signed number in `DX` and `AX`. `CWD` is useful when preparing for `IMUL` and `IDIV`. `CWDE` also extends the sign of `AX` but does it in the upper 16 bits of `EAX`. The 32-bit result is in one register. No flags are affected.

---

■ **EXAMPLE 4.18** What is in `DX` after `CWD` executes, with `AX` containing 4000H? What if `CWDE` is executed?

**Solution:** Because 4000H is positive (the MSB is 0), `DX` will contain 0000. When `CWDE` executes, `EAX` will contain 00004000H.

The machine code for `CWD` is 99. The machine code for `CWDE` is 66 98. ■

---

**CDQ (Convert Double-Word to Quadword).** This instruction is similar to `CWD`. The sign bit of `EAX` is extended through `EDX`. This gives a 64-bit result in `EDX:EAX`.

---

**■ EXAMPLE 4.19** What are the results of CDQ if EAX contains C8000000H?

**Solution:** Because the MSB of EAX is a one, the contents of EDX are set to FFFFFFFFH. The machine code for CDQ is 66 99. ■

---

**DAA (Decimal Adjust for Addition).** When a hexadecimal number contains only the digits 0–9 (as in 07H, 35H, 72H, and 98H), the number is referred to as a **packed decimal number**. So, instead of a range of 00 to FFH we get 00 to 99H. When these types of numbers are added together, they do not always produce the correct packed decimal result. Consider the addition of 15H and 25H. Straight hexadecimal addition gives 3AH as the result. Because we are interpreting our numbers as packed decimal numbers, 3AH seems to be an illegal answer. The processor is able to correct this problem with the use of DAA, which will modify the byte stored in AL so that it looks like a packed decimal number. Numbers of this type are also referred to as **binary coded decimal (BCD)** numbers.

All flags are affected.

---

**■ EXAMPLE 4.20** If ADD AL,BL is executed with AL containing 15H and BL containing 25H, what is the resulting value of AL? What does AL contain if DAA is executed next?

**Solution:** Adding 15H to 25H gives 3AH, as we just saw. When DAA executes, it will examine AL and determine that it should be corrected to 40H to give the correct packed decimal answer. It is interesting that the processor adds 06H to AL to convert it into a packed number.

The machine code for DAA is 27. ■

---

**DAS (Decimal Adjust for Subtraction).** This instruction performs the same function as DAA except it is used after a SUB or SBB instruction. All flags are affected.

---

**■ EXAMPLE 4.21** If AL contains 10H and CL contains 02H, what is the result of SUB AL,CL? What happens if DAS is executed next?

**Solution:** Subtracting 02H from 10H will result in AL containing 0EH. Following the SUB instruction with DAS will cause AL to be converted into 08H, the correct packed decimal answer.

The machine code for DAS is 2F. ■

---

**AAA (ASCII Adjust for Addition).** Occasionally the need arises to perform addition on ASCII numbers. The ASCII codes for 0–9 are 30H through 39H. Addition of two ASCII codes unfortunately does not result in a correct ASCII or decimal answer. For example, adding 33H (the ASCII code for 3) and 39H (the ASCII code for 9) gives 6CH. Using DAA to correct this result will give 72H, which is the correct packed decimal answer, but not the answer we are looking for. After all, 3 plus 9 equals 12, which must be represented by the ASCII characters 31H and 32H. AAA is used to perform this correction by using it after ADD or ADC. AAA will examine the contents of AL, adjusting the lower 4 bits to make the correct decimal result. Then it will clear the upper 4 bits and add 1 to AH if the lower 4 bits of AL were initially greater than 9. Only the carry and auxiliary carry flags are directly affected.

■ **EXAMPLE 4.22** Register AX is loaded with 0033H. Register BL is loaded with 39H. ADD AL,BL is executed, giving AL a new value of 6CH. What happens if AAA is executed next?

**Solution:** AAA will see the C part of 6CH and correct it to 2 (by adding 6). Next, the upper 4 bits of AL will be cleared. Now AL contains 02. Because the lower 4 bits of AL (prior to execution of AAA) were greater than 9, 1 will be added to AH, making its final value 01. We end up with AX containing 0102H. If we now add 30H to each byte in AX, we will get 3132H, the two ASCII digits that represent 12.

The machine code for AAA is 37. ■

**AAS (ASCII Adjust for Subtraction).** This instruction performs the same correction procedure that AAA does, except it is used after SUB or SBB to modify the results of a subtraction. Also, if the number in the lower 4 bits of AL is greater than 9, 1 will be *subtracted* from AH.

■ **EXAMPLE 4.23** AX is loaded with 0037H and BL is loaded with 32H. SUB AL,BL is executed and the processor replaces the contents of AL with 05H. What happens if AAS is now executed? What are the results if BL was initially loaded with 39H?

**Solution:** Because the number in the lower 4 bits of AL (5) is not greater than 9, AAS does not change it. The upper 4 bits of AL are cleared. Because no change was needed, there is no need to add 1 to AH. The final value of AX is 0005H. Adding 30H to each byte in AX gives 3035H, the two ASCII codes for the number 05.

If BL is initially loaded with 39H, the result of SUB AL,BL is FEH, which is placed in AL. When this value is examined by AAS, the E in FEH will be changed to 8 (by subtracting 6) and the upper 4 bits will be cleared. AL now contains 08H. Note that 7 minus 9 is -8 using 10s complement BCD arithmetic. AL now contains the 8 part of the answer. Because AL required modification, AH will be decremented. The final value of AX is FF08H (-2 in BCD). The FFH in AH indicates that a borrow occurred (and so does the carry flag). Adding 30H to AL results in the correct ASCII code for 8 (which is 38H).

The machine code for AAS is 3F. ■

**AAM (ASCII Adjust for Multiply).** This instruction is used to adjust the results of a MUL instruction so that the results can be interpreted as ASCII characters. Both the operands used in the multiplication must be less than or equal to 9 (their upper 4 bits must be 0). This ensures that the upper byte of the result placed in AX will be 00, a necessary requirement for proper operation of AAM. AAM will convert the binary product found in AL into two separate digits. One digit replaces the contents of AL and the other digit replaces the contents of AH. Software can then be used to add 30H to each value to obtain the ASCII codes for the correct product. Only the parity, sign, and zero flags are directly affected.

■ **EXAMPLE 4.24** AL and BL are given initial values of 7 and 6, respectively. MUL BL produces 002AH in AX. Note that 7 times 6 is 42 and that 2AH is equal to 42. What are the contents of AX after AAM is executed?

**Solution:** Because the upper 4 bits of each operand used in the multiplication were 0 and the lower 4 bits contained numbers less than or equal to 9, AAM will be able to correctly convert the product in AX. AAM converts the 2AH in AL into the digits 4 and 2 and writes

them in AH and AL, respectively. The final contents of AX are 0402H. If we now add 30H to each byte in AX, we get 3432H, the two ASCII codes for 42.

The machine code for AAM is D4 0A. ■

**AAD (ASCII Adjust for Division).** This instruction is executed *before* DIV to change an unpacked two-digit BCD number in AX into its binary equivalent in AL. AH must be 00 after the change to produce the correct results. The number in AL can then be divided (via DIV) to get the correct binary quotient in AL. Only the parity, sign, and zero flags are directly affected.

■ **EXAMPLE 4.25** Register AX is loaded with 0600H. These 2 bytes correspond to the unpacked BCD number 60, not its decimal equivalent of 1,536. If BL is loaded with 04H, we expect that the use of AAD and DIV will produce the correct answer of 15 (which is 60 divided by 4). What are the contents of AX after AAD and after DIV BL?

**Solution:** AAD examines AX and finds that it contains a valid unpacked BCD number. The value 600H is converted to 003CH, the correct binary equivalent of 60. DIV BL then divides this number by 4 to produce 000FH in AX. Remember that AL contains the quotient and AH contains the remainder. The quotient 0FH equals 15 and the remainder is 0, so AAD correctly adjusted AX before the division.

The machine code for AAD is D5 0A. ■

**XADD Destination,Source (Exchange and Add Bytes, Words, or Double-Words).** In this instruction, the source operand is a register and the destination is a register or memory location. The source and destination operands are added together, with the sum replacing the contents of the destination. The original destination is copied into the source register.

All flags are affected.

■ **EXAMPLE 4.26** What is the result of XADD AX,BX if AX contains 2000H and BX contains 3000H?

**Solution:** The sum of AX and BX is 5000H. This becomes the new value stored in AX. BX is loaded with 2000H.

The machine code for XADD is 0F C1 D8. ■

## 4.3 LOGICAL INSTRUCTIONS

The instructions in this group are used to perform Boolean (logical) operations on binary data.

### The Instructions

**NOT Destination ("NOT" Byte, Word, or Double-Word).** This instruction finds the complement of the binary data stored in the destination operand. All 0s are changed to 1s, and all 1s to 0s. No flags are affected.

■ **EXAMPLE 4.27** What is the result of `NOT AL` if AL contains 3EH?

**Solution:** The binary equivalent of 3EH is 00111110. When these bits are complemented we get 11000001, which is C1H. This is the final value in AL.

The machine code for `NOT AL` is F6 D0. ■

**AND Destination,Source ("AND" Bytes, Words, or Double-Words).** This instruction performs the logical AND operation on the destination and source operands. AND is a useful way of turning bits off (making them 0). Refer to Figure 4.4 for a review of the truth table for a two-input AND gate. Note that it takes two 1s to make a 1 on the output of the AND gate. The processor ANDs the bits in each operand together in a bitwise fashion (bit 0 of source with bit 0 of destination, etc.) to produce the final result. This logical operation is very useful for masking off unwanted bits during many types of comparison operations. All flags are affected.

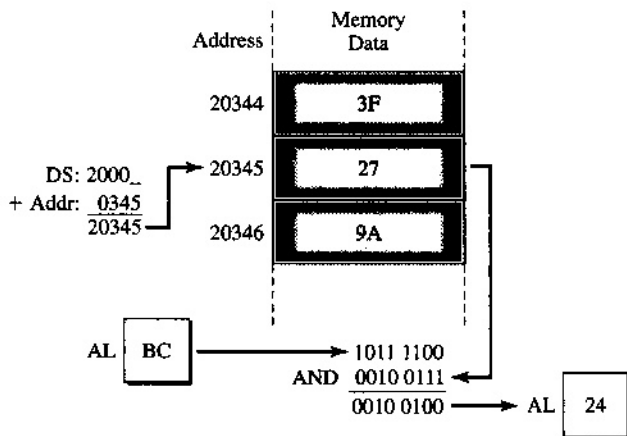
**FIGURE 4.4** Truth table for an AND gate

| A | B | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

■ **EXAMPLE 4.28** What is the result of `AND AL,[345H]`? Assume that the DS register contains 2000H and that AL contains BCH.

**Solution:** Figure 4.5 shows how the processor accesses location 20345H and ANDs its contents (27H = 00100111) with that of the AL register (BCH = 10111100). The result is 24H, which is placed into AL.

**FIGURE 4.5** Operation of `AND AL,[345H]`



The machine code for `AND AL,[345H]` is 22 06 45 03. ■

**OR Destination,Source ("OR" Bytes, Words, or Double-Words).** This instruction performs a logical OR of each bit in the source and destination operands. OR can be used to turn bits on (make them 1). Figure 4.6 shows the truth table for a two-input OR gate. The output of

**FIGURE 4.6** Truth table for an OR gate

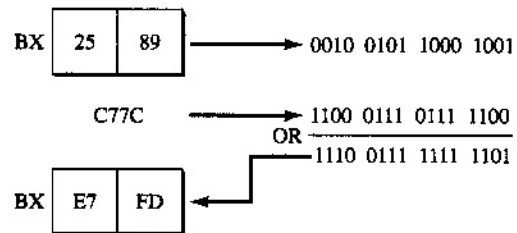
| A | B | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

the OR gate will be high if a 1 is present on either (or both) inputs. Respective bits in each operand are ORed together to produce the final result. All flags are affected.

■ **EXAMPLE 4.29** Register BX contains 2589H. What is the result of OR BX,0C77CH?

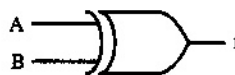
**Solution:** As a matter of habit, whenever we use a hexadecimal number in an instruction, we precede it with a leading 0 if the first hexadecimal digit is greater than 9. So, the correct way to specify C77CH in the OR instruction is to write 0C77CH. The purpose for doing this is to prevent the assembler from thinking that C77CH is the name of a label somewhere in the program. Putting a leading 0 first does not change the value of the hexadecimal number, but it does tell the assembler that it is working with a number and not a name.

So, in Figure 4.7 we see how the OR instruction operates on BX and the immediate data to produce the final result of E7FDH in register BX.

**FIGURE 4.7** Operation of OR BX,0C77CH

The machine code for OR BX,0C77CH is 81 CB 7C C7. ■

**XOR Destination,Source (Exclusive-OR Bytes, Words, or Double-Words).** XOR is a special form of OR in which the inputs *must be different* to get a 1 out of the logic gate. Figure 4.8 shows the truth table for a two-input XOR gate. Clearly, the output is high only when the inputs are different. The XOR function can be used to toggle bits to their opposite states. For example, 011 XORed with 010 gives 001. Note that the second bit in the first group has been changed to 0. If we XOR 001 with 010 again, we get 011. The two XOR operations simply toggle the second bit back and forth. This is due to the 1 in the second group of bits. It may be interesting to note that XOR AL,0FFH does the same job as NOT AL. All flags are affected by XOR.

**FIGURE 4.8** Truth table for an Exclusive-OR gate

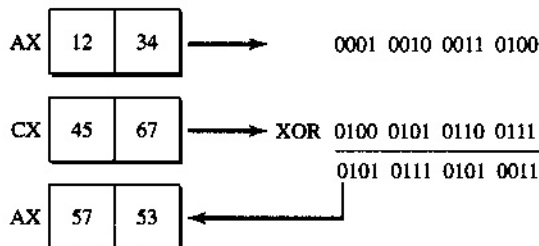
| A | B | f |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



■ **EXAMPLE 4.30** What is the result of XOR AX,CX if AX contains 1234H and CX contains 4567H?

**Solution:** Figure 4.9 shows how the two 16-bit binary numbers in AX and CX are XORed together. The result is placed in AX.

**FIGURE 4.9** Operation of XOR AX,CX



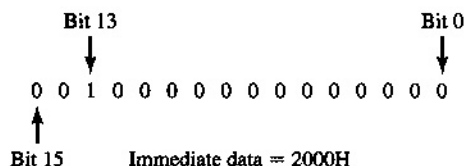
The machine code for XOR AX,CX is 31 C8. ■

**TEST Destination,Source ("TEST" Bytes, Words, or Double-Words).** TEST can be used to examine the state of individual bits, or groups of bits. Internally, the processor performs an AND operation on the two operands without changing their contents. The flags are updated based on the results of the AND operation. In this way, testing individual bits is easily accomplished by putting a 1 in the correct bit position of one operand and examining the zero flag after TEST executes. If the bit being tested was high, the zero flag will be cleared (a *not-zero* condition). If the bit was low, the zero flag will be set.

■ **EXAMPLE 4.31** What instruction is needed to test bit 13 in register DX?

**Solution:** We must determine the immediate data needed to perform the desired test. Figure 4.10 shows how the 1 needed to perform the test requires the immediate word 2000H. Because immediate data cannot be used as a destination operand, we end up with the instruction TEST DX,2000H. It will be necessary to examine the flags (possibly only the zero flag) to decide what to do with the results of the test.

**FIGURE 4.10** Operation of TEST DX,2000H



The machine code for TEST DX,2000H is F7 C2 00 20. ■

**BSF/BSR Destination,Source (Bit Scan Forward/Reverse).** The BSF instruction scans the source operand for the first bit that equals 1, beginning with the LSB. The bit position (index) of the first 1 found is saved in the destination. The destination must be a 16- or 32-bit register. The source may be a 16- or 32-bit register or memory operand.

BSR operates in a similar manner, except for the direction of the scan, which begins with the MSB instead.

■ **EXAMPLE 4.32** Register EBX contains 4CE00000H. What is the result of BSF EAX,EBX? What happens if BSR is used?

**Solution:** As indicated in Figure 4.11, BSF begins scanning for a 1 at bit 0. The first 1 is found at bit 21. Register EAX is loaded with 15H (21 decimal).

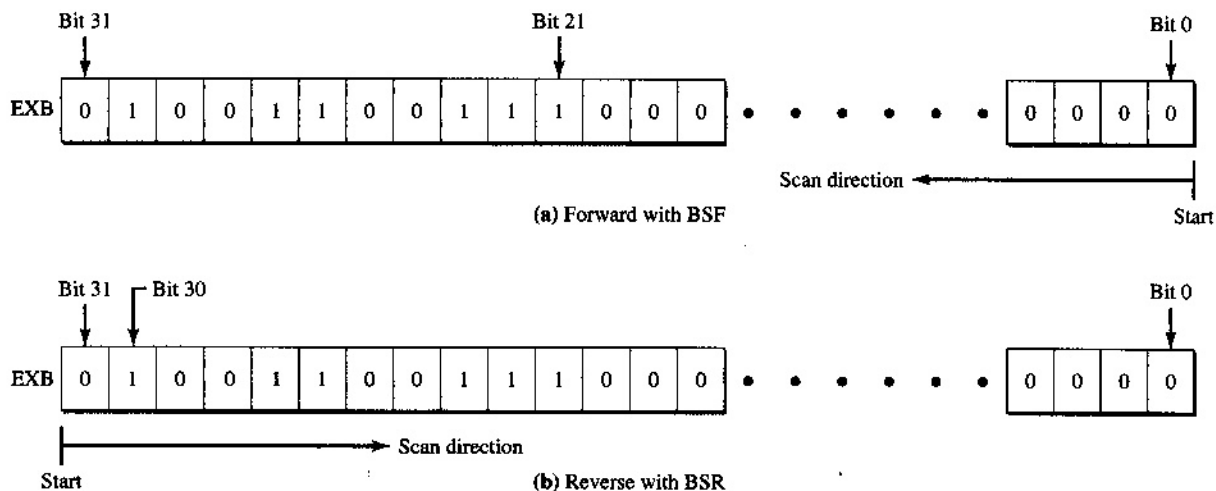


FIGURE 4.11 Scanning bits in EBX

When BSR EAX,EBX is used, the result in EAX is 1EH (30 decimal), because scanning begins with the MSB. This indicates that the first 1 found is bit 30.

The machine code for BSF EAX,EBX is 66 0F BC C3. The machine code for BSR EAX,EBX is 66 0F BD C3. ■

■ **EXAMPLE 4.33** A very useful application of BSF/BSR is edge detection in an image-processing application. A special image called a *binary image* that contains only black and white colors has the property that an edge exists anywhere there is a zero to one transition. For example, consider this 32-bit sample from a binary image (zero is black, one is white):

00000000011111110000000000000000

The edges of the seven-pixel wide bar occur at bit positions 16 and 22. These positions may be easily found using BSF and BSR. ■

**BT/BTC/BTS/BTR** *Destination,Source (Bit Test and Complement/Set/Reset)*. These four instructions are used to determine the value of a specific bit in the 16- or 32-bit destination operand (register or memory). The bit to be tested is indicated by the source operand, which may be a register or an immediate value. Bits are numbered from 0 to 31.

The state of the bit that is tested is copied into the carry flag. Thus, if the bit is zero, the carry flag will be zero.

After testing the bit, the BTC instruction complements it. The destination is modified. The BTS instruction tests and then sets the indicated bit. BTR tests and resets the chosen bit.

■ **EXAMPLE 4.34** The contents of register AX and the carry flag are shown after each instruction executes:

```

B8 5555      MOV     AX,5555H      ;AX = 5555,carry = 0
0F BA E0 0A   BT      AX,10        ;AX = 5555,carry = 1
0F BA E0 0B   BT      AX,11        ;AX = 5555,carry = 0
BB 000A      MOV     BX,10         ;AX = 5555,carry = 0
0F A3 D8      BT      AX,BX        ;AX = 5555,carry = 1
43           INC     BX
0F A3 D8      BT      AX,BX        ;AX = 5555,carry = 0
0F BA F8 0F   BTC     AX,15        ;AX = D555,carry = 0
0F BA E8 01   BTS     AX,1         ;AX = D557,carry = 0
0F BA F0 00   BTR     AX,0         ;AX = D556,carry = 1

```

In control applications, a single bit is often used to operate a device (open/close a relay, turn an indicator light on/off) or sense a specific input condition (door open/closed, control switches). The bit test instructions provide a simple way to perform all the necessary operations on a single bit. This makes the job of writing the control system code quicker and easier. ■

**SETcc Destination (Set Byte on Condition).** This instruction tests the specified condition and sets the destination byte to 01H if true. If the condition is false, the destination is set to zero. The true/false condition is based on the state of one or more processor flags. Table 4.1 shows all the conditions that may be tested with SETcc.

**TABLE 4.1** Conditions used with SETcc

| Instruction | Meaning                        | Instruction | Meaning                          |
|-------------|--------------------------------|-------------|----------------------------------|
| SETA        | Set byte if above              | SETNE       | Set byte if not equal            |
| SETAE       | Set byte if above or equal     | SETNG       | Set byte if not greater          |
| SETB        | Set byte if below              | SETNGE      | Set byte if not greater or equal |
| SETBE       | Set byte if below or equal     | SETNL       | Set byte if not less             |
| SETC        | Set byte if carry              | SETNLE      | Set byte if not less or equal    |
| SETE        | Set byte if equal              | SETNO       | Set byte if not overflow         |
| SETG        | Set byte if greater            | SETNP       | Set byte if not parity           |
| SETGE       | Set byte if greater or equal   | SETNS       | Set byte if not sign             |
| SETL        | Set byte if less               | SETNZ       | Set byte if not zero             |
| SETLE       | Set byte if less or equal      | SETO        | Set byte if overflow             |
| SETNA       | Set byte if not above          | SETP        | Set byte if parity               |
| SETNAE      | Set byte if not above or equal | SETPE       | Set byte if parity even          |
| SETNB       | Set byte if not below          | SETPO       | Set byte if parity odd           |
| SETNBE      | Set byte if not below or equal | SETS        | Set byte if sign                 |
| SETNC       | Set byte if not carry          | SETZ        | Set byte if zero                 |

■ **EXAMPLE 4.35** What does SETNZ AL do?

**Solution:** SETNZ checks the state of the zero flag. If the zero flag indicates an NZ condition (zero flag is clear), register AL is set to 01H. Otherwise, AL is set to 00H.

The machine code for SETNZ AL is 0F 95 C0. ■

## 4.4 BIT MANIPULATION INSTRUCTIONS

This group of instructions is used to shift or rotate bits left or right in register or memory operands. These operations are very useful when converting data from one form to another or for manipulating specific patterns, such as a single bit that moves through each position of a register.

### The Instructions

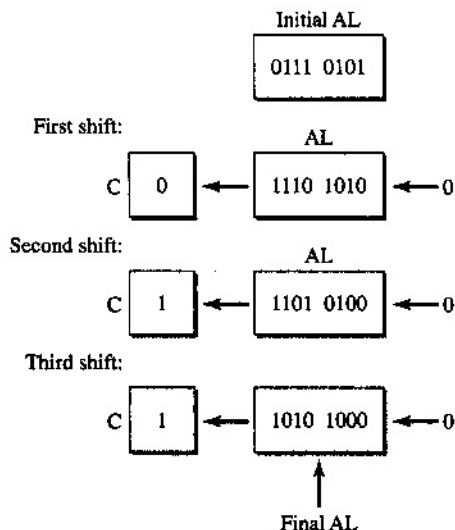
**SHL/SAL Destination, Count (Shift Logical/Arithmetic Left Byte, Word, or Double-Word).** This is the first of a number of different instructions that we will encounter that go by two names. The programmer may have personal reasons for using SHL instead of SAL, or vice versa, but to the assembler both names are identical and generate the same machine code.

The count operand indicates how many bits the destination operand is to be shifted to the left. Shifts may be from 1 to 31 bits and may be specified as an immediate value, as in `SHL AX, 7`. For multi-bit shifts, the count may also be placed in register CL. The corresponding instruction for a multi-bit shift would then be `SHL AL, CL`. All bits in the destination operand are shifted left, with a 0 entering the LSB each time. Bits that shift out of the MSB position are placed into the carry flag. Other flags are modified according to the final results.

■ **EXAMPLE 4.36** What are the results of `SHL AL, CL` if AL contains 75H and CL contains 3?

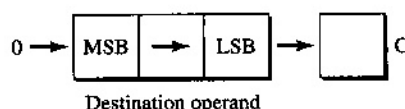
**Solution:** Figure 4.12 shows the state of register AL after each shift is performed. A 0 is shifted in from the right each time. The last bit shifted out of the MSB position is the final state of the carry flag. The final contents of AL are A8H. Could the same results be obtained by executing `SHL AL, 1` three times? The answer is yes, at the expense of generating more code and consuming more execution time.

**FIGURE 4.12** Execution of `SHL AL, CL`



This type of shift instruction is very useful for computing powers of 2, because each time a binary number is shifted left it doubles in value.

The machine code for `SHL AL, CL` is `D2 E0`. The machine code for `SHL AL, 1` is `D0 E0`. ■

**FIGURE 4.13** Operation of SHR

**SHR** *Destination, Count* (Shift Logical Right Byte, Word, or Double-Word). This instruction has the opposite effect of SHL, with bits shifting to the right in the destination operand. Zeros are shifted in from the left and the bits that fall out of the LSB position end up in the carry flag. Figure 4.13 details this operation. Keep in mind that shifting right 1 bit is equivalent to dividing by 2.

■ **EXAMPLE 4.37** What instruction (or instructions) is needed to divide the number in BX by 32?

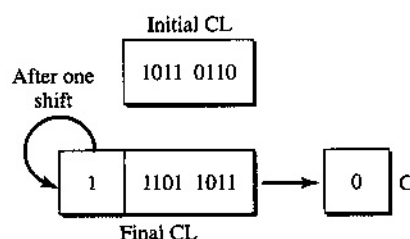
**Solution:** Because 32 equals 2 raised to the fifth power, shifting BX to the right five times is equivalent to dividing it by 32. The shift count 5 must be placed into register CL before the shift instruction `SHR BX,CL` is executed. An alternate solution would involve execution of `SHR BX,5`.

The machine code for `SHR BX,CL` is D3 EB. The machine code for `SHR BX,5` is C1 EB 05. ■

**SAR** *Destination, Count* (Shift Arithmetic Right Byte, Word, or Double-Word). SAR is very similar to SHR with the important difference being that the MSB is shifted back into itself. This serves to preserve the original sign of the destination operand. When we deal with signed numbers, the MSB is used as the sign bit. If SHR is used to shift a signed negative number to the right, the 0 that gets shifted into the MSB position forces the computer to then interpret the result as a positive number. SAR prevents this from happening. Negative numbers stay negative and positive numbers stay positive, and each gets smaller with every shift right.

■ **EXAMPLE 4.38** What are the results of `SAR CL,1` if CL initially contains B6H?

**Solution:** B6H is the signed 2's complement representation of -74. Figure 4.14 shows how the contents of register CL are shifted right 1 bit, with the MSB coming back in from the left to preserve the negative sign. The final value in CL is DBH, which corresponds to -37, exactly one-half of the original number. A 0 is shifted out of the LSB into the carry flag.

**FIGURE 4.14** Operation of SAR CL,1

The machine code for `SAR CL,1` is D0 F9. ■

**SHLD/SHRD** *Destination,Source,Count (Shift Left/Right Double Precision)*. These instructions are similar to SHL and SHR, with the exception that instead of simply shifting zeros into the destination from left or right, bits from the source operand are shifted in. The destination operand may be a 16- or 32-bit register or memory location. The source operand must be a 16- or 32-bit register. The count may be specified by the value in register CL, or by an immediate value.

SHLD shifts the destination left the number of bits specified by the count. Bits from the source operand, beginning with the MSB, are shifted into the LSB of the destination.

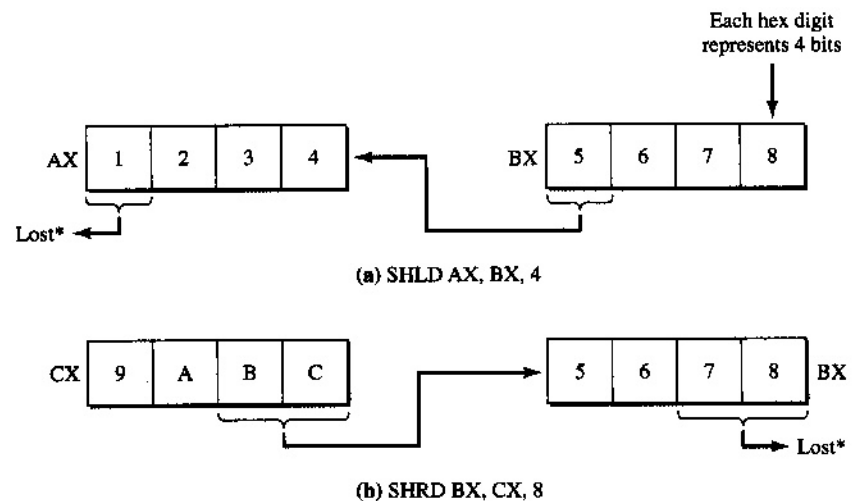
SHRD shifts the destination right the number of bits specified by the count. Source operand bits, beginning with the LSB, are shifted into the MSB of the destination.

The source operand is not affected.

■ **EXAMPLE 4.39** Registers AX, BX, and CX contain the following values, respectively: 1234H, 5678H, and 9ABCH. What are the results of these instructions?

```
SHLD  AX, BX, 4
SHRD  BX, CX, 8
```

**Solution:** As indicated in Figure 4.15(a), the SHLD instruction shifts 4 bits from BX into AX. The final result in AX is 2345H. SHRD shifts 8 bits from CX into BX, giving BX a final value of BC56H. This is illustrated in Figure 4.15(b).



\*Except carry is set to last bit shifted out

**FIGURE 4.15** Double-precision shifts

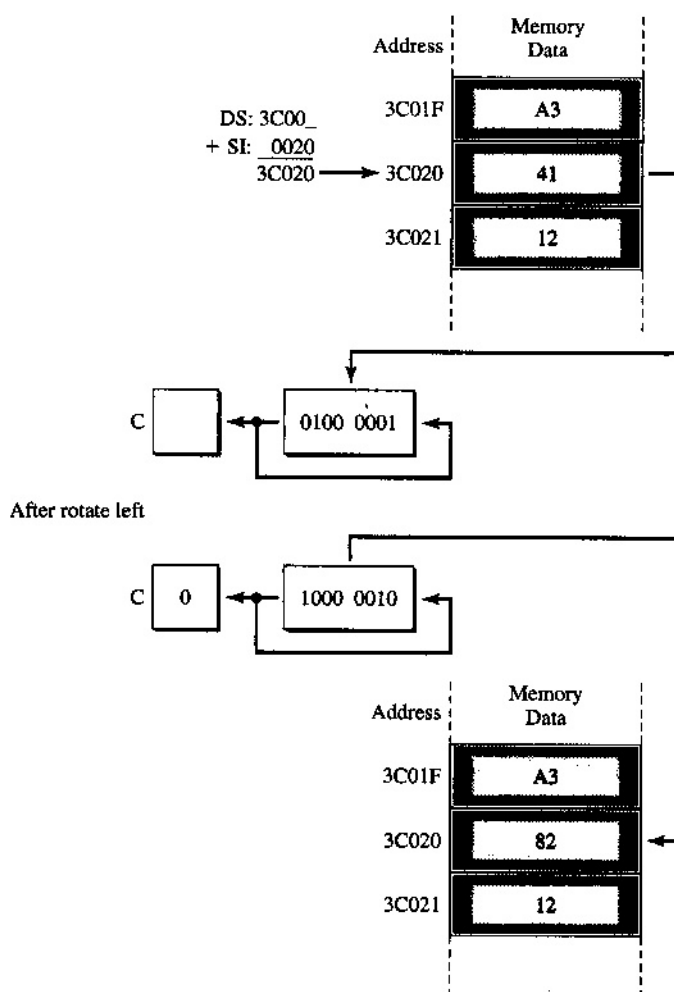
The machine code for SHLD AX,BX,4 is 0F A4 D8 04. The machine code for SHRD BX,CX,8 is 0F AC CB 08. ■

**ROL** *Destination,Count (Rotate Left Byte, Word, or Double-Word)*. The difference between ROL and SHL is that bits that get rotated out of the LSB position get rotated back into the

MSB. Thus, data inside the destination operand is never lost, only circulated within itself. A copy of the bit that rotates out of the LSB is placed into the carry flag. The only other flag affected is the overflow flag.

■ **EXAMPLE 4.40** What is the result of `ROL byte ptr [SI],1`? Assume that the DS register contains 3C00H and that register SI contains 0020H.

**FIGURE 4.16** Operation of `ROL BYTE PTR [SI],1`



**Solution:** In Figure 4.16 the DS and SI registers combine to form an effective address of 3C020H. The byte stored at this location is rotated 1 bit left, resulting in a final value of 82H. The 0 rotated out of the MSB is placed in the carry flag.

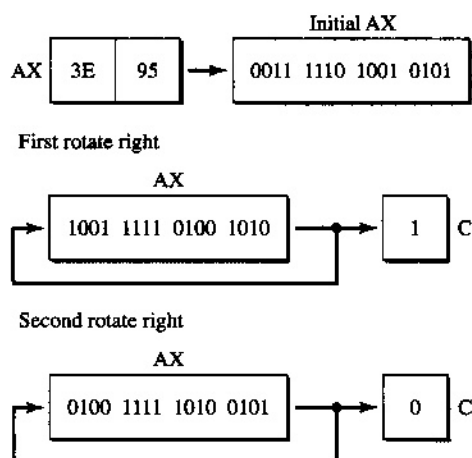
The machine code for `ROL byte ptr [SI],1` is D0 04. ■

**ROR** *Destination, Count (Rotate Right Byte, Word, or Double-Word).* This instruction has the opposite effect of ROL, with bits moving to the right within the destination operand. The bit that rotates out of the LSB goes into the carry flag and also into the MSB.

■ **EXAMPLE 4.41** What is the result of executing `ROR AX,1` twice? AX initially contains 3E95H.

**Solution:** Figure 4.17 shows the results of each rotate right operation. The final value in AX is 4FA5H. The carry flag is cleared as a result of the second shift.

**FIGURE 4.17** Operation of two `ROR AX,1` instructions



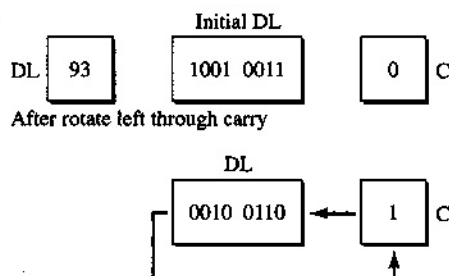
The machine code for `ROR AX,1` is D1 C8. ■

**RCL Destination, Count (Rotate Left Through Carry Byte, Word, or Double-Word).** The operation of RCL is similar to ROL except for how the carry flag is used. ROL is an 8-, 16-, or 32-bit rotate. RCL is a 9-, 17-, or 32-bit rotate. The bit that gets rotated out of the MSB goes into the carry flag and the bit that was in the carry flag gets rotated into the LSB. By controlling the carry flag, we can place new data into the destination operand, if that is our desire.

■ **EXAMPLE 4.42** What is the result of executing `RCL DL,1`? Assume that the carry flag is initially cleared and that DL contains 93H.

**Solution:** The contents of DL are rotated left 1 bit through the carry. The 0 in the carry flag rotates into the LSB of DL. The 1 in the MSB of DL rotates out of the register into the carry flag. This is shown in Figure 4.18. The final value in DL is 26H.

**FIGURE 4.18** Operation of `RCL DL,1`



The machine code for `RCL DL,1` is D0 D2. ■

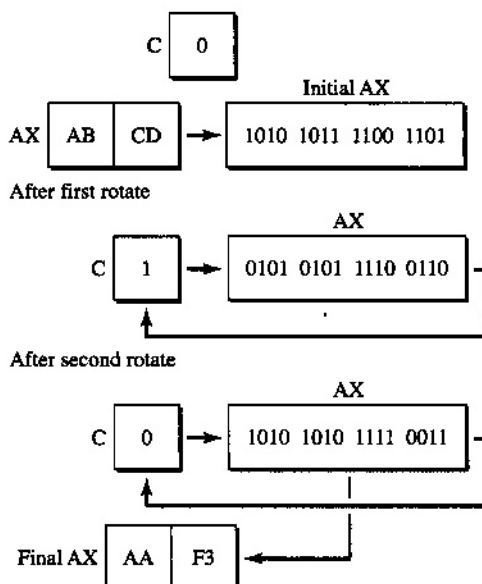


**RCR** *Destination, Count* (Rotate Right Through Carry Byte, Word, or Double-Word). This instruction complements the operation of RCL, rotating data bits right within the destination operand. The bit rotated out of the LSB goes into the carry flag. The bit that comes out of the carry flag goes into the MSB.

■ **EXAMPLE 4.43** What is in AX after execution of `RCR AX, CL` if CL contains 2 and AX contains ABCDH? The carry flag is initially cleared.

**Solution:** Figure 4.19 shows the result of each rotate right operation. The final value in AX is AAF3H. The carry flag is cleared after the second rotate.

**FIGURE 4.19** Operation of `RCR AX, CL`



The machine code for `RCR AX, CL` is D3 D8. ■

## 4.5 PROGRAM TRANSFER INSTRUCTIONS

This group of instructions allows the programmer to choose where the processor fetches its next instruction from. There are two main ways to change the execution path of a program. In general, the programmer may cause the program to jump to a new instruction location or change its path temporarily by the use of a subroutine. Let us see how this is done.

### Loop and Jump Instructions

Normally, the processor fetches instructions from sequential memory locations and places them into an instruction queue. Many instructions simply perform a specific operation on one of the processor's registers or on the data contained in a memory location. There are, however, many times when the results of an instruction need to be interpreted. Did the last instruction set the zero flag? Was the result of the last subtraction negative? Did a 0 rotate into bit 7 of register BL? There may be many different choices to make during execution,

and these choices require a number of different portions of machine code. The program needs to be able to change its execution path to respond appropriately. For example, in a program designed to play tic-tac-toe it makes a big difference who goes first and whether you put an X in the center or not. The machine language responsible for handling these differences may need to be executed in a different order each time the game is played. For this reason we need to be able to change the path of program execution by forcing the processor to fetch its next instruction from a new location. This is commonly referred to as a **jump**. A jump alters the contents of the processor's instruction pointer (and sometimes its CS register as well). Remember that the CS and IP registers are combined to determine the address of the next instruction fetch. If we change the contents of these registers, the next instruction is fetched from a new address. The CS and IP registers define a 64KB segment of memory. Any instruction that jumps to an address within this segment performs an intrasegment transfer. For example, an instruction located within a code segment beginning at address 10000H can perform an intrasegment transfer to address 1C000H. But what about instructions located in other areas of the system memory? Suppose that a certain 80x86-based system has 32KB of EPROM located at address 48000H. It is not possible for an instruction located in the code segment at 10000H to do an intrasegment jump to one at 48000H. In this case an **intersegment transfer** is needed. The difference between intrasegment transfers and intersegment transfers is that an intrasegment transfer requires a change only in the IP register, whereas the intersegment transfer requires both the CS and IP registers to be modified. For example, if the CS register contains 1000H, then all values of the IP from 0000 to FFFFH generate addresses 10000H to 1FFFFH. No other addresses are possible. If the CS register instead contained 4800H, then the range of reachable addresses is from 48000H to 57FFFH. Do you see why an intersegment transfer needs also to modify the CS register?

When the assembler examines a jump instruction, it is often able to determine what kind of jump is required, but many times it needs specific guidance from the programmer to generate the correct code. Assembler directives near and far are used to indicate intrasegment and intersegment transfers and are included directly in the source statement. For instance, `JMP FAR PTR TESTBIT` indicates an intersegment transfer to the address associated with the label TESTBIT. The machine code generated for this instruction would contain information for both the CS and IP registers. `JMP NEXTITEM`, on the other hand, is assumed to contain a near label. NEXTITEM must be contained within the current code segment.

**JMP Target (Unconditional Jump to Target).** The word unconditional in the description of this instruction means that no condition has to be met for the processor to jump. *JMP always* causes a program transfer (the IP register is loaded with a new address). The target operand may be a near or far variable and must be indicated as such for the assembler to produce machine code. In a **direct jump**, the target address is given in the instruction. In an **indirect jump**, the target address might be contained in a register or memory location. A special form of the direct jump, called the *short jump*, is used whenever the target address is within +127 or -128 bytes of the current instruction pointer (IP) address. Instead of specifying the actual target address within the instruction, a **relative offset** is used instead. In the case of the short jump, the relative offset is coded in a single byte, which is interpreted as a signed binary number. Thus, short jumps are capable of moving forward up to 127 locations and backward up to 128 locations. When the target address is outside of this range, but still within the same segment, a near jump is used and the target address is coded as a 2-byte offset within the segment. Far jumps require 4 bytes for addresses: 2 for the new IP address and 2 for the new CS address.

No flags are affected.

**EXAMPLE 4.44**

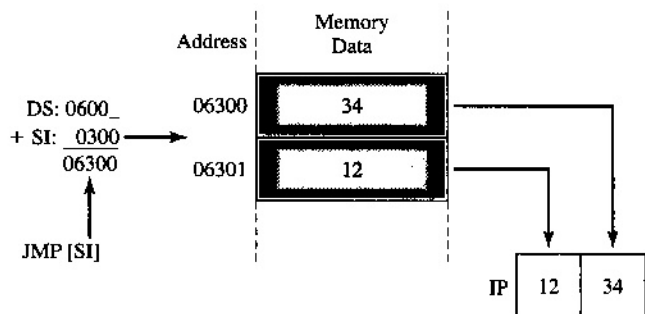
Let us examine the code for three different types of JMP instructions. A short jump JMP 120H is located at address 100H. The corresponding machine code is EB 1E. The EB part of the machine code means “short jump.” The 1E part is the relative offset of the target address. This offset is added to the current IP value, which is 102H, the first available location following JMP 120H. If we add 1EH to 102H, we get 120H, the desired target address to jump to. Now let us assume that a second short jump JMP 0C0H is located at address 102H. The machine code for this instruction is EB BC. Because the target address is smaller than the current instruction pointer address of 104H, the relative offset will be negative. The offset value BC is a negative signed number equal to the difference between C0H and 104H. In both of these instructions, the current IP value was *not* the address of the JMP instruction itself, but the address of the *next* instruction. Because JMP 120H is located at 100H and only required 2 bytes, the next instruction begins at 102H.

Now consider a near jump JMP 3000H located at address 200H. The assembler knows that it should use a near jump because the difference between the 3000H target and the current IP was greater than 127. The machine code for JMP 3000H is E9 00 30. The E9 part indicates a near jump and 00 30 is the byte-swapped target address.

A far jump JMP FAR PTR XYZ requires 5 bytes of code. The first byte is EA, which indicates a far jump. The next 2 bytes are the new IP address, and the last 2 bytes are the new CS value. Where does EA 00 10 00 20 take the processor? The new IP address from this machine code is 1000H. The new CS address is 2000H. The processor will perform a far jump to address 21000H.

Other legal forms of JMP involve using registers or the contents of memory locations to supply the target. For example, JMP AX indicates a near jump to the address contained in register AX. If AX contains 4500H, then the processor will jump to address 4500H within the code segment. JMP [SI] causes the processor to read the word stored in the memory locations pointed to by the SI register, and use that word as the target address. Figure 4.20 shows this instruction in more detail. The [SI] operand indicates an indirect jump and requires that the memory word pointed to by SI be read and placed into the IP register. In this fashion, JMP [SI] sets the processor up to fetch its next instruction from address 1234H within the current code segment.

**FIGURE 4.20** Operation of an indirect JMP



The machine code for JMP AX is FF E0. The machine code for JMP [SI] is FF 24. ■

**Conditional Jumps.** The 80x86 has a large variety of conditional jumps, all of which depend on specific flag states to enable their operation. Many of these conditional jumps test only one specific flag. For example, the JC and JNC instructions examine only the state of the carry flag to determine their next step. Others, such as JA and JNA, perform a Boolean test of more than one flag. Furthermore, none of the flags are affected by the jumps. As we

have previously seen, a number of instructions go by more than one name. Such is the case for the unconditional jumps. The instructions, their required flag conditions, and their interpretations are as follows:

| <i>Instruction</i> | <i>Flag Tested</i>      | <i>Explanation</i>           |
|--------------------|-------------------------|------------------------------|
| JNC                | CF = 0                  | jump if no carry             |
| JAE                | "                       | jump if above or equal       |
| JNB                | "                       | jump if not below            |
| JC                 | CF = 1                  | jump if carry                |
| JB                 | "                       | jump if below                |
| JNAE               | "                       | jump if not above or equal   |
| JNZ                | ZF = 0                  | jump if not zero             |
| JNE                | ZF = 0                  | jump if not equal            |
| JZ                 | ZF = 1                  | jump if zero                 |
| JE                 | "                       | jump if equal                |
| JNS                | SF = 0                  | jump if no sign              |
| JS                 | SF = 1                  | jump if sign                 |
| JNO                | OF = 0                  | jump if no overflow          |
| JO                 | OF = 1                  | jump if overflow             |
| JNP                | PF = 0                  | jump if no parity            |
| JPO                | "                       | jump if parity odd           |
| JP                 | PF = 1                  | jump if parity               |
| JPE                | "                       | jump if parity even          |
| JA                 | (CF or ZF) = 0          | jump if above                |
| JBNE               | "                       | jump if not below or equal   |
| JBE                | (CF or ZF) = 1          | jump if below or equal       |
| JNA                | "                       | jump if not above            |
| JGE                | (SF xor OF) = 0         | jump if greater or equal     |
| JNL                | "                       | jump if not less             |
| JL                 | (SF xor OF) = 1         | jump if less                 |
| JNGE               | "                       | jump if not greater or equal |
| JG                 | ((SF xor OF) or ZF) = 0 | jump if greater              |
| JNLE               | "                       | jump if not less or equal    |
| JLE                | ((SF xor OF) or ZF) = 1 | jump if less or equal        |
| JNG                | "                       | jump if not greater          |

Note that you must supply the target address in the jump instruction. Unlike JMP, we are not allowed to use register names or other fancy addressing mode operands. The nice thing about relative jumps is that they always jump to the correct location within the code segment, no matter where the code segment appears in memory.

#### ■ EXAMPLE 4.45 Let us look at a few ways conditional instructions may be used to test information.

Consider the following group of instructions, which determines if the AL register contains a lowercase ASCII letter.

```

CMP    AL, 'a'
JB     NOTLOWER
CMP    AL, 'z'+1
JNB    NOTLOWER

```

If AL does not contain the ASCII code of a lowercase letter (61H to 7AH for "a" or "z"), the processor will jump to NOTLOWER. If AL does contain a valid code, then neither of the conditional jumps will be taken, and the processor will execute the first instruction

following `JNB NOTLOWER`. Remember that `JB` and `JC` are equivalent instructions, as are `JNB` and `JNC`. Thus, the same lowercase test can be performed like this:

```
CMP AL, 'a'
JC  NOTLOWER
CMP AL, 'z'+1
JNC NOTLOWER
```

Next, suppose that you are working with a portion of code that adds small increments to a register, as in:

```
ADD BL, 8
```

If `BL` contains a value in the range 00 to 77H, the result of the `ADD` will be in the range 08 to 7FH. If, instead, `BL` contained a value in the range 78H to 7FH, the result after adding eight would be 80H to 87H. In terms of signed numbers, `BL` has gone through a sign change. Why has this happened? The value 78H corresponds to 120 decimal. Adding eight to this gives us 128, which is represented by 80H, or 10000000 binary. Note that the MSB is high, which indicates the *negative* value  $-128$  from the 2's complement point of view. If we simply interpret 80H as an *unsigned* integer, the 80H means  $+128$ . This unintentional sign change will be indicated by the processor's **overflow flag**. A similar result is obtained when

```
SUB BL, 8
```

is used, with the overflow flag being set when the 80H barrier is passed. To test for this condition, use the `JO` and the `JNO` instruction, as in:

```
ADD BL, 8
JO  OVERFLOW
```

Note that the overflow flag tells us something different than the carry flag does. There is no carry when `BL` goes from 78H to 80H, but there is an overflow.

Many conditional jumps test more than one flag. For instance, `JG` uses three flags (sign, overflow, and zero) to determine its outcome. Examine the following three groups of code. Which `JG` instructions are taken?

|                          |                          |                          |
|--------------------------|--------------------------|--------------------------|
| <code>MOV AL, 30H</code> | <code>MOV AL, 30H</code> | <code>MOV AL, 30H</code> |
| <code>CMP AL, 20H</code> | <code>CMP AL, 40H</code> | <code>CMP AL, 90H</code> |
| <code>JG XXX</code>      | <code>JG YYY</code>      | <code>JG ZZZ</code>      |

The `JG` to `XXX` is taken because 30H is greater than 20H. The `JG` to `YYY` is not taken because 30H is not greater than 40H. The `JG` to `ZZZ` is taken because the 90H value is interpreted by the processor as the negative number  $-112$  (due to its MSB being high). Thus, the `CMP` instruction determines that 30H is greater than  $-112$  (90H), and so the `JG` is taken. Note that if you want 90H to be interpreted as the *unsigned* value 144, a different conditional jump (such as `JC` or `JNC`) should be used. ■

#### ■ EXAMPLE 4.46

Is it possible to perform a conditional jump that has a target address outside of the relative range of  $+127/-128$  bytes?

**Solution:** Yes. Two instructions are needed to synthesize a conditional jump of this type. Consider the following code:

```
                JNZ  SKIPJMP
                JMP  NEWPLACE
SKIPJMP:      ---
```

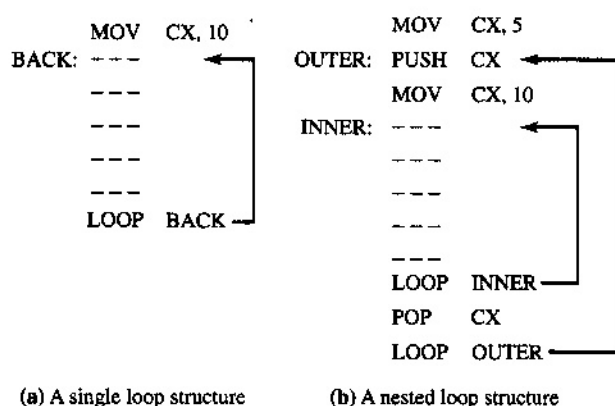
Here, the conditional jump JNZ is used to jump over a near JMP to NEWPLACE whenever the zero flag is cleared. When the zero flag is set, the JNZ will *not* jump to SKIPJMP but will simply continue execution with the next instruction in memory, which is the JMP NEWPLACE instruction. So, the zero flag must be set to perform a near JMP to NEWPLACE. ■

**LOOP Short-Label (Loop).** This instruction will decrement register CX and perform a short jump to the target address if CX does not equal 0. LOOP is very useful for routines that need to repeat a specific number of times. CX must be loaded prior to entering the section of code terminated by LOOP.

#### ■ EXAMPLE 4.47

In Figure 4.21(a) a LOOP instruction is used to execute a short section of code. The number placed into CX at the beginning of the loop (10 decimal in this case) determines how many times the loop repeats. Notice that the LOOP BACK instruction does not contain a reference to CX. The processor knows it should automatically decrement CX while executing this instruction. LOOP BACK actually performs a short jump to BACK as long as CX does not equal 0.

**FIGURE 4.21** Using LOOP instructions



In some cases it becomes necessary to perform a “loop within a loop,” or *nested* loop as it is more commonly known. Figure 4.21(b) shows an example of a nested loop. The outer loop executes five times and the inner loop ten times, and instructions in the body of the loop execute fifty times. The PUSH and POP instructions are used to preserve the contents of the outer loop counter.

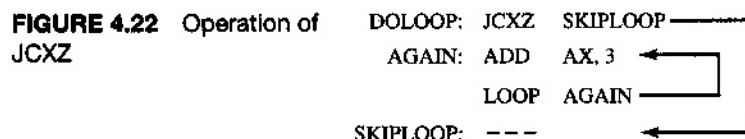
A single LOOP can be repeated up to 65,535 times using CX as the counter. How many loops are possible with a nested loop? The answer is over 4 billion! ■

**LOOPE/LOOPZ Short-Label (Loop if Equal, Loop if Zero).** This instruction is similar to LOOP except for a secondary condition that must be met for the jump to take place. In addition to decrementing CX, LOOPZ also examines the state of the zero flag. If the zero flag is set and CX does not equal 0, LOOPZ will jump to the target. If CX equals 0, or if the zero flag gets cleared within the loop, the loop will terminate. As always, when a conditional instruction does not have the correct flag condition, execution continues with the next instruction. LOOPE is an alternate name for this instruction.

**LOOPNE/LOOPNZ** *Short-Label (Loop if not Equal, Loop if not Zero).* LOOPNZ is the opposite of LOOPZ. The zero flag must be *cleared* to allow further looping. LOOPNE has the same function.

**JCXZ/JECXZ** *Short-Label (Jump if CX/ECX = 0).* JCXZ jumps to the target address if register CX contains 0. JECXZ does the same if ECX equals 0. CX and ECX are not adjusted in any way. There are times when JCXZ or JECXZ should be used at the beginning of a section of LOOP code to skip over the code when CX or ECX equals 0.

■ **EXAMPLE 4.48** In Figure 4.22 a JCXZ instruction is used to skip over the AGAIN loop whenever CX equals 0. If the JCXZ instruction were absent, how many times would the loop execute?



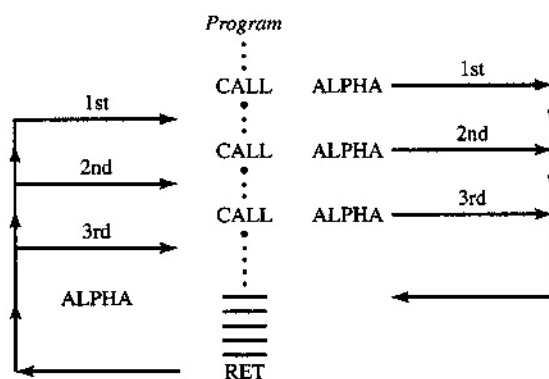
**Solution:** If JCXZ is removed and the loop is entered with CX equal to 0, it will repeat 65,536 times! This is because LOOP decrements CX before it tests for 0. If CX is decremented from 0 it becomes FFFFH, which means 65,535 more decrements are needed to get back to 0. Note that the code (JCXZ included or not) works properly for every other starting value of CX. ■

## Subroutine and Interrupt Instructions

Although subroutines and interrupts are called or generated in different ways, they share some common ground. Both require the use of the stack for proper operation. Both have a means of returning to where they came from (via information returned from the stack).

A **subroutine** is a collection of instructions that is **CALLed** from one or many other different locations within a program. At the end of a subroutine is an instruction that tells the processor how to **RETurn** (go back) to where it was called from. Figure 4.23 shows how a subroutine called ALPHA can be called from many different places within a program. The RET instruction at the end of ALPHA always causes the processor to go to the instruction immediately following the last CALL. It does this by popping a return address off the stack, which was placed there by the corresponding CALL. For example, when the first

**FIGURE 4.23** Calling a subroutine from many different places



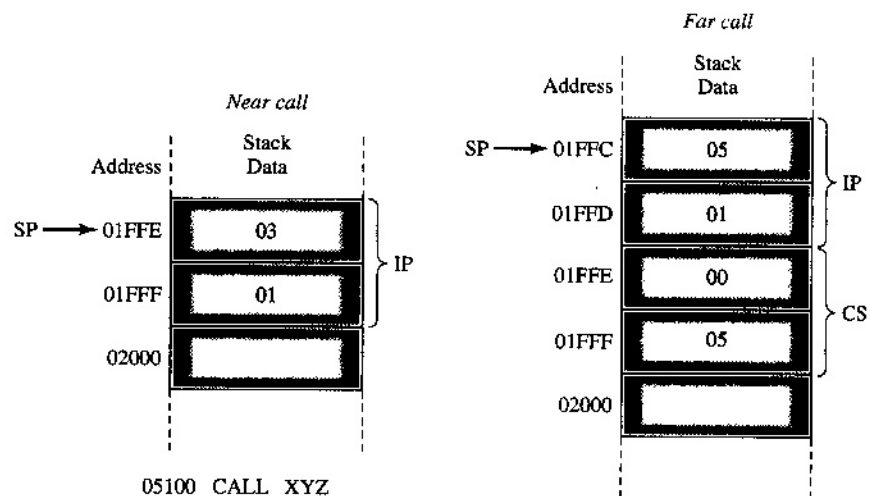
**CALL** is encountered, the address of the instruction immediately following the **CALL** is pushed onto the stack. When the **RET** instruction at the end of **ALPHA** executes, the return address is popped off the stack and placed into the **IP** register. Using the stack in this way guarantees that **RET** will go back to the right place no matter which of the three **CALL** instructions has been used.

Interrupts operate in a similar manner, but require more use of the stack. In addition to a return address, the 80x86 interrupt mechanism also pushes the processor flag register onto the stack. An interrupt is generated through software by the use of the **INT** or **INTO** instructions, which are to interrupts what **CALL** is to subroutines. An **interrupt service routine**, or **ISR** for short, is a common way of referring to the code used to handle an interrupt. Because the stack is used differently, **RET** is not used to return from an **ISR**. **IRET** is used instead to ensure that the stack is popped correctly. The processor has many different types of interrupts, which we will cover in detail in Chapter 5.

**CALL Procedure-Name (Call Procedure).** This instruction is used to call a subroutine. The subroutine may be located in the current code segment or in a different one. Like the intrasegment and intersegment jumps, we also have intrasegment and intersegment **CALL**s. We think of these as calls to near or far **procedures**. A procedure is another way of referring to a subroutine and has to do with the assembler directives **PROC** and **ENDP** used in the source file. The direct and indirect types of **JMP**s that we have previously seen also apply to **CALL**. In a direct **CALL**, the address of the procedure is specified in the instruction (usually via the label naming the procedure). In an indirect **CALL**, the procedure address may be contained in a processor register or memory location. **CALL** pushes the address of the instruction immediately following itself onto the stack. In the case of a near **CALL**, this address is just one word of data. In the case of a far **CALL**, two pushes are performed: one for the instruction offset and the other for the code segment address. **CALL** then jumps to the address of the procedure and resumes execution.

■ **EXAMPLE 4.49** Figure 4.24 shows the contents of the stack after near and far **CALL**s to the same procedure. For the near call, only the offset of the next instruction is pushed. For the far call, the instruction offset and the code segment value are pushed. If both instructions were located at offset 100H within a code segment located at 5000H (e.g., **CS** equals 0500H), the stack

**FIGURE 4.24** Stack contents during **CALL**





would be filled with the data you see in the figure. The near call to XYZ will cause only the return address 103H to be pushed. Because the near call requires 3 bytes of machine code, address 103H is the first available instruction address.

The far call is a 5-byte instruction and needs to push more information onto the stack. First the contents of the CS register are pushed (0500H). Then the instruction offset is pushed. This offset is 105H now because of the length of the far call instruction. The procedure must not modify the contents of these stack locations. It is also important to guarantee that the SP register points to the correct address before executing RET. It is okay to push items onto the stack while inside the procedure, but these items must all be popped off before returning. ■

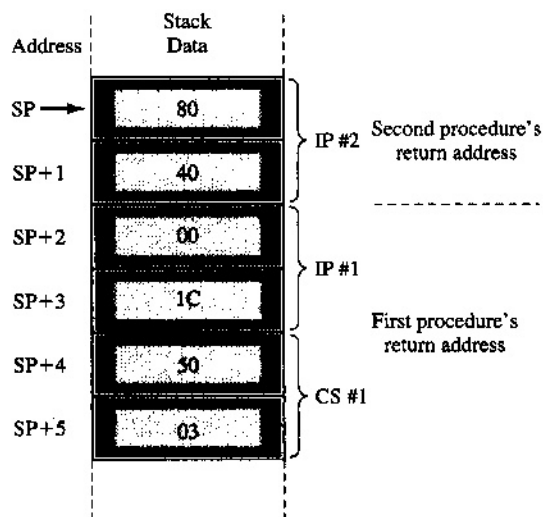
**RET Optional-Pop-Value (Return from Procedure).** This instruction is used to provide an orderly exit from a procedure that has been called. It is important to understand that RET does not know *anything* about the procedure it terminates except for its type being far or near. All RET does is pop the appropriate information off the stack and place it into the assigned registers. For a near procedure, RET pops one word off the stack and writes it into the IP register. For a far procedure, RET pops two words. The first goes into the IP register, the second into the CS register. Once these pops are made (for either type of return), the optional-pop-value is used to pop additional words off the stack, if any at all. The optional-pop-value is added to the stack pointer, advancing it past information that was pushed before the procedure was called. This value is assumed to be 0 when no pop-value is included.

#### ■ EXAMPLE 4.50

The return addresses of two different subroutines are contained in the stack. The subroutine called first was a far procedure. The instruction that called it pushed the return address 0350:1C00, where 0350H is the segment address and 1C00H is the offset. The second subroutine called was a near procedure with a return address of 4080H. The second procedure was called from inside the first procedure. What are the top three items on the stack?

**Solution:** Figure 4.25 shows the stack created by the return addresses for each procedure. When the second procedure finishes, its near RET will pop 4080H off the stack and

**FIGURE 4.25** Nested return addresses



put it into the IP register. This gets us back into the first procedure (which called the second). The far RET in this procedure pops 1C00H off the stack, puts it into the IP register, and then pops 0350H off the stack and places it into the CS register. Any number of procedure return addresses can be nested in this way, as long as the correct RETs are always used and you do not run out of stack memory.

The machine code for a near RET is C3. The machine code for a far RET is CB. When pop-values are used, 2 additional bytes are added to the instructions. They contain the pop-value. The machine code for a near RET 2 is C2 02 00. ■

---

**INT Interrupt-Type (Interrupt).** The 80x86 supports 256 software interrupts, which initially operate in a manner similar to CALL. When INT is encountered, the processor will first push the flags onto the stack. This is an important step, because the flags tell us part of the processor's internal state at the time it was interrupted. Then it clears two special flags called the **trace flag** and the **interrupt-enable flag**. It does this to prevent additional software interrupts from occurring while it processes the first.

Next, the processor pushes the current CS register onto the stack and follows that with the contents of the IP register. The last two pushes resemble the operation of a far call. The *number* of the interrupt, rather than its address, is coded into the instruction and is now used to find the correct place in the processor's **interrupt vector table**, a 1KB block of memory located in the beginning of the processor's address space. Its range of address is from 00000 to 003FFH. The interrupt number is called the **interrupt-type** and is multiplied by 4 to get the address within the vector table that contains the interrupt service routine address. All ISR addresses are 4 bytes long and in the standard CS-IP format. Once the vector address is known, the processor reads the ISR address out of the table, places the information into the CS and IP registers, and resumes program execution at the new location.

---

■ **EXAMPLE 4.51** What is the sequence of events for INT 08H if it generates a CS:IP return address of 0100:0200? The flag register contains 0081H.

**Solution:** The flags are pushed first. The return CS and return IP addresses are pushed next. Multiplying interrupt type 8 by 4 gives 32, which is 00020H. The ISR address stored at this vector address (see Figure 4.26) is read into CS:IP and execution continues at address 05810H. ■

---

**IRET/IRETD (Interrupt Return).** Because the operation of an interrupt requires different handling of the stack, an IRET instruction must be used at the end of an ISR. A normal RET instruction will not pop the flags off and will most likely result in incorrect program execution. IRET pops the CS and IP return addresses and then restores the flag register by popping its value too. Thus, the flags will resemble their state at the exact moment of the interrupt. When 32-bit addressing is in effect, IRETD is used to pop 32-bit addresses off the stack.

---

■ **EXAMPLE 4.52** What is the return address and the contents of the flag register when IRET uses the stack information from Figure 4.27?

**Solution:** The word 4500H is popped into the IP register. Then, 8000H is popped into the CS register. Together, these values indicate a return address of 84500H. Next, 03C4H is popped into the flags. The new value of the stack pointer is 04006H.

The machine code for IRET is CF. ■

---

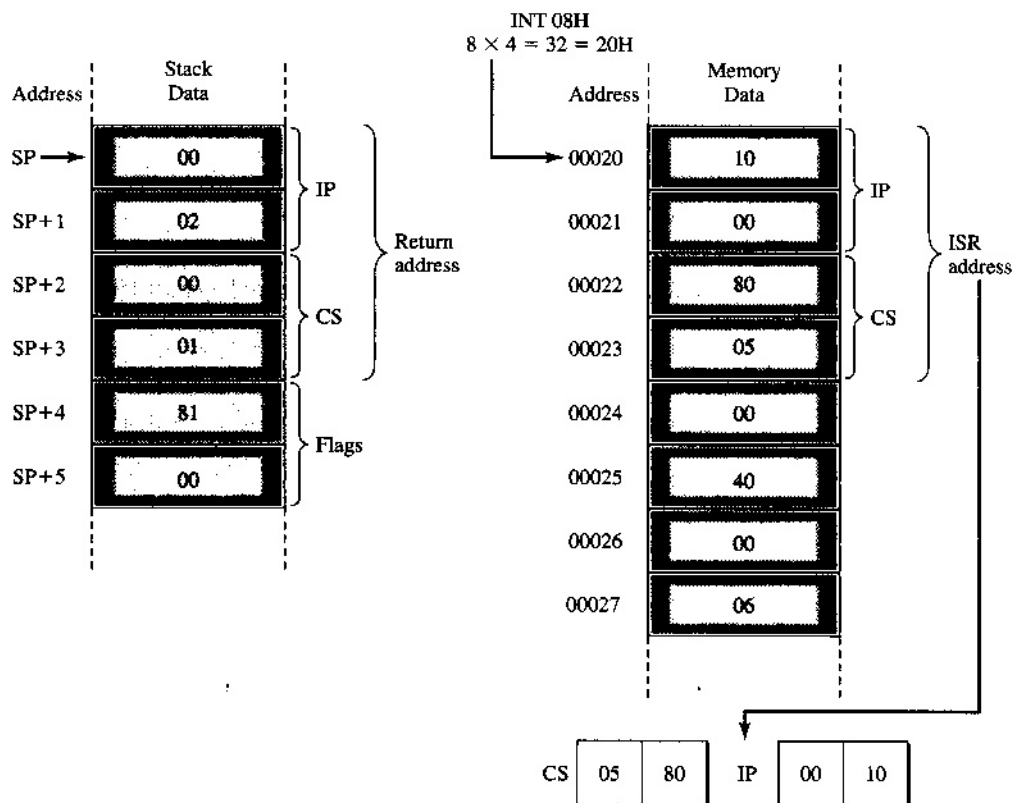
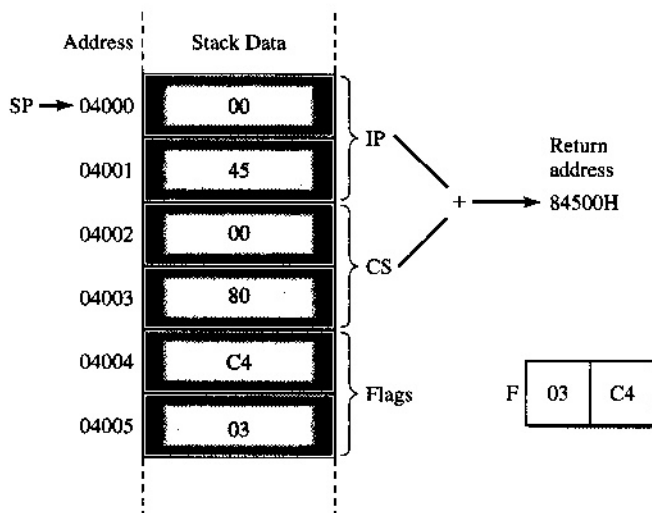


FIGURE 4.26 Operation of INT 08H

FIGURE 4.27 Operation of IRET



**INTO (Interrupt on Overflow).** If the overflow flag is set when INTO is encountered, it will generate a type-4 interrupt, with all of the pushes and operations performed by INT. The interrupt vector address of INTO is 00010H. If the overflow flag is cleared when INTO executes, no interrupt is generated and execution resumes with the next instruction. This interrupt is useful for determining when the results of an arithmetic operation have gone out of bounds.

**BOUND** *Index, Range (Check Array Index Against Bounds)*. This instruction tests the value of an index register against a predefined allowed range of indexes. If the array index is within range, no further action is taken. If the array index is out of range, an INT 05H is generated. The index operand may be any 16- or 32-bit register. The range operand specifies the address of two 16- or 32-bit numbers that indicate the allowed index range.

■ **EXAMPLE 4.53** The following excerpt from a source file shows how the BOUND instruction may be used. The index contained in the SI register is checked against the range 0 to 9.

```

        .DATA
RANGE   DW      0,9
ARRAY   DB      10 DUP(?)

        .CODE
        .
        .
        BOUND    SI, RANGE
        MOV      AL, [SI]
        .
        .

```

If the SI register has an index value between 0 and 9, the MOV instruction will be executed. If not, an INT 05H is generated. Note that MS-DOS uses INT 05H to print the contents of the screen, so using BOUND in real mode will require you to change the operation of INT 05H's ISR. ■

**ENTER/LEAVE** (*Enter/Leave a Procedure*). These instructions are used to manage the **stack frame** used by high-level languages such as C. A stack frame stores pointers and variables used by the current function or procedure being executed in the high-level language. By convention, the BP register is used as a **frame pointer**, and indicates the beginning of the previous stack frame. The first thing the ENTER instruction does is to push the BP onto the stack. Thus, the current stack frame now points back to the previous stack frame. This value will be popped by the LEAVE instruction before the function RETURNS, restoring the old frame pointer.

Next, the new value of the SP is copied into BP. This sets the frame pointer to the beginning of the current stack frame.

Finally, the amount of dynamic storage space to reserve, in bytes, is subtracted from the SP. Dynamic storage space is used by the procedure for storage of local variables.

The ENTER instruction takes two operands. The first operand specifies the amount of dynamic storage space in bytes. This value must be a multiple of 2. The second operand indicates the lexical nesting level of the new procedure being entered. The lexical nesting level (or simply level) is determined by how many subprocedures are currently active. For instance, if the main program calls Procedure A, A's level is 1. If Procedure A then calls Procedure B, Procedure B's level is 2. When the level is 0, ENTER is equivalent to:

```

PUSH    BP
MOV      BP, SP
SUB      SP, dsp

```

where dsp specifies the amount of dynamic storage space.

When the specified level is not 0, ENTER also pushes pointers onto the stack that references the stack frames of all previous procedures.

The LEAVE instruction restores the original (previous) values of the BP and SP registers. Equivalent instructions are:

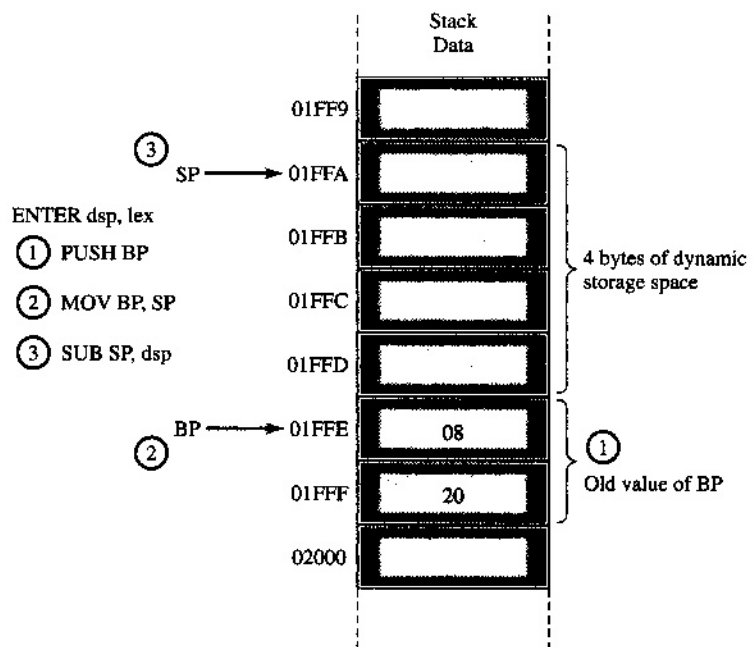
```
MOV     SP, BP
POP     BP
```

We will examine how ENTER and LEAVE are used in the C programming language in Chapter 7.

■ **EXAMPLE 4.54** The BP and SP registers contain the offsets 2008H and 2000H, respectively. The stack segment is at 0000H. What is the result of executing ENTER 4,0?

**Solution:** Figure 4.28 shows the changes made to the stack when ENTER 4,0 executes. First, the BP is pushed, which decrements the SP to 1FFEH. This value is then copied into the BP register. Last, 4 is subtracted from the SP, giving it a final value of 1FFAH.

**FIGURE 4.28** Operation of ENTER 4,0



The corresponding LEAVE instruction will restore the value of the BP to 2008H and the SP to 2000H.

The machine code for ENTER 4,0 is C8 0004 00. The machine code for LEAVE is C9. ■

## Assembler Directives PROC and ENDP

As mentioned previously, a subroutine or procedure must be of a specific type. The two types, near and far, are specified by the PROC statement that begins the subroutine. In the



**STD** (*Set Direction Flag*). The direction flag is set by this instruction. Its machine code is FD.

**CLI** (*Clear Interrupt-Enable Flag*). This instruction clears the interrupt-enable flag. This prevents the processor from responding to a hardware INTR request. No other interrupts are affected. Its machine code is FA.

**STI** (*Set Interrupt-Enable Flag*). Setting the interrupt-enable flag allows the processor to acknowledge an interrupt signal on the INTR input. Its machine code is FB. If a request was made to INTR while the interrupt-enable flag was cleared (which we call a *pending* interrupt), that interrupt request would be acknowledged after completion of the next instruction after STI.

**HLT** (*Halt Until Interrupt or Reset*). This instruction terminates program execution. Its machine code is F4. It places the processor in a halt state, in which it will remain until it receives a hardware reset, or an interrupt request on NMI or INTR. If the request is on INTR, interrupts must be enabled to leave the halt state.

**NOP** (*No Operation*). Although this instruction does not do anything at all (no flags or registers are affected), it is still quite useful in many ways. Specifically, one application of NOP is its use within timing delay loops. There are applications that require specific time delays to be generated (for slowing down a graphics display to see the action, for example). Sometimes it is useful to throw a couple of NOPs into the timing loop, to increase the loop time. After all, it takes a few clock cycles to fetch, decode, and execute NOP. It is nice to be able to use an instruction that takes up a little time but does nothing. Its machine code is 90.

**LOCK** (*Lock Bus During Next Instruction*). Executing LOCK causes the processor's  $\overline{LOCK}$  output to go low for the duration of the next instruction. Thus, LOCK is used as a prefix to another instruction. Locking the bus is a way of giving the processor some privacy while it executes the subsequent instruction, which cannot be interrupted until it completes. The prefix code for LOCK is F0.

We will examine other processor control instructions in Chapter 14.

---

## 4.7 HOW AN ASSEMBLER GENERATES MACHINE CODE

During assembly, source statements are read one line at a time and examined. If they contain legal instructions, the assembler can determine the required machine code by filling in missing bits in a basic opcode format. For example, the instructions MOVSB and MOVSW both have the same basic opcode format: 1010010 $w$ , where  $w$  is a variable that takes on 0 for a byte operation and 1 for a word operation. Thus, if the assembler reads a source statement that contains MOVSB, once it recognizes the MOVS part of the instruction, it will then look for a "B" or a "W" to determine the correct value of  $w$ . It is easy to see that MOVSB has A4 as its machine code and that MOVSW has A5 as its machine code. If the assembler does not see a "B" or a "W" after MOVS, it will generate an error message.

Other instructions are more complicated to assemble than MOVS. One form of the multiplication instruction MUL can have two sizes of operands (byte and word), and the operands can be registers or memory locations. Clearly, the basic opcode format for MUL must be more complicated. In fact, the format is 1111011 $w$  mod 100  $r/m$ . This strange-looking string of symbols is used to specify all possible opcodes for MUL.

Consider the instruction MUL CX. The  $w$  bit works as it did in the MOVS example. Because CX is a word-size register,  $w$  will be set to 1.  $Mod$  is a 2-bit variable that is set

to 11 by the assembler when the operand is a processor register. The register specified in the instruction is indicated by the 3-bit variable *r/m*. Register CX has the code 001 associated with it. So, `MUL CX` becomes 11110111 11100001, or F7 E1 in machine code. `MUL SI` would be F7 E6, because SI's internal code is 110.

Suppose an instruction has two operand fields. In `XCHG AL,BL`, the operands are both byte-size registers. Each register will have its own internal code. The destination register AL is coded as 000. The source register BL is coded as 011. The basic opcode format for `XCHG` is 1000011 *w* mod reg *r/m*. The new variable *reg* represents the 3-bit code for the destination operand. *R/m* indicates the source operand. When exchanging registers, *w* will be set to 11. Inserting these 2 bits and the 6 bits that are used to specify AL and BL, we get 10000110 11000011. This is the machine code 86 C3. Similarly, `XCHG CX,DX` has 87 CA for its machine code. Can you determine the 3-bit codes associated with CX and DX? (You should get 001 and 010, respectively.)

So, we see that the process of assembling a program involves a great deal of time picking the right combination of bits to place into opcode patterns. Let us keep this in mind when we assemble our own programs later in the book. When the assembler indicates an error, it is not because it picked the wrong bits; it is because the information supplied was incorrect, missing, or of the wrong type.

## 4.8 THE BEAUTY OF RELOCATABLE CODE

In the early days of microprocessors, few addressing modes were available. The limited ability of the early microprocessor to access its own memory required, in most cases, the generation of specific *absolute* addresses within instructions. For example, if an instruction located at address 1000H needed to examine the data in memory location 106FH, the address 106FH would have to be coded in the instruction as a 2-byte absolute address. The relative mode of addressing gets rid of the absolute address portion of the instruction and replaces it with a single byte of relative address information. Now only the offset 6FH must be saved within the instruction's machine code. You may think that saving a single byte here and there will make little difference. In short programs this is true. But in longer programs, which are more likely to access temporary values or data tables stored in memory or perform a significant number of jumps, a substantial savings in space results.

A second, more important feature of relative addressing is its built-in **relocatability**. A relocatable program is a program that can be loaded into memory at any address and then executed normally. Early microprocessors did not generate relocatable code and thus required their programs to load into memory at a specific location. For instance, a program ORGed at address 3C00H had to load into memory at address 3C00H in order to run properly. The same program loaded at 5000H, or anywhere else, would contain absolute addresses (such as the address 106FH with the example instruction at the beginning of this section) that did not look at the correct locations. For the program to run correctly, each absolute address would have to be adjusted depending on the new load address. For example, the 106FH address might be changed to 506FH. While this type of program modification is possible, programs will require additional load time when we try to execute them. Relocatable code does not have this disadvantage. An instruction that accesses the location "30 locations forward" of its own address will get the address right no matter where it is loaded. The 80x86's ability to use relocatable code is an attractive feature.



---

## 4.9 TROUBLESHOOTING TECHNIQUES

Now that we have seen the entire 80x86 instruction set, let us spend a moment on some tips designed to keep the software development process running smoothly.

- Periodically review the entire instruction set. Programmers tend to fall in love with certain instructions and methods of writing code, and it is easy to forget that the processor already contains an instruction to fill your need.
- Understanding the binary number system and its associated operations is just as important as knowing the addressing modes and instruction types. It is difficult to perform any kind of interfacing, design, or programming without a good background knowledge of 1s and 0s. Take the time to review Appendix B if you have not already done so; it may save you some effort in the future.
- The arithmetic, logical, and bit manipulation instructions all rely heavily on the flags. As we have seen again and again, the flags are very important. Get into the habit of checking out an instruction to see how it affects the flags. For example, you may write a short loop like this:

```
BACK:      DEC    BL
           ADD    AL, 7
           JNZ    BACK
```

Logically, everything seems in order. AL and BL are both changed in the loop. The problem is that the JNZ instruction is meant to test the NZ condition created by the DEC instruction, but the ADD instruction, which also updates the zero flag, changes the outcome. It is best to place the instruction that affects a flag as close as possible to the instruction that uses it.

- When using subroutines, or any group of instructions that manipulate the stack, it helps to go through the code to verify that the same number of pushes and pops are made. It is very easy to add an extra push or pop when writing stack-based code for the first time. Generally, the DOS environment on the personal computer is very unforgiving when its stack is abused. If the machine locks up or reboots when you run your stack-based program, it would be good to begin your troubleshooting by examining what the stack is doing.

It may be a good idea to keep a logbook of the problems you encounter working with assembly language. Until you have seen the same problem enough times, or learn how to avoid it, keeping track of the problem and its solution will save a lot of time and effort should you encounter it again.

---

## SUMMARY

In this chapter we examined the remainder of the 80x86 instruction set. The arithmetic, logical, bit manipulation, program transfer, and processor control instructions were all covered in detail. The basic operation of an assembler was covered, with examples given to show how bit fields within the instruction are filled in. The ability of relocatable code to execute at any load address was also discussed.

Use this chapter, and Chapter 3, for reference when you begin studying programming applications in Chapter 6.

## STUDY QUESTIONS

1. If AX contains 1234H, what is the result of ADD AL,AH?
2. What is the state of each flag after ADD AL,AH in the previous question?
3. Memory location 2000H has the word 5000H stored in it. What does each location contain after INC BYTE PTR [2000H]?
4. Repeat Question 3 for DEC WORD PTR [2000H].
5. What is the state of the zero flag after CMP CL,30H if CL does not contain 30H?
6. What instruction is needed to check whether the upper bytes of AX and BX are equal?
7. Show the instructions needed to multiply AX by 25. Assume the results are unsigned.
8. If DX contains 00EEH and AX contains 0980, what is the result of:

```
MOV  BX,0F0H
DIV  BX
```

9. Repeat Question 8 for the instruction IDIV BX.
10. What instruction is needed to find the signed 2's complement of CX?
11. What are the results of CBW if AL contains 30H? What if AL contains 98H?
12. What is the final value of AL in this series of instructions?

```
MOV  AL,27H
MOV  BL,37H
ADD  AL,BL
DAA
```

13. If DX contains 7C9AH, what is the result of NOT DX?
14. If AL contains 55H and BL contains AAH, what is the result of:
  - (a) AND AL,BL
  - (b) OR AL,BL
  - (c) XOR BL,AL
15. What does SHL AL,1 do if AL contains 35H?
16. What is the data in BX after SHR BX,CL if CL contains 6?
17. Memory location 1000H contains the byte 9F. What instructions are needed to rotate it 1 bit right?
18. Explain the difference between short and near jumps.
19. What conditional jump instruction should be used after CMP AL,30H to jump when AL equals 30H?
20. What instruction is needed in Question 19 to jump when AL is less than 30H?
21. The lower byte of the flag register contains 95H. Which of the following instructions will actually jump?
  - (a) JZ
  - (b) JNC
  - (c) JP
  - (d) JA
  - (e) JGE
  - (f) JLE
22. How many times does the NOP instruction execute in the following sequence?

```
      MOV  CX,20H
XYZ:  PUSH  CX
      MOV  CX,9
ABC:  NOP
      LOOP ABC
      POP  CX
      LOOP XYZ
```

23. A near CALL generates a return IP of 1036H. What are the contents of stack memory after the call if SP initially contains 2800H?

24. Repeat Question 23 for a far CALL located in code segment 0400H.
25. What are the final SP values in Questions 23 and 24?
26. Explain the operation of near and far RETURNS.
27. What are all of the activities following an INT 16H?
28. What does INTO do in this sequence of instructions?

```

MOV     AL, 30H
MOV     BL, 0E0H
ADD     AL, BL
INTO

```

29. What instructions are needed to add AL, BL, and DL together, and place the result in CL? Do not destroy BL or DL.
30. What is the difference between MOV AX,0 and SUB AX,AX? There may be more than one difference to comment on.
31. Multiply the contents of AX by 0.125. Because fractional multiplication is not available, you must think of an alternate way to solve this problem. Assume that AX is unsigned.
32. What does this sequence of instructions do?
 

```

      MUL     BL
      DIV     CL
      
```
33. What are the largest two decimal numbers that may be multiplied by MUL? Repeat for IMUL.
34. Find the volume of a cube whose length on one side has been placed into BL. The volume should be in DX when finished.
35. Write the appropriate AND instruction to preserve bits 0, 3–9, and 13 of register BX, and clear all others.
36. What OR instruction is needed to set bits 2, 3, and 5 of AL?
37. Show how ROL can be used to rotate a 32-bit register composed of AX and BX, with AX containing the upper 16 bits.
38. Why is it important for SAL to preserve the value of the MSB?
39. Why must a subroutine contain RET as the final instruction?
40. Modify the data summing example so that bytes are added together instead of words.
41. Write a subroutine to compute the area of a right triangle whose side lengths are stored in AL and BL. Return the result in AX.
42. Write a subroutine that will compute the factorial of the number contained in DL. Store the result in word location FACTOR.
43. A data byte at location STATUS controls the calling of four subroutines. If bit 7 is set, ROUT1 is called. If bit 5 is clear, ROUT2 is called. ROUT3 is called when bits 2 and 3 are high, and ROUT4 is called if bit 0 is clear and bit 1 is set. These conditions may all exist at one time, so prioritize the routines in this way: ROUT1, ROUT3, ROUT2, and ROUT4.
44. Write a routine to swap nibbles in AL. For example, if AL contains 3E, then it will contain E3 after execution.
45. Write a subroutine that will increment the packed-decimal value stored in COUNT and reset it to 00 each time it reaches 60.
46. Write a subroutine that sets the zero flag if AH is in the range 30 to 212, and clears the zero flag otherwise.
47. Show the instructions needed to count the number of 1s found in AL. For example, if AL contains 10110001, the number of 1s is 4.
48. Write a routine that determines if BH contains a *palindrome*. A palindrome (in binary) is a pattern of 0s and 1s that reads the same forward and backward. For example, 11000011 is a palindrome, and 11001000 is not.

49. What instructions are necessary to determine the largest number contained in BX, CX, and DX? The number should be placed in AX when found.
50. Repeat Question 49 for the smallest number in AX, BX, CX, and DX.
51. Show the instructions needed to solve this equation:

$$AX = (5BX + CX/3)/DX$$

52. Registers AX, BX, and CX contain the respective values 2000H, 1000H, and 3000H. What is the result of `CMPXCHG BX,CX`?
53. What does `IMUL BX,CX,12` do?
54. What is the result of dividing the decimal number 100,000,000 by 1024? Show how this can be done using `IDIV`.
55. What is the result of executing `CDQ` if EAX contains 8F005000H?
56. What does `XADD BX, CX` do if BX contains 000AH and CX contains 1000H?
57. Register ECX contains the value 07E98600H. What happens when these instructions are executed?
- (a) `BSF EBX,ECX`
  - (b) `BSR EDX,ECX`
58. The AX register contains the value ABCDH. What are the results of executing the following?
- (a) `BT AX,1`
  - (b) `BTC AX,5`
  - (c) `BFS AX,10`
  - (d) `BTR AX,14`
59. If the zero flag is set, what does `SETNZ CL` do?
60. If AX and BX contain 2468H and 1357H, respectively, what are the contents of each register after execution of the following?
- (a) `SHLD AX,BX,6`
  - (b) `SHRD BX,AX,6`
61. Because the `BOUND` instruction causes problems with `INT 05H` in the MS-DOS environment, can you devise an equivalent set of instructions that tests the bounds of an array index? If the index is out of bounds, jump to the label `BADINDEX`.
62. How is the stack changed when `ENTER 2,0` executes? The initial stack pointer is at offset 1E00H and BP contains 1E20H.

---

# CHAPTER 5

---

## Interrupt Processing

---

### OBJECTIVES

In this chapter you will learn about:

- The differences between hardware and software interrupts
- The differences between maskable and nonmaskable interrupts
- Interrupt processing procedures
- The vector address table
- Multiple interrupts and interrupt priorities
- Special function interrupts
- The general requirements of all interrupt handlers

### KEY TERMS

Context

Environment

Interrupt acknowledge cycle

Interrupt request

Interrupt vector table

Non maskable interrupt

---

## 5.1 INTRODUCTION

The 80x86 provides a very flexible method for recovering from what are known as *catastrophic* system faults. Through the same mechanism, external and internal interrupts may be handled and other events not normally associated with program execution may be taken care of. The method that does all of this for us is the 80x86 interrupt handler. In this chapter we will see that there are many kinds of interrupts. Some of these deal with issues that have always plagued programmers (such as the divide-by-zero operation), while still others may be defined by the programmer. The emphasis in this chapter is on the definition of the numerous interrupts available. Actual programming examples designed to handle interrupts will be covered in the next chapter.

Section 5.2 explains the differences between hardware and software interrupts. Section 5.3 gives details on the processor's interrupt vector table. The entire process of how an interrupt is handled is presented in Section 5.4, followed by a description of multiple interrupts in Section 5.5. Section 5.6 explains the special interrupts incorporated into the 80x86, such as

the divide-by-zero interrupt. Examples of actual interrupt service routines in Section 5.7 are presented next, with troubleshooting techniques completing the chapter in Section 5.8.

## 5.2 HARDWARE AND SOFTWARE INTERRUPTS

An interrupt is an event that causes the processor to stop its current program execution and perform a specific task to service the interrupt. We are all interrupted many times during the day. A ringing telephone or doorbell, a knock on the door, or a question from a friend all indicate the need to communicate with you. The situation is much the same with the microprocessor. The interrupt is used to get the processor's attention. Interrupts may be used to inform the processor in an alarm system that a fire has started or a window has been opened. In a personal computer, interrupts are used to keep accurate time, read the keyboard, operate the disk drives, and access the power of the disk operating system.

Two kinds of interrupts are available: hardware and software interrupts. Hardware interrupts are generated by changing the logic levels on either of the processor's hardware interrupt inputs. These inputs are **NMI (nonmaskable interrupt)** and **INTR (interrupt request)**. INTR can be enabled and disabled by the state of the interrupt-enable flag (IF). The CLI (clear interrupt-enable flag) instruction clears IF, which disables INTR. The STI (set interrupt-enable flag) instruction sets IF, allowing INTR to respond to requests. This means that INTR can be *masked* (disabled). NMI gets its name from the fact that its operation cannot be disabled. NMI *always* causes an interrupt sequence when it is activated. Chapter 8 shows that a rising edge is needed on NMI to trigger the interrupt mechanism, and that INTR is level sensitive, requiring a high logic level to interrupt the processor.

Software interrupts are generated directly by an executing program. These types of interrupts are also called *exceptions*. Some instructions, like INT and INTO, initiate interrupt processing when they are executed. Other instructions are capable of generating an interrupt when a certain condition is met. DIV and IDIV, for example, cause a type-0 interrupt (divide error) whenever division by 0 is attempted.

What happens when a software and hardware interrupt occur at the same time? The processor has a technique for handling this situation; it requires that the interrupts be *prioritized*. Table 5.1 shows the interrupt priority scheme used by the 80x86.

Interrupts with the highest priority are divide-error, INT, and INTO. NMI and INTR have lower priorities, with single-step having the lowest. If both hardware interrupts are activated simultaneously, NMI will be serviced first, with INTR *pending* until it gets its chance to be recognized by the processor. If a divide-error and NMI occur simultaneously, divide-error will be recognized first, followed by NMI.

In the next section we will examine the details of the interrupt vector table, which is accessed when any type of interrupt is initiated.

**TABLE 5.1** Interrupt priorities

| <i>Interrupt</i> | <i>Priority</i> |
|------------------|-----------------|
| Divide-error     | Highest         |
| INT, INTO        |                 |
| NMI              |                 |
| INTR             |                 |
| Single-step      | Lowest          |

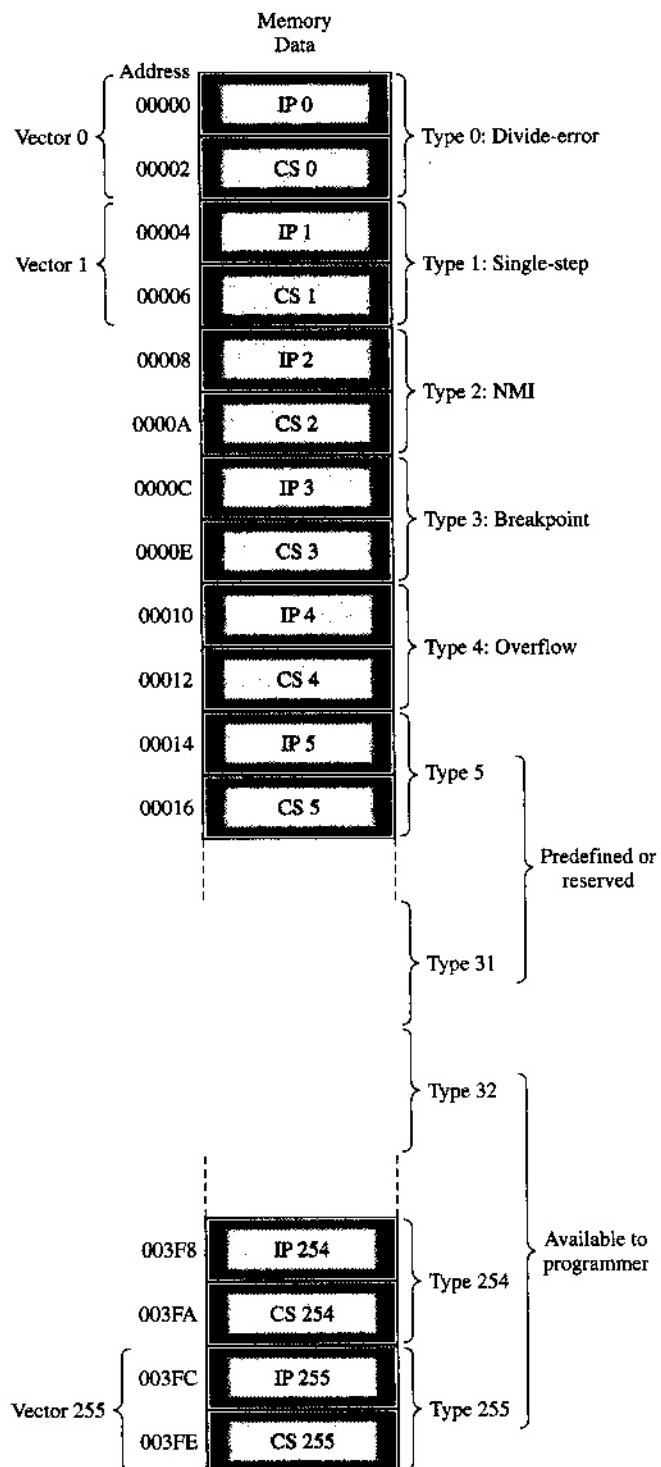
## 5.3 THE INTERRUPT VECTOR TABLE

All types of interrupts, whether hardware or software generated, point to a single entry in the processor's **interrupt vector table**. This table is a collection of 4-byte addresses (two for CS and two for IP) that indicate where the processor should jump to execute the associated interrupt service routine, the code designed to handle the specific interrupt. Because 256 interrupts are available, the interrupt vector table is 1,024 bytes long. The 1KB block of memory reserved for the table is located in the address range 00000 to 003FFH. Some earlier processors automatically loaded their first instruction from address 0000 after a reset. The 80x86 fetches its first address from location FFFF0H. This indicates that we are able to begin program execution without first having to place values into the interrupt vector table. If we plan on using any interrupts, it will be necessary to initialize the required vectors within the table. This can be done easily with a few MOV instructions.

Figure 5.1 shows the organization of the interrupt vector table. Each 4-byte entry consists of a 2-byte IP register value followed by a 2-byte CS register value. This indicates that interrupt service routines are considered far routines. Notice that some of the vectors are predefined. Vector 0 (the same as type-0) has been chosen to handle division-by-zero errors. Vector 1 (type-1) helps to implement single-step operation. Vector 2 is used when NMI is activated. Vector 3 (breakpoint) is normally used when troubleshooting a new program. Vector 4 is associated with the INTO instruction. Vector 5 is used when the BOUND instruction reports an out-of-range index. Vectors 6 through 18 perform various housekeeping duties. Some of these vectors (10, 11, 14) are operational only in protected or virtual-8086 mode. Table 5.2 shows the associated vector assignments. Vectors 19 through 31 are reserved by Intel for use in their products. This does not mean that these interrupt vectors are unavailable to us, but that we should refrain from using them in an Intel machine unless we know how they have been assigned.

**TABLE 5.2** Interrupt/exception vectors

| <i>Vector</i> | <i>Description</i>                |
|---------------|-----------------------------------|
| 0             | Divide error                      |
| 1             | Debugger call (single-step)       |
| 2             | NMI                               |
| 3             | Breakpoint                        |
| 4             | INTO                              |
| 5             | BOUND range exceeded              |
| 6             | Invalid opcode                    |
| 7             | Device not available              |
| 8             | Double fault                      |
| 9             | Reserved                          |
| 10            | Invalid task state segment        |
| 11            | Segment not present               |
| 12            | Stack exception                   |
| 13            | General protection                |
| 14            | Page fault                        |
| 15            | Reserved                          |
| 16            | Floating-point error              |
| 17            | Alignment check                   |
| 18            | Machine check                     |
| 19-31         | Reserved                          |
| 32-255        | Maskable interrupts (nonreserved) |

**FIGURE 5.1** Interrupt vector table

Vectors 32 through 255 are unassigned and free for us to use. To initialize an interrupt vector, we must write the 4 bytes of the interrupt service routine address into the table locations reserved for the interrupt. A short example shows one way this can be done.



## ■ EXAMPLE 5.1

The interrupt service routine for a type-40 interrupt is located at address 28000H. How is the interrupt vector table set up to handle this interrupt?

**Solution:** An easy way to determine the address within the interrupt vector table that is used by an interrupt is to multiply the interrupt number by 4. Multiplying 40 by 4 and converting into hexadecimal gives 000A0H as the starting address of the vector for INT 40. The interrupt service routine address 28000H can be generated by many different combinations of CS and IP values. If CS is loaded with 2800H and IP with 0000, we get the correct address of 28000H. Thus, it is necessary to write these two address values into memory starting at 000A0H. The short section of code shown here is one way to do this:

```

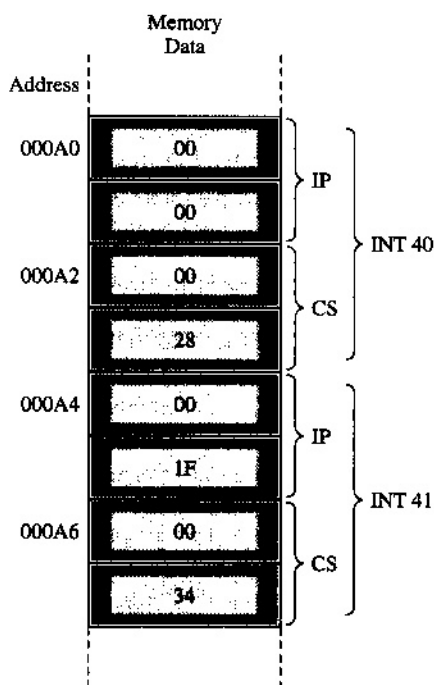
PUSH    DS                      ;save current DS address
MOV     AX,0                    ;set new DS address at 0000
MOV     DS,AX
MOV     DI,00A0H                ;offset for INT 40 vector
MOV     WORD PTR [DI],0         ;store IP address
MOV     WORD PTR [DI + 2],2800H ;store CS address
POP     DS                      ;get old DS address back

```

The PUSH DS instruction is used to save the current value of the DS register on the stack. Because we need to access memory in the 00000 to 003FFH range, it is convenient to load DS with 0000 and use DI as the offset into the vector table. Notice that the first word written is 0000, which goes into locations 000A0H and 000A1H. Then the CS value 2800H is written into locations 000A2H and 000A3H. Figure 5.2 provides a snapshot of memory after these instructions execute. Now, when INT 40 executes, it will cause a jump to the interrupt service routine located at address 28000H. The POP DS instruction restores the old value of the DS register.

Figure 5.2 also shows the contents of memory for the vector associated with INT 41. What is the address of the ISR? As usual, the two words have been byte-swapped. The

**FIGURE 5.2** ISR address for INT 40H and INT 41H



word at address 000A4H is 1F00H. The word at address 000A6H is 3400H. The effective address created by the addition of these two words is 35F00H, which is the address of the ISR for INT 41.

The machine code for INT 40 is CD 28 (note that 28H equals 40 decimal). The machine code for INT 41 is CD 29. All INT instructions begin with CD as their first byte and have the interrupt number as the second byte. The only exception to this rule is INT 3, *breakpoint*, which has only the byte CC as its opcode. ■

In the examples presented later, we will see other ways in which the interrupt vector table can be initialized.

## 5.4 THE INTERRUPT PROCESSING SEQUENCE

In Chapter 4 we were introduced to the interrupt process in the coverage of the INT and INTO instructions. These software interrupts initiate a sequence of steps in which the flags and return address are saved prior to loading CS and IP with the ISR address. The same process is followed for hardware interrupt NMI, which automatically generates a type-2 interrupt. The sequence initiated by INTR (when interrupts are enabled) is slightly different, because the processor must first read the interrupt number from the data bus. When interrupts are enabled, INTR causes the processor to perform two **interrupt acknowledge cycles**. In the 8088, external devices recognize these cycles by examining the state of the  $\overline{INTA}$  output, which goes low during each cycle. The first low-going pulse on  $\overline{INTA}$  is used to indicate to other devices on the system bus that the processor is beginning an interrupt acknowledge cycle. In minimum mode this indicates that the processor will not acknowledge a hold request until the interrupt acknowledge cycle completes. In maximum mode the processor activates its  $\overline{LOCK}$  output to prevent a system bus takeover during the interrupt acknowledge cycle.

The second low pulse on  $\overline{INTA}$  indicates that the interrupt number should be placed onto the lower byte of the processor's data bus. A special peripheral designed to respond to the 8088's interrupt acknowledge cycle is the 8259A programmable interrupt controller, which we will cover in Chapter 11.

In the Pentium, output signals  $\overline{M/\overline{IO}}$ ,  $\overline{D/\overline{C}}$ ,  $\overline{W/\overline{R}}$ , and  $\overline{ADS}$  all go low to indicate an interrupt acknowledge cycle.

Once the interrupt number is read from the data bus, the processor performs all of the steps that we are familiar with. Let us review the overall process once more.

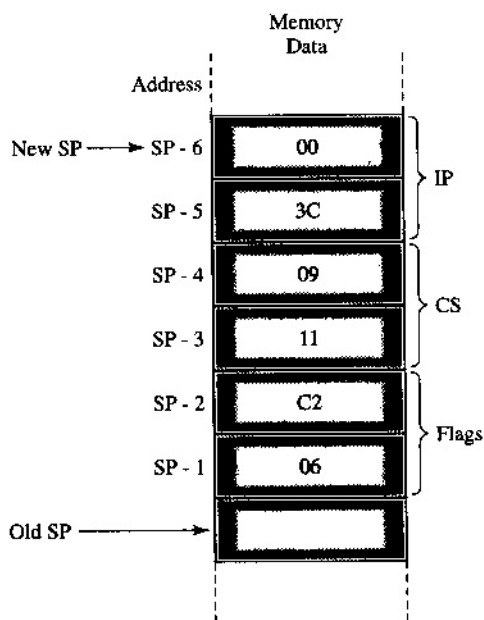
### Get Vector Number

The processor obtains the interrupt number in one of three ways. First, the interrupt type number may be specified directly using one of the INT instructions. Second, the processor may automatically generate the interrupt type number, as it does for INTO, NMI, and divide-error. Third, it may have to read the interrupt type number from the data bus (after receiving INTR).

Once the interrupt number is obtained, it is used to form the location within the interrupt vector table that contains the requested ISR address.

### Save Processor Information

Once the interrupt vector is known, the processor pushes the flag register onto the stack. This is done to preserve some of the processor's internal state at the time of the interrupt (a very necessary step if we are to resume normal execution later). Once the flags are pushed,

**FIGURE 5.3** Stack contents during an interrupt

the processor clears the interrupt enable and trace flags to disable INTR while interrupt processing is taking place. Next, the IP and CS values at the time of the interrupt are pushed onto the stack. Figure 5.3 shows how the stack is used when an interrupt occurs. The contents of the flag register at the time of the interrupt were 06C2H. The address of the instruction that was about to be fetched when the interrupt occurred was 1109:3C00 (in CS:IP format).

### Fetch New Instruction Pointer

Once the return address has been pushed, the processor can fetch the new values of IP and CS out of the interrupt vector table and begin execution of the interrupt service routine. The address generated by the interrupt number is used to read the two ISR address words out of the table.

*One word of caution:* Because the stack contains the information needed to return to the interrupt point, we must be careful not to change the contents of stack memory or alter the stack pointer in any way that would prevent the correct information from being popped off. The processor will not remember anything about the interrupt and relies only on the data popped off the stack for a proper return.

## 5.5 MULTIPLE INTERRUPTS

In the course of program execution, chances are good that eventually two interrupts might request the processor's attention at the same time. For example, just as division-by-zero is attempted in an executing program, NMI is also activated. The processor needs to "break the tie" when this happens and recognize one of the two interrupts first. When we examined Table 5.1, we saw that divide-error has a higher priority than NMI. So, in our current example, divide-error will be recognized first, and the following sequence of steps will occur:

1. Divide-error is recognized.
2. The flags are pushed.

3. The return address (CS and IP) is pushed.
4. The interrupt-enable and trace flags are cleared.
5. NMI is recognized.
6. The new flags are pushed.
7. The new return address is pushed.
8. The interrupt-enable and trace flags are cleared.
9. The NMI ISR is executed.
10. The second return address is popped.
11. The second set of flags is popped.
12. The divide-error ISR is executed.
13. The first return address is popped.
14. The first set of flags is popped.
15. Execution resumes at the instruction following the one that initiated the divide-error.

It is easy to see that the stack plays an important role during this process.

A more common occurrence of a multiple interrupt is seen when the processor's trace flag is set. The trace flag, when set, puts the processor into single-step mode, where a type-1 interrupt is generated after completion of every instruction. If the current instruction is INT or INTO, you can see that two interrupts will need servicing: the INT or INTO interrupt and the single-step interrupt. Single-step has the lowest priority of all interrupts and thus gets recognized last. We will see how single-stepping with the trace flag can be a useful tool when debugging a program.

---

## 5.6 SPECIAL INTERRUPTS

We will now examine the specific operation of a few selected interrupts. Some may be very useful to implement, while others may never be needed in a program for proper operation. Even so, you are better off knowing how each one works and when to use it.

### Divide-Error

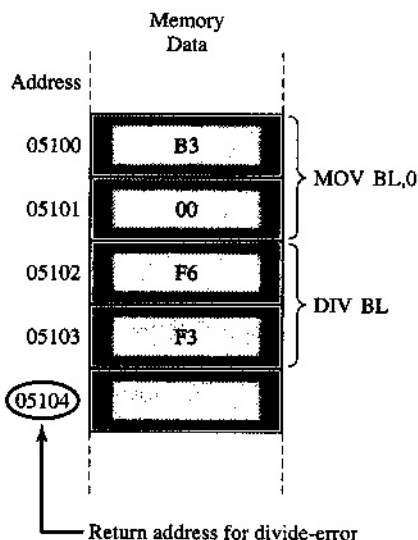
Figure 5.4 shows the contents of four memory locations that contain the code for these two instructions:

```
B3 00  MOV  BL,0
F6 F3  DIV  BL
```

The first instruction is located at address 05100H. The second instruction is located at address 05102H. Because DIV BL is a 2-byte instruction, the instruction must be located at address 05104H. This address becomes the return when DIV BL generates a divide-error interrupt.

The interrupt service routine for divide-error can do anything the user wishes to recover from the error. One programmer may wish simply to load the accumulator with 0 or some other number, while another may want to display an error message on the user's display screen.

Because divide-error is a type-0 interrupt, its address vector is stored in memory locations 00000 through 00003.



**FIGURE 5.4** Instruction sequences causing a divide-error Interrupt

## Single-Step

This interrupt relies on the setting of the trace flag in the flag register. When the trace flag is set, the processor will generate a type-1 interrupt after each instruction executes. Remember from the interrupt processing sequence that after the flags are pushed the processor clears the trace flag, disabling single-stepping while it executes the trace ISR. An extremely useful debugging tool can be written and used within the trace ISR. This single-step debugger may be programmed to display the contents of each processor register, the state of the flags, and other useful information after execution of each instruction in a user program. Because the trace flag is cleared before the ISR is called, we need not worry about single-stepping through the trace ISR.

You may remember from our coverage of 80x86 instructions in Chapters 3 and 4 that there are no instructions available that directly affect the state of the trace flag. There are other techniques that can be used to do this. For instance, a copy of the flags can be loaded into AX by first pushing a copy of the flag register onto the stack and then popping them into AX. Then an OR instruction can be used to set the trace flag. Once this is done, the flags are restored by pushing AX back onto the stack and then popping the flags. The instructions needed to accomplish this are:

```
PUSHF
POP AX
OR AX,100H
PUSH AX
POPF
```

Once this is done, the processor will enter and remain in single-step mode until the trace flag is cleared. This can be done with the same set of instructions, replacing the OR with AND AX,0FEFFH.

**■ EXAMPLE 5.2**

Assume that the trace flag is set and that the trace ISR displays the contents of AX on the screen after each instruction executes. What do we expect to see when this group of instructions executes?

```
MOV AX, 1234H
INC AL
DEC AH
NOT AX
```

**Solution:** Because the trace flag is set, a single-step interrupt will be generated after each of the four instructions. When the first instruction completes, the trace ISR will display AX=1234. The second instruction will increase AL to 35, causing the ISR to display AX=1235 next. The third instruction will decrease AH to 11. The trace ISR will then display AX=1135. Finally, after all bits in AX are inverted, the trace ISR will display AX=ECCA. We will see that the personal computer has built-in routines capable of displaying messages and data on the screen, so writing a trace ISR is not as complicated a task as it appears. ■

The ISR address vector for single-step must be stored in memory locations 00004 through 00007.

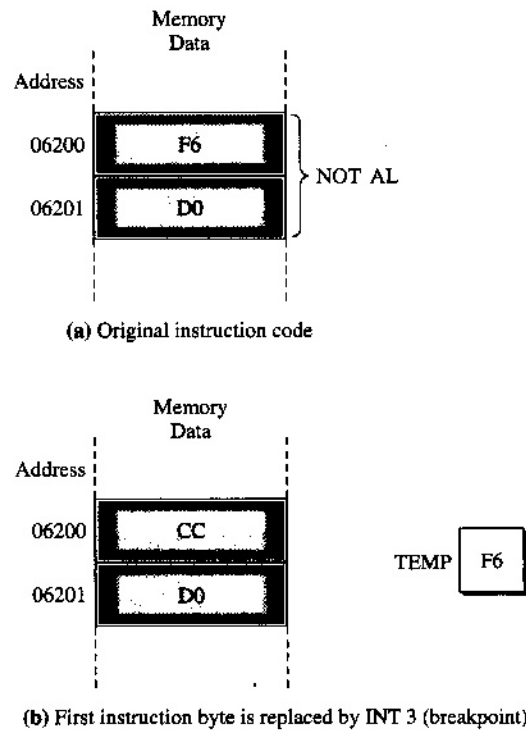
### NMI

Because NMI (non maskable interrupt) can never be ignored by the processor, it finds useful application in events that the computer absolutely must respond to. One such event is the disastrous power-fail. The processor unfortunately forgets the contents of its registers and flags when power is turned off and thus has no chance of getting back to the correct place in a program if its power is interrupted. One way to prevent this from happening and provide a way for the processor to resume execution is to use NMI to interrupt the processor at the beginning of a power-fail. Because the computer's power supply will continue to supply a stable voltage for a few milliseconds after it loses AC, the processor has plenty of time to execute the necessary instructions. Suppose that a certain system contains a small amount of non-volatile memory. This type of memory retains its data after it loses power and acts like RAM when power is applied. So, in the event of a power-fail, the NMI ISR should store the contents of each processor register in the NVM. These values can then be reloaded when power comes back up. In this fashion we can recover from a power-fail without loss of intelligence.

The ISR address vector for NMI is stored in memory locations 00008 through 0000BH.

### Breakpoint

This interrupt is a type-3 interrupt but is coded as a single byte for reasons of efficiency. Breakpoint aids in debugging in the following way: a program being debugged will have the first byte of one of its instructions replaced by the code for breakpoint (CC). When the processor gets to this instruction, it will generate a type-3 interrupt. The ISR associated with breakpoint is similar to the trace ISR and should be capable of displaying the processor register contents and also the address at which the breakpoint occurred. Before the ISR exits, it will replace the breakpoint byte with the original first byte of the instruction. Figure 5.5 shows how the breakpoint routine makes a copy of the first byte in the NOT AL instruction stored at location 06200H. The first byte of the instruction (F6H) is copied into a temporary location and then replaced by the breakpoint instruction code CC. Some people like to refer to this as *setting the breakpoint*. Once the breakpoint is set, a fetch from address 06200H will initiate a breakpoint interrupt.

**FIGURE 5.5** Setting a breakpoint

Clearing the breakpoint is accomplished by copying the instruction byte from the temporary location back into its original location.

### ■ EXAMPLE 5.3

A programmer wishes to find out if a conditional jump takes place. Where should the programmer place the breakpoint instruction? The code being tested looks like this:

```

CMP    AL, 0
JNZ    XYZ
NOT    AL
XYZ:   INC    AL

```

**Solution:** The programmer has two choices for placement of the breakpoint instruction. It could be placed in the location occupied by `NOT AL`, which would cause a breakpoint when the `JNZ` does *not* jump to `XYZ`. It could also be placed in the location occupied by `INC AL`, activating when the `JNZ` *does* take place. Either way, the programmer will know the results of the `CMP` and `JNZ` instructions (by the presence or absence of a breakpoint). ■

The ISR address vector for breakpoint is stored in memory locations 0000CH through 0000FH.

### Overflow

This type-4 interrupt is initiated only when the `INTO` instruction is executed with the overflow flag set. Its applications, like divide-error, tend to be of a corrective nature. You may think of overflow as the watchdog for multi-bit addition and subtraction operations, much like divide-error watches out for division-by-zero. If the overflow flag is cleared, `INTO` will not generate an interrupt (essentially operating as a `NOP` in this case).

■ **EXAMPLE 5.4** Will the following sequence of code generate an overflow interrupt?

```
MOV AL, 70H
MOV BL, 60H
ADD AL, BL
INTO
```

**Solution:** Yes. Although the numbers in AL and BL can both be interpreted as positive signed numbers, the sum (D0H) looks like a negative signed number. In this case the overflow flag is set and INTO will generate an interrupt. ■

The ISR address vector for overflow is stored in memory locations 00010H through 00013H.

## INTR

Up to now we have only discussed the basics of this hardware interrupt signal. Let us take a closer look. No interrupt is generated by INTR unless the interrupt-enable flag (IF) is set. This can easily be accomplished with the STI (set interrupt-enable flag) instruction. INTR must remain high until sampled by the processor, unlike NMI, which is a rising-edge triggered input. It is therefore necessary when using INTR to allow it to remain high only until an interrupt acknowledge cycle begins. For 8088 systems, the 8259A programmable interrupt controller, which we will cover in Chapter 11, automatically interfaces with INTR and  $\overline{INTA}$ . If the power of this peripheral is not needed, then custom interrupt circuitry must be designed. First, we need the INTR connection. The circuit shown in Figure 5.6 uses a flip-flop to condition the INTR input. A D-type flip-flop is used to convert the high-level requirement of INTR into an edge-sensitive request. A rising edge on MINT (maskable interrupt) will clock a 1 through the flip-flop, placing a high level on INTR. When the 8088 recognizes INTR and begins its interrupt acknowledge cycle,  $\overline{INTA}$  will go low. This will clear the flip-flop and remove the INTR request. MINT must go low and back high again for another interrupt to be recognized. Once again, the Pentium uses different signals to indicate an interrupt acknowledge cycle. Figure 5.7 shows how the signals are decoded to generate an  $\overline{INTAK}$  signal.

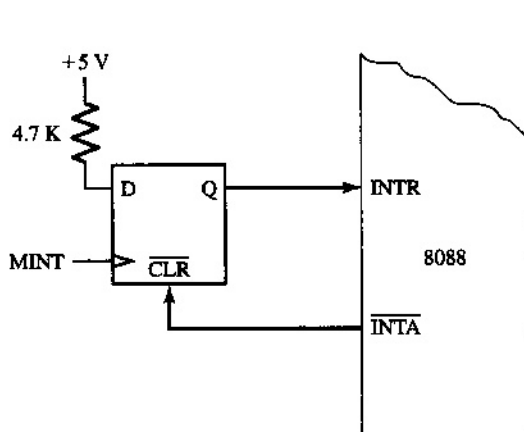


FIGURE 5.6 INTR conditioning circuitry

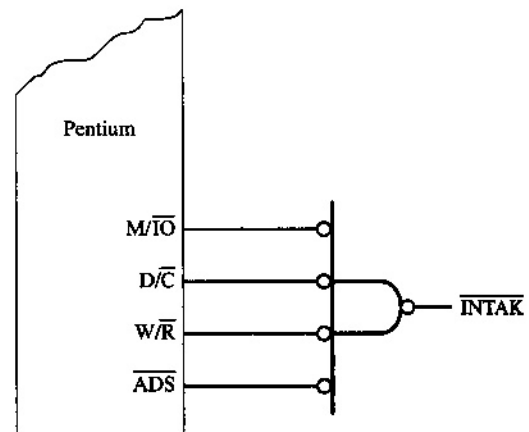
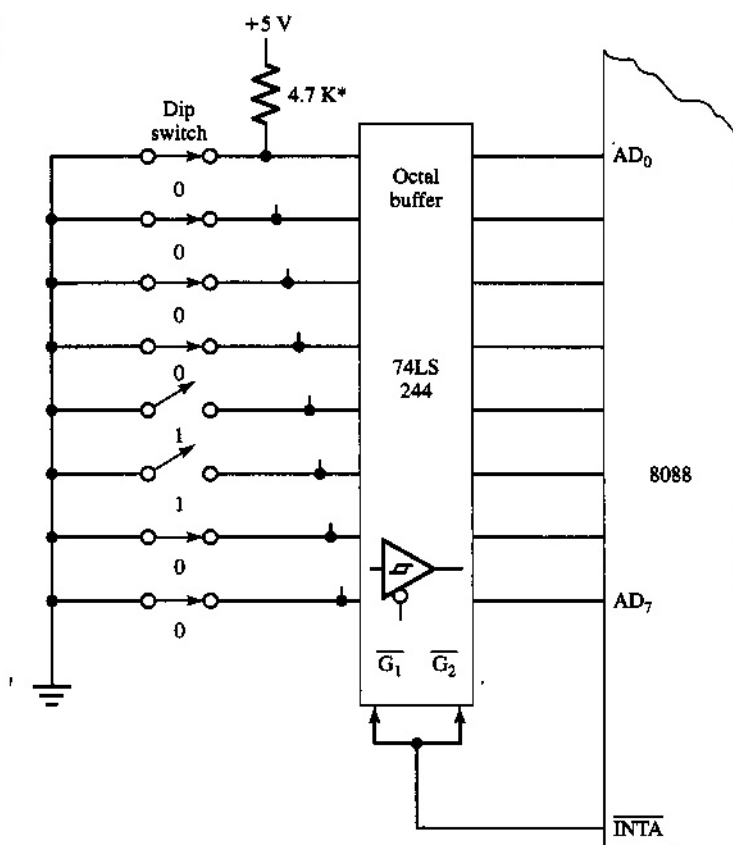


FIGURE 5.7 Decoding an interrupt acknowledge cycle on the Pentium



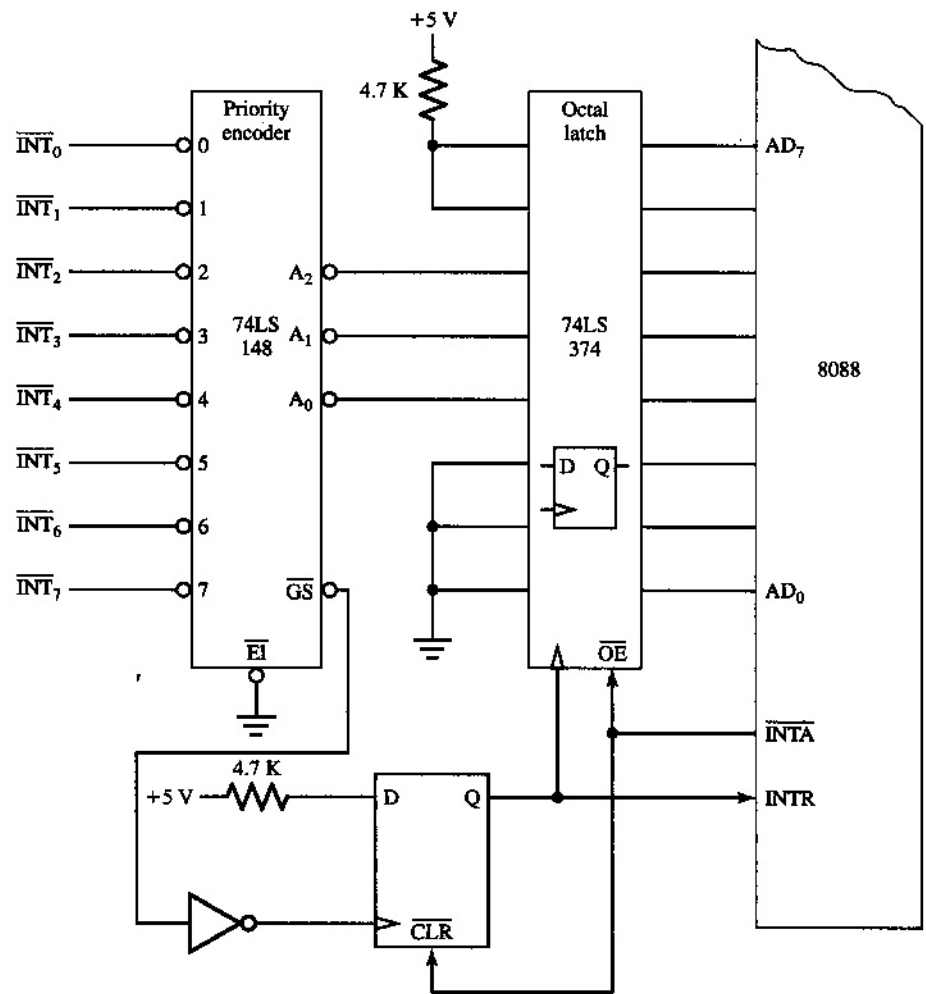
**FIGURE 5.8** Circuitry needed to place an 8-bit interrupt number onto data bus



\*All switches must be pulled up with individual resistors.

During the second low-going pulse on  $\overline{INTA}$ , the processor will expect an interrupt number to be placed onto the data bus. The additional circuitry of Figure 5.8 uses a tri-state buffer to jam the 8-bit interrupt number onto the data bus when  $\overline{INTA}$  goes low. You may notice that the interrupt number will appear on the data bus twice, once for each low-going transition of  $\overline{INTA}$ . The processor will ignore the first appearance because it tri-states the data bus during the first transition of  $\overline{INTA}$ . The 8-bit interrupt number will appear on the data bus but will be ignored by the processor until the second transition of  $\overline{INTA}$ . The DIP switch allows any of the 256 interrupt codes to be used. The DIP switch is currently set to produce interrupt code 30H. On the Pentium, the interrupt vector is applied to data bus signals D0 through D7.

For systems that require additional interrupts but still do not require the use of the 8259A, a few additional parts are needed to expand this simple interrupt circuit into a more complex one with more interrupts and a prioritization scheme. Figure 5.9 shows an interrupt circuit that allows up to eight levels of prioritized interrupts. The 74LS148 priority encoder will output a 3-bit binary number whenever any of its eight inputs go low. Also, only the highest priority input is recognized. So, if  $\overline{INT_0}$  and  $\overline{INT_4}$  are both low,  $\overline{INT_4}$  is recognized and the output becomes 011. Note that the output is actually the inverted binary value of the input that is active. When any input is grounded, the  $\overline{GS}$  output will go low. This causes a 1 to be clocked through the D-type flip-flop, signaling an INTR and latching the priority encoder's output in the 74LS374 octal flip-flop. When  $\overline{INTA}$  goes low in



**FIGURE 5.9** Prioritized interrupt circuitry

acknowledgment of the INTR signal, the D-type flip-flop is cleared (removing the INTR request) and the output of the octal flip-flop is enabled, placing the interrupt number onto the data bus. A similar circuit may be used with the Pentium. The bit pattern stored in the octal flip-flop is 11(???)000 for any interrupt. Specifically,  $\overline{\text{INT}}_0$  causes the outputs of the 74LS148 to become 111, giving us the interrupt number F8H.  $\overline{\text{INT}}_1$  generates 110 at the 148's output, for an interrupt code of F0H. Take the time to find the other six interrupt numbers right now. Do you see a pattern emerge?

## 5.7 INTERRUPT SERVICE ROUTINES

The interrupt service routine referred to many times in this chapter is the actual section of code that takes care of processing a specific interrupt. The ISR for a divide-error is necessarily different from one designed to handle breakpoint or NMI interrupts.

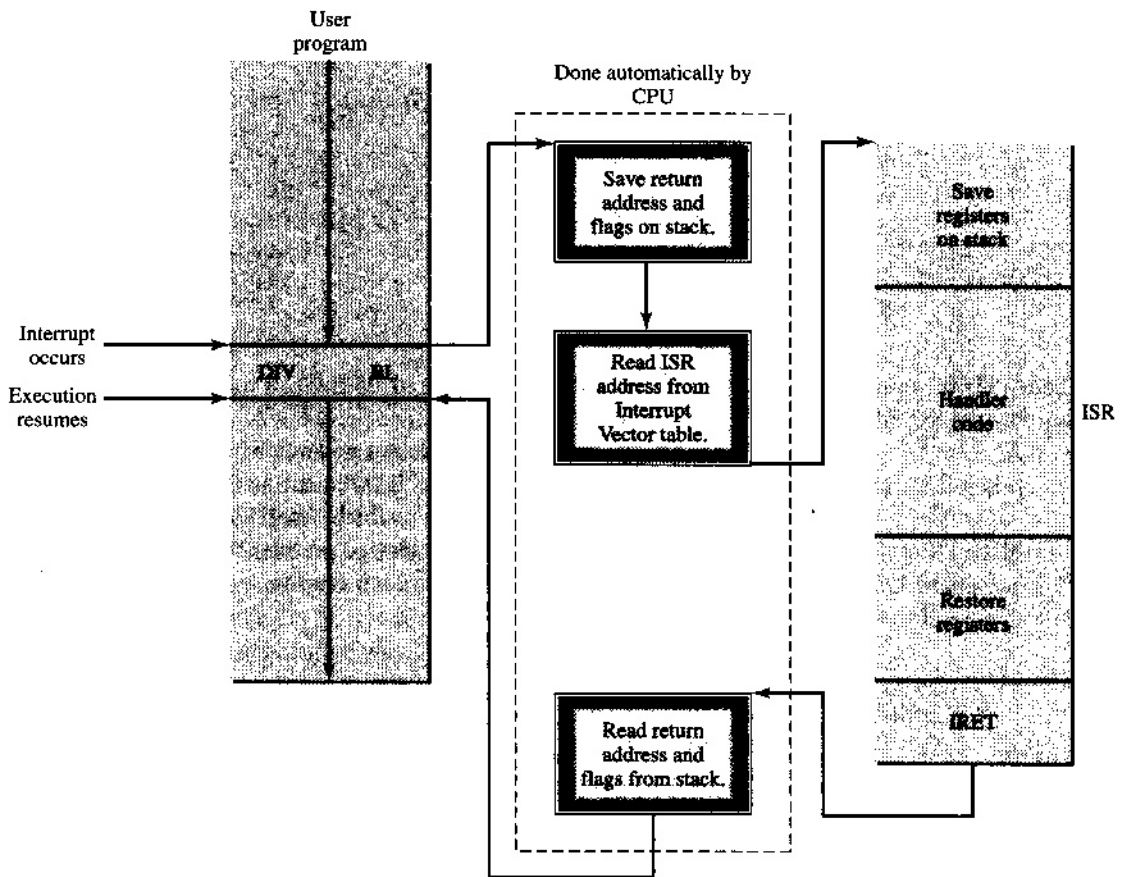


FIGURE 5.10 Storing environment during interrupt processing

Even though these interrupt service routines are written to accomplish different goals, there are portions of each that, for the sake of good programming, look and operate the same. Recall that any time an interrupt occurs, the processor pushes the flags and return address onto the stack before vectoring to the address of the ISR. Clearly, we must see that the ISR will change the data in various registers while it is processing the interrupt. Because we desire to return to the same point in our program where we left off *before* the interrupt occurred and resume processing, we insist that all prior conditions exist upon return. This means that we must return from the ISR with the state of all registers preserved. It is now the responsibility of the ISR to preserve the state of any registers that it alters. Figure 5.10 shows how this is done. In this example, DIV BL causes a divide-error interrupt. The first thing the ISR does is save the processor registers on the stack. These registers are saved with the PUSH instruction. You will need a PUSH for each register you need to save (or you can push all the registers with PUSH A). The contents of all processor registers, including the flags, are often called the **environment** or **context** of the machine. Putting a copy of everything onto the stack saves the environment that existed at the time of the interrupt.

When the body of the ISR finishes execution, it is necessary to reload the registers that were saved at the beginning of the routine. The POP instruction is used for this

purpose. POPs must be done in the reverse order of the PUSHes. An ISR that uses AX, BX, and CX would look something like this:

```
ANISR:  PUSH  AX      ;save registers
        PUSH  BX
        PUSH  CX
        ;
        ;body of ISR
        ;

        POP   CX      ;get registers back
        POP   BX
        POP   AX
        IRET          ;return from interrupt
```

Saving all registers is preferable, and will save you much heartache in the future, when you find that saving one or two registers was insufficient as the needs of the routine became more complex. In this case, use POPA to restore all registers.

A few examples of actual interrupt service routines will prepare you for writing your own later. Try to find similarities between each routine.

### An NMI Time Clock

Figure 5.11 shows a simple way to provide the processor with some timekeeping intelligence. In this application, the processor's NMI input is connected to a 60-Hz clock source. Thus, the processor gets interrupted 60 times per second. The only task the NMITIME ISR needs to perform is to decrement a counter until it reaches 0, and then call the far routine ONESEC. ONESEC is then called once every second. The counter is decremented once for each NMI signal. Other code is needed to initially set the count to 60, with NMITIME resetting it automatically on each 0 count.

```
NMITIME: DEC  COUNT          ;decrement 60th's counter
          JNZ  EXIT          ;did we go to 0?
          MOV  COUNT,60      ;yes, reset counter and
          CALL FAR PTR ONESEC ;call ONESEC
EXIT:     IRET
```

The user program must reserve room for the COUNT location in its data segment area. COUNT DB 60 is all that is needed to reserve the byte location. Initialization software is

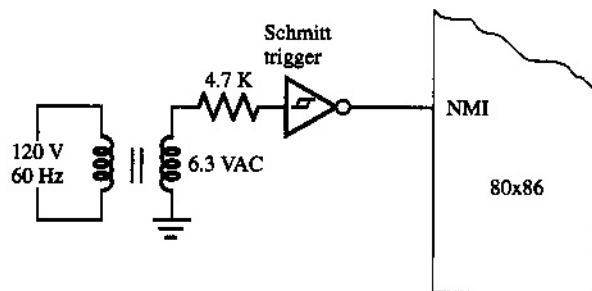


FIGURE 5.11 Interrupt circuit for NMITIME

also needed to load 60 into COUNT and place the ISR address for NMITIME into the interrupt vector table. One way to do this would be:

```
MOV  COUNT,60          ;init 60th's counter
PUSH DS                ;save current DS address
SUB  AX,AX              ;set new DS address to 0000
MOV  DS,AX
LEA  AX,NMITIME         ;load address of NMITIME ISR
MOV  [8],AX             ;store IP address
MOV  AX,CS              ;and CS address
MOV  [0AH],AX
POP  DS                ;get old DS address back
```

The offset of NMITIME within the code segment is stored in locations 00008 and 00009, and the CS value in locations 0000AH and 0000BH. The value of the DS register remains unchanged after execution of the initialization software.

Notice that NMITIME does not destroy a single register. In this example we can get by without having to save anything on the stack.

ONESEC can be used to do a number of things. Most likely, it will update a second set of count locations that keep track of the seconds, minutes, and hours for a 12- or 24-hour clock. Software with access to these counters will be able to use the passage of time in an accurate way.

## A Divide-Error Handler

This routine is used to handle a division-by-zero. Because the AX and DX registers may be undefined as a result of division by 0, DIVERR will load AX with 0101H and DX with 0. This guarantees that 8- or 16-bit division always ends up with a non-zero result. DIVERR also calls a special routine called DISPMSG, which is used to output an ASCII text message on the display screen of the computer. The ASCII message must end with a "\$" character, and the address of the first byte of the message must be loaded into the SI register prior to calling DISPMSG.

```
DIVERR:  PUSH SI          ;save current SI value
         MOV  AX,101H      ;load result with default
         SUB  DX,DX        ;clear DX
         LEA  SI,DIVMSG    ;init pointer to error message
         CALL FAR PTR DISPMSG ;output error message
         POP  SI          ;get old SI value back
         IRET
```

PUSH and POP are used to preserve the contents of SI. The error message must be located in the program's data segment and look similar to this:

```
DIVMSG  DB  'Division by zero attempted!$'
```

The first byte of the text message ("D") will be located in the address associated with the label DIVMSG. The last byte of the message is the required "\$" character.

## An ISR with Multiple Functions

This interrupt service routine will be used to perform one of four different functions when it is executed. The ISR is called with an INT 20H instruction. Register AH is examined

upon entry into ISR20H to determine what should be done. If AH equals 0, AL and BL will be added together, with the result placed in AL. If AH equals 1, the registers will be subtracted. Multiplication occurs when AH equals 2, and division when AH equals 3. Any other values of AH cause ISR20H to return without changing either register. Using AH in this way lets us do more than one thing with INT 20H. This technique is commonly used when we write programs for 80x86-based PCs running a disk operating system. One interrupt might have many different functions, all of which interface with a disk drive, display device, or printer connected to the computer. We will see specific examples of these kinds of special-function interrupts in the next chapter. The code for this ISR looks like this:

```
ISR20H:  CMP     AX,4      ;AH must be 0-3 only
         JNC     EXIT
         CMP     AH,0      ;is AH 0?
         JZ      ADDAB
         CMP     AH,1      ;is AH 1?
         JZ      SUBAB
         CMP     AH,2      ;is AH 2?
         JZ      MULAB
         DIV     BL        ;AH is 3, use divide function
EXIT:    IRET
ADDAB:   ADD     AL,BL     ;add function
         IRET
SUBAB:   SUB     AL,BL     ;subtract function
         IRET
MULAB:   MUL     BL        ;multiply function
         IRET
```

The method of using a register value to select an interrupt function is also used by MS-DOS programs that require the use of operating system functions. For example, the instructions

```
LEA     DX,MESSAGE
MOV     AH,9
INT     21H
```

select the MS-DOS Display Message function of INT 21H. The function number is passed to INT 21H through register AH. The address of the ASCII message to display must be placed in DX prior to the interrupt.

We will make extensive use of the MS-DOS interrupts in Chapter 6.

The initialization code required for ISR20H is:

```
PUSH    DS                ;save old DS address
SUB     AX,AX             ;set new DS address to 0000
MOV     DS,AX
LEA     AX,ISR20H         ;load address of ISR20H
MOV     [80H],AX          ;store IP address
MOV     AX,CS             ;and CS address
MOV     [82H],AX
POP     DS                ;get old DS address back
```

It is easy to verify that 4 times 20H is 80H, the interrupt vector table address required.

---

## 5.8 TROUBLESHOOTING TECHNIQUES

It pays to remember the details of interrupt processing when troubleshooting a program. Many times the fault of erratic execution in an 80x86-based system is a poorly written or incomplete interrupt handler. Reviewing the basic principles can help eliminate some of the more obvious problems.

- A valid stack must exist to save all of the information required to support the interrupt handler.
- A typical interrupt pushes the current flags and return address (CS:IP).
- Vector addresses are equal to four times the vector number.
- In the interrupt vector table, the handler address is stored in byte-swapped form, with IP as the first word, and CS as the second.
- It may be necessary to save and restore registers (via PUSH/POP) in the interrupt handler.
- Use IRET (return from interrupt) to return from an interrupt handler. RET does not work properly with interrupts.
- For an interrupt to work, its vector must be loaded with the starting address of the handler, and the handler code must be in place as well.

These points are a minimal set of programming tips. You will discover others as you begin writing your own interrupt handlers.

---

## SUMMARY

We have seen that there is a fixed process used by the 80x86 to implement and process an interrupt. The CPU, when interrupted, saves the flag register and program counter on the stack, clears the trace and interrupt-enable flags, and loads the interrupt service routine address from the interrupt vector table. The interrupt vector table occupies memory locations 00000 through 003FFH and contains pairs of words that represent the execution addresses for each of the 256 interrupts. These pairs correspond to IP and CS values of each ISR. The interrupt number used to access the table may be internally generated by the processor, or may be supplied by external hardware during an interrupt acknowledge cycle. The processor has only two hardware interrupts: NMI and INTR. NMI cannot be disabled, but INTR can.

Interrupts are caused through software or by an external hardware request. The software interrupt may be generated intentionally by the programmer via INT and INTO, or by accident, via a run-time error such as division-by-zero. All interrupt service routines should preserve the state of any registers used to allow a proper return.

Three examples of actual interrupt service routines were also presented, followed by a short set of troubleshooting tips.

---

## STUDY QUESTIONS

1. What is the processor's environment? Why is it important to save the environment during interrupt processing?
2. Explain the different ways interrupts are generated.

3. How is INTR disabled? How is it enabled?
4. What is the interrupt vector table address for an INT 21H?
5. The address of the ISR for INT 25H is 03C0:9AE2 (in CS:IP format). Show how this address is stored within the interrupt vector table.
6. What is the effective address of the ISR in Question 5?
7. Write the instructions necessary to place the ISR address of Question 5 into its proper place in the interrupt vector table.
8. Show the contents of stack memory after an interrupt has been initiated. Assume that the stack pointer is at address 3C00H prior to the interrupt and that CS, IP, and the flag register contain 0400H, 1890H, and 0182H, respectively.
9. What interrupt number has a vector table address range of 00280H to 00283H?
10. Which interrupt is recognized first, NMI or single-step?
11. Repeat Example 5.2 with an initial AX value of E03FH.
12. What high-priority event might require the use of NMI in a computer designed for aircraft engine control?
13. An analysis of a computer power failure showed that the computer had valid voltage levels for 2.5 milliseconds. Suppose that the microprocessor had an average instruction execution time of 0.2 microseconds. How many instructions can be executed during the power failure?
14. Explain why an orderly software shutdown is possible in Question 13. Assume that the shutdown involves pushing all processor registers onto a stack in NVM.
15. In Example 5.4 what is the highest AL value that will not cause an INTO interrupt?
16. Design an INTR circuit that has only two inputs:  $\overline{XINT}$  and  $\overline{YINT}$ . Both signals are active low.  $\overline{XINT}$  should generate interrupt number 90H, and  $\overline{YINT}$ , 91H.
17. What are all eight interrupt numbers for the circuit in Figure 5.9?
18. Modify the interrupt circuit of Figure 5.9 so that interrupt numbers 48H through 4FH are produced.
19. If the ONESEC procedure called by NMIME took more than 18 milliseconds to execute, would any problems arise?
20. What changes must be made to NMIME if the frequency of the NMI clock is 1800 Hz? We still want to call ONESEC once per second.
21. What is the result of executing INT 20H with AX containing 0303H and BL containing 04? Refer to ISR20H in Section 5.7 for details.
22. Rewrite ISR20H so that four new functions are added. These functions are:

```

AH=05H : NOT AL
AH=10H : AND AL,BL
AH=20H : OR AL,BL
AH=80H : XOR AL,BL

```

23. Figure 5.12 shows the contents of a few locations within the interrupt vector table. What will the new program counter be when the interrupt that uses these locations is processed?
24. What INT instruction is required in Question 23?
25. How is the single-step interrupt useful for examining the operation of a running program?
26. During an interrupt acknowledge cycle, 30H is placed on the data bus. Where is the ISR address fetched from?
27. The flag register contains 0346H. Will INTO generate an interrupt? Is trace enabled? Are interrupts enabled?



**FIGURE 5.12** For Question 23

| Address | Memory Data |
|---------|-------------|
| 003C0   | 34          |
| 003C1   | 12          |
| 003C2   | 00          |
| 003C3   | 64          |

28. What do you imagine are some of the problems with these two interrupt service routines:

|                                                                                                |                                                                                                              |
|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <pre>ISR1: PUSH  AX       PUSH  BX       ;       ;body       ;       POP   AX       IRET</pre> | <pre>ISR2: PUSH  AX       PUSH  CX       ;       ;body       ;       POP   BX       POP   AX       RET</pre> |
|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|

29. Write an interrupt service routine that will multiply the contents of AX by 7. If the new value of AX is greater than 8400H, call the far routine OVERSCAN. *Note:* OVERSCAN destroys AX, BX, and DI.
30. What instructions are necessary to load a copy of the return address (CS:IP) into registers AX and BX, from *inside* the interrupt service routine? What stack operations may be used?
31. Use DEBUG to execute these three instructions:

```
MOV  DL, 41
MOV  AH, 2
INT  21
```

What are the results? Remember to use "p" and not "t" to execute INT 21.

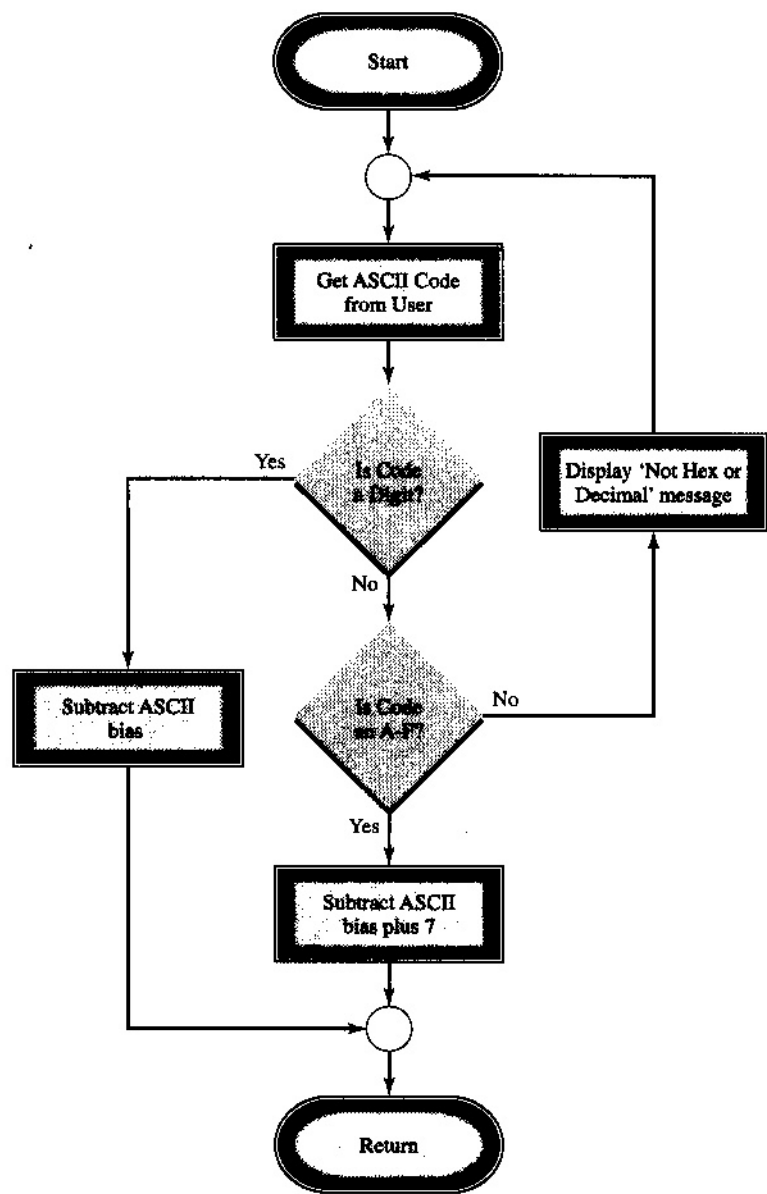
# PART 3

## Programming

---

- 6 An Introduction to Programming the 80x86
- 7 Advanced Programming Applications

Flowcharts allow a graphical approach to program design and analysis



---

## CHAPTER 6

---

# An Introduction to Programming the 80x86

---

### OBJECTIVES

In this chapter you will learn about:

- Breaking down a large program into small tasks
- How to write a software tester
- Reading character strings from a keyboard
- Packing BCD digits into a byte
- Item search and lookup in a data table
- Comparison of data strings
- Sorting algorithms
- The use of condition flags to return results from a routine
- Binary and BCD math
- Writing a routine to perform a complex mathematical function
- Open- and closed-loop control systems (simplified theory)
- How to convert binary numbers to decimal numbers, and vice versa
- Insertion of an item into a linked-list
- The operation of stacks and queues

### KEY TERMS

Binary-to-decimal  
conversion

Bubble sort

Command recognizer

Closed-loop control system

Decimal-to-binary  
conversion

FIFO structure

Function

Integrated development  
environment

Iterative formula

LIFO structure

Linked-list

Node

Normalization

Null

Open-loop control system

Polling

Pseudocode

Specification

Structured programming

---

## 6.1 INTRODUCTION

Getting the most use out of your microprocessor requires expertise both in designing the hardware around it and in writing the code that it will execute. The purpose of this chapter is to familiarize you with some of the standard programming principles as they apply to the 80x86. We will limit ourselves to writing assembly language in this chapter, using only the power of the 80x86 instruction set. Hopefully, you will see that many complex tasks may be performed in this way, without the use of external peripherals, which will be covered in Chapters 10 and 11.

Section 6.2 explains how large programming jobs are broken down into smaller tasks. Section 6.3 shows how a software driver program is written to test a new code module. Sections 6.4 and 6.5 deal, respectively, with collecting data and simple search techniques for use with data tables (or arrays). Section 6.6 details the various string operations. Integer sorting is covered in Section 6.7 and is followed by a detailed treatment of binary and BCD mathematical routines in Section 6.8. Section 6.9 shows two examples of the processor in control applications. Section 6.10 gives examples of binary and decimal conversion techniques. Section 6.11 discusses three basic data structures: linked-lists, stacks, and queues. The chapter concludes with another set of troubleshooting tips in Section 6.12.

---

## 6.2 TACKLING A LARGE PROGRAMMING ASSIGNMENT

Writing a large, complex program from scratch is a difficult job, even for the most seasoned programmers. Even if this could be done easily, other considerations exist to complicate matters. The final program must be tested to ensure correct operation. It is a rare occurrence for a new program to work perfectly the first time. Typically, an **integrated development environment (IDE)** containing a C compiler, assembler, and debugger is used during program development.

For these reasons, a more sensible approach is to break down the large program into smaller tasks. Each task may be thought of as a subroutine to be called, when needed, by the main program. The subroutines will each perform a single task and, thus, will be easier to individually test and correct, as necessary. The technique of writing a large program in this way is often referred to as **structured programming**. We will not concern ourselves with all the details of structured programming. Instead, we will study a sample programming assignment and use the techniques previously mentioned to break down the assignment into smaller jobs.

### The Assignment

The programming assignment is presented to us in the form of a **specification**. The specification describes the job that must be performed by the program. It also contains information concerning any input and output that may need to be performed, and sometimes a limit on the amount of time the program may take to execute.

Consider the following specification:

Specification: Subroutine WORDCOUNT

Purpose: To generate a data table of all different words contained in a paragraph of text, and a second data table containing the frequency of occurrence of each word.

- Restrictions:** Do not distinguish between uppercase and lowercase characters. Ignore punctuation, except where it defines the end of a word. No words will appear more than 255 times.
- Input:** A data table, headed by symbol TEXT, that contains the paragraph to be analyzed, represented by ASCII codes. The length of the paragraph text is undefined, but the last character in the text will always be "\$." This character will not appear anywhere else in the text.
- Output:** A data table, headed by symbol WORDS, that contains a list of all different words encountered in the paragraph text. Each word ends with ".", and the entire table ends with "\$." A second data table, headed by symbol COUNTS, contains the frequency counts for each entry in WORDS.

There is sufficient detail in the specification for us to determine what must be done. How to do it is another matter.

## Breaking Down the Program into Modules

Once we understand what is required of the program through information presented in the specification, the next step is to break down the program into smaller modules. This means that subroutine WORDCOUNT will actually become a main subroutine, which calls other subroutines. The idea is to create a subroutine to accomplish only *one* task. We must identify the other subroutines needed. This step of the process requires skill and practice. When you have given it enough thought, you might agree that these subroutines are required:

|            |                                                                |
|------------|----------------------------------------------------------------|
| INITIALIZE | Initialize all pointers, counters, and tables needed.          |
| GETWORD    | Get the next word from the paragraph text.                     |
| LOOKUP     | Search WORDS to see if it contains the present word.           |
| INSERT     | Insert new word into WORDS.                                    |
| MAKECOUNT  | Make a new entry in COUNTS.                                    |
| INCREASE   | Increase frequency count in COUNTS for a word found by LOOKUP. |

There may, of course, be other required routines, depending on who is writing the code. None of the identified routines performs more than one task.

Once the input and output for each subroutine are identified, the code can be written for each one and the subroutines tested.

## Testing the Modules

Testing of each subroutine module is done separately through a special software testing program. The tester supplies the subroutine with sample input data and examines the subroutine's output for correctness. It is up to the programmer to select the type and quantity of the sample data. We will look at an example of a software tester in Section 6.3.

When all modules have been tested and verified for proper operation, they can be combined into one large module—WORDCOUNT in our example—and this module can be tested also.

## Creating the Final Module

WORDCOUNT, as mentioned before, will consist of calls to the subroutines identified in the section on breaking down the program into modules. **Pseudocode**, a generic programming language, can be used to determine the structure of WORDCOUNT (and of the

other subroutines as well). The following pseudocode is one way WORDCOUNT may be implemented:

```

subroutine WORDCOUNT
  INITIALIZE
  repeat
    GETWORD
    if word found then
      LOOKUP
      if word found then
        INCREASE
      else
        INSERT
        MAKECOUNT
    while word found
  end WORDCOUNT

```

WORDCOUNT is implemented as a loop because the length of the paragraph text is unknown. The only way out of the loop is to have GETWORD fail to find a new word in the text (that is, by reaching the "\$"). This approach satisfies another requirement of structured programming: routines should contain one entry point and one exit point. Many of the routine examples that we will study in this chapter will be written in this fashion. The programmer decides how the REPEAT-WHILE and IF-THEN-ELSE statements are implemented.

## Using the C Language

Writing in pseudocode is useful, but writing statements in an actual programming language is also useful. For example, here is the C code used to implement the WORDCOUNT subroutine (called a **function** in C):

```

void wordcount()
{
    initialize();
    do
    {
        found = getword();
        if(found)
        {
            gotword = lookup();
            if(gotword)
                increase();
            else
            {
                insert();
                makecount();
            }
        }
    } while(!found);
}

```

As we saw with the pseudocode shown previously, the wordcount() function has one entry point and one exit point. The getword() and lookup() functions return values that are tested in the if() and if-else statements. Note that we are not concerned with writing compiler-ready code when first developing a program, but we want to get as close as possible and use good

programming habits so that a minimum of work is needed to finish the code. This includes proper indenting of statements, matching braces {}, and proper syntax.

## Standard Control Structures

The if() statement can be coded in many different ways. One way to code the if() statement might look like this:

```
if (found == 1)      MOV  AL, FOUND
{                   CMP  AL, 1
    lookup();       JNZ  NEXT
}                   CALL LOOKUP
                    NEXT: ---
```

Here, the value of the "found" variable will determine if the lookup() function is called. The CMP instruction is used to determine if AL contains 1. If it does not, a jump to NEXT is performed. If AL does contain 1, the JNZ will not take place and the CALL LOOKUP instruction will execute instead.

Other structures include the do-while() and while(). The do-while() structure looks like this:

```
do
{
    statements;
} while (condition);
```

Coding the do-while() structure depends on the type of condition being tested. A sample structure and its associated code may look like this:

```
counter = 100;      MOV    BX, 100
do                AGAIN:
{
    getdata();      CALL    GETDATA
    processdata();  CALL    PROCESSDATA
    counter = counter - 1; DEC    BX
} while (counter != 0); JNZ    AGAIN
```

One important point about using loop counters is that the loop-count register (BX in this example) must be altered during execution of the statements within the loop.

The while() structure is slightly different, performing the condition test at the *beginning* of the loop instead of the end. One example of a while() is:

```
while (char != 'A')
{
    statements;
}
```

The corresponding machine instructions for this loop might look like this:

```
WHILE:  CMP    AL, 'A'
        JZ     NEXT
        <statements>
        JMP    WHILE
NEXT:    ---
```

Here, it is important to modify the loop variable (AL in this case) somewhere within the loop, to avoid getting stuck inside it.

Another programming structure that is useful is the `switch()` statement. The structure of the `switch()` statement may look like this:

```
switch(item)
{
    case item-1 : statement-1; break;
    case item-2 : statement-2; break;
    .
    .
    case item-x : statement-x;
}
```

Each item (1...x) is checked for a match with the item at the beginning of the `switch()` statement. Only the statement associated with the matching item is executed. The number of items to match is not limited. An example of a `switch()` statement with three choices is as follows:

```
switch(selvalue)
{
    case 0 : counter = 0; break;
    case 1 : counter = counter + 1; break;
    case 2 : counter = counter - 1;
}
```

In this example, the *selvalue* variable must contain a value from 0 to 2 to select one of the three statements. The corresponding assembly language for this `switch()` statement is:

```

        CMP     AL,0       ;is it 0?
        JNZ     C1
        MOV     BL,0       ;clear counter
        JMP     NEXT
C1:     CMP     AL,1       ;is it 1?
        JNZ     C2
        INC     BL         ;increment counter
        JMP     NEXT
C2:     CMP     AL,2       ;is it 2?
        JNZ     NEXT       ;no matches
        DEC     BL         ;decrement counter
NEXT:   ---
```

In this example, the *selvalue* variable is stored in AL and the counter is represented by BL. Note that the conditional jump JNE may be used in place of JNZ if desired, because they are equivalent. The same is true for conditional jumps JZ and JE.

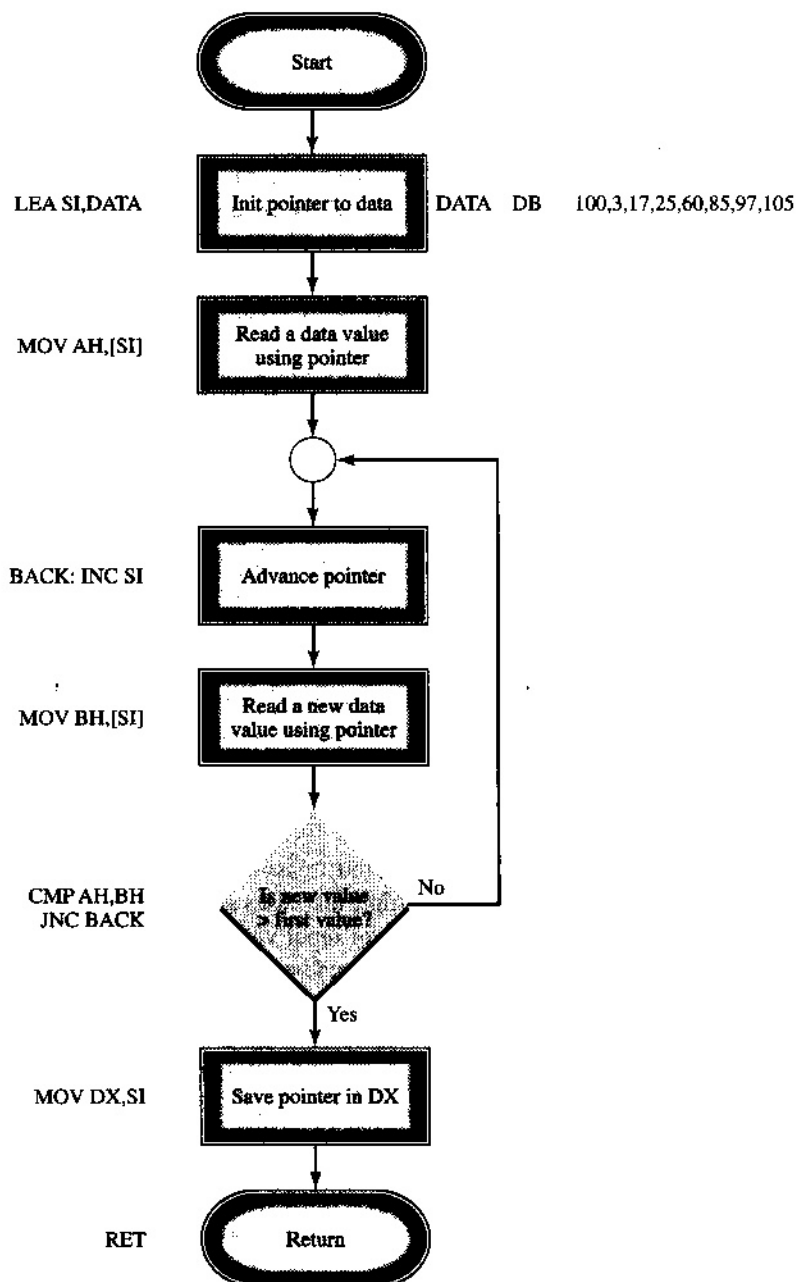
Some programmers use a modified form of the `switch()` structure that allows execution of a statement when no match is made with any item. The pseudocode for this structure is:

```
switch(item)
{
    case item-1 : statement-1; break;
    case item-2 : statement-2; break;
    .
    .
    case item-x : statement-x; break;
    default : default-statement;
}
```

It is a simple matter to execute the default statement when no matches are found by performing a JNZ. Others are more comfortable drawing a flowchart for a programming task, and then converting the flowchart into the appropriate instructions. Figure 6.1 shows a



**FIGURE 6.1** A flowchart and its associated assembly language instructions



flowchart used to illustrate the process for searching an array of data values for one that is larger than the others.

Each process or decision block in the flowchart is converted into one or more assembly language instructions. The size of the data values used in the array help determine the operands that must be used in some of the instructions (AH and BH for byte-size values versus AX and BX for word-size values).

Remember that there are no fixed methods for converting pseudocode or flowcharts into machine instructions. However, once you know how the standard control structures are written in assembly language, the conversion from pseudocode or flowchart is straight forward.

### 6.3 WRITING A SOFTWARE TESTER

The programs presented in the remaining sections of this chapter are written as subroutines (or procedures) that must be called to perform their functions. As we saw in the previous section, there may be many subroutines combined in a single application, with each subroutine possibly written by a different person. Thus, each programmer must test (and correct if necessary) the subroutine he or she has written. In this section we see how a new programming module is tested with a *software tester* program. The tester executes the new module with data supplied by the programmer and verifies that the module performs the associated task correctly.

The following procedure was written to solve the quadratic equation  $Y = 5X^2 - 2X + 6$ , where the value for  $X$  is stored in  $AL$  and the result of the equation ( $Y$ ) is returned in  $AX$ :

```
QUAD    PROC    NEAR
        MOV     BL,AL      ;save copy of input value
        MUL     BL         ;compute X^2
        MOV     CX,5
        MUL     CX         ;compute 5X^2
        XCHG    DX,AX      ;save temporary result
        MOV     AL,2
        MUL     BL         ;compute 2X
        SUB     DX,AX      ;compute 5X^2 - 2X
        XCHG    DX,AX      ;get current result into AX
        ADD     AX,6       ;compute 5X^2 - 2X + 6
        RET
QUAD    ENDP
```

The tester program must pass an  $X$  value into the procedure and check the returned value for accuracy. Multiple test cases are preferable because they will show how the routine performs over a range of input values. This requires the programmer to first determine what the correct results should be. Consider these input and output pairs:

| <i>X Input</i> | <i>Y Output</i> |
|----------------|-----------------|
| 0              | 6               |
| 1              | 9               |
| 10             | 486             |
| 100            | 49806           |

The software tester presented here will send each  $X$ -input value to the procedure one at a time and check for a match with the expected  $Y$ -output value each time. If all four tests pass, the tester assumes the new routine is acceptable. If any one test fails, an error message is output.

```
;Program TESTQUAD.ASM: Software tester program for QUAD procedure.
.MODEL SMALL
.DATA
X1    DB    0           ;test case 1
Y1    DW    6
X2    DB    1           ;test case 2
Y2    DW    9
```

```

X3      DB      10              ;test case 3
Y3      DW      486
X4      DB      100            ;test case 4
Y4      DW      49806
PASS    DB      'Procedure passes.',0DH,0AH,'$'
FAIL    DB      'Procedure fails.',0DH,0AH,'$'

        .CODE
        .STARTUP
MOV      AL,X1                ;load first test value
CALL     QUAD                 ;compute result
CMP      AX,Y1                ;look for match
JNZ      BAD
MOV      AL,X2                ;load second test value
CALL     QUAD                 ;compute result
CMP      AX,Y2                ;look for match
JNZ      BAD
MOV      AL,X3                ;load third test value
CALL     QUAD                 ;compute result
CMP      AX,Y3                ;look for match
JNZ      BAD
MOV      AL,X4                ;load fourth test value
CALL     QUAD                 ;compute result
CMP      AX,Y4                ;look for match
JNZ      BAD
LEA      DX,PASS              ;set up pointer to pass message
JMP      SEND                 ;go output message
BAD:     LEA      DX,FAIL       ;set up pointer to fail message
SEND:    MOV      AH,9          ;display string function
INT      21H                  ;DOS call
        .EXIT

QUAD     PROC     NEAR
MOV      BL,AL                ;save copy of input value
MUL      BL                   ;compute X^2
MOV      CX,5
MUL      CX                   ;compute 5X^2
XCHG     DX,AX                ;save temporary result
MOV      AL,2
MUL      BL                   ;compute 2X
SUB      DX,AX                ;compute 5X^2 - 2X
XCHG     DX,AX                ;get current result into Ax
ADD      AX,6                  ;compute 5X^2 - 2X + 6
RET
QUAD     ENDP

        END

```

If more test cases are needed, the test data should be arranged as a data table so that a loop can be used to step through each test case.

Some procedures may require only a single test case to determine whether they function correctly. In any case, if the new procedure should fail, it may be necessary to use DEBUG to single-step through the tester program until the error is found.

Writing tester programs for the routines presented in the remaining sections should be a rewarding programming experience.

## 6.4 DATA GATHERING

When a microprocessor is used in a control application, one of its most important tasks is to gather data from the external process. These data may be composed of inputs from different types of sensors, parallel or serial information transmitted to the system from a separate source, or simply keystrokes from the user's keyboard.

Usually, a section of memory is set aside for the storage of the accumulated data, so the processor can alter or examine it at a later time. The rate at which new data arrive, as in keystrokes from a keyboard, may be very slow, with a new item arriving every few milliseconds or so. When the data rate is slow, the processor will waste valuable execution time waiting for the next new data item. Therefore, an efficient solution is to store the data as they arrive, and process them only when all items have been stored. We will examine two examples of gathering data in this section. The first example deals with keyboard buffering and the second example deals with packing BCD numbers.

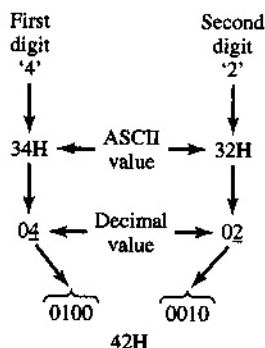
### The Keyboard Buffer

One of the first things anyone involved with computers learns is that interesting things happen when you hit return. All keystrokes up to return must be saved for processing after return is hit. The subroutine presented here, KEYBUFF, is used to store these keystrokes in a buffer until return is hit. The processor will then be free to examine the contents of the keyboard buffer at a later time. KEYBUFF makes use of an INT 21H function, which is used to get a keystroke from the keyboard. The ASCII code for the key is returned by INT 21H in the lower byte of AL. INT 21H takes care of echoing the key back to the user's display. An important point to keep in mind is that INT 21H will not return a value in AL until a key is struck.

The ASCII codes for the keys entered are saved in a buffer called KEYS, which is limited to 128 characters. No code is provided to prevent more than this number of keystrokes. Can you imagine what problems occur when the 129th key is entered?

```
In the current data segment...
KEYS      DB      128 DUP(?)
.
.
.
KEYBUFF   PROC    FAR
          LEA      DI,KEYS      ;DI points to start of buffer
NEXTKEY:  MOV      AH,1
          INT      21H          ;read keystroke and echo to screen
          MOV      [DI],AL      ;save key in buffer
          INC      DI           ;point to next buffer location
          CMP      AL,0DH        ;continue until return is seen
          JNZ      NEXTKEY
          RET
KEYBUFF   ENDP
```

An important feature missing in this example is the use of special codes for editing. No means are provided for editing mistaken keys entered by the user. At the very least, the user should be able to enter a backspace to correct a previous error. You are encouraged to solve this problem, and the other one dealing with limiting the number of keystrokes, yourself.

**FIGURE 6.2** Packing two BCD digits together

### Packing BCD Numbers

Any program that deals with numbers must use one of two approaches to numeric processing. The program must treat the numbers either as ASCII, floating-point, binary values, or BCD values. The use of binary operations provides for large numbers with a small number of bits (integers over 16 million can be represented with only 24 bits) but is limited in accuracy when it comes to dealing with fractions. The use of BCD provides for greater accuracy, but requires software to support the mathematical routines, and this software greatly increases the execution time required to get a result. Even so, BCD numbers have found many uses, especially in smaller computing systems. The example we will study here is used to accept a multidigit BCD number from a keyboard and store it in a buffer called BCDNUM. The trick is to take the ASCII codes that represent the numbers 0 through 9 and convert them into BCD numbers. Two BCD numbers at a time are packed into a byte as shown in Figure 6.2. BCDNUM will be limited to 6 bytes, thus making 12-digit BCD numbers possible. The subroutine PACKBCD will take care of packing the received BCD numbers into bytes and storing them in BCDNUM. No error checking is provided to ensure that no more than 12 digits are entered, or that the user has entered a valid digit. If the number entered is less than 12 digits long, the user hits return to complete the entry. All numbers will be right justified when saved in BCDNUM. This means that numbers less than 12 digits long will be filled with leading zeros.

In current data segment...

```
BCDNUM      DB      6 DUP(?)
```

```
.
```

```
.
```

```
PACKBCD     PROC    FAR
             LEA     DI,BCDNUM           ;point to beginning of buffer
             MOV     CX,6                ;init loop counter
CLEARBUFF:   MOV     BYTE PTR [DI],0    ;clear all bytes in BCDNUM
             INC     DI                  ;with this loop
             LOOP    CLEARBUFF
             DEC     DI                  ;move to end of buffer
GETDIGIT:    MOV     AH,1                ;get a number from the user
             INT     21H
             CMP     AL,0DH              ;done?
             JZ      DONE
             SUB     AL,30H               ;remove ASCII bias
             MOV     BL,AL               ;save first digit
             MOV     AH,1                ;get another number
```

```

                INT     21H
                CMP     AL, 0DH           ;done?
                JZ      SAVEIT
                SDB     AL, 30H          ;remove ASCII bias
                MOV     CL, 4            ;prepare for 4-bit shift
                SHL     BL, CL           ;move BCD digit into upper nybble
                OR      AL, BL           ;pack both digits into AL
                MOV     [DI], AL         ;save digits in buffer
                DEC     DI               ;decrement pointer
                JMP     GETDIGIT
SAVEIT:         MOV     [DI], BL         ;save last digit in buffer
DONE:          RET
PACKBCD        ENDP

```

The loop at the beginning of PACKBCD writes zeros into all 6 bytes of BCDNUM. This is done to automatically place all leading zeros into the buffer before any digits are accepted. Notice also that DI has been advanced to the end of the buffer when the loop has finished. We need DI to start at the end of BCDNUM because we decrement it to store the digits as they are entered. INT 21H is used to get a BCD number from the user (assuming that no invalid digits are entered). Subtracting 30H from the ASCII values returned by INT 21H converts the ASCII character code (35H for "5") into the correct BCD value. The SHL and OR instructions perform the packing of two BCD digits into a single byte. The input number 12345 is stored as 00 00 00 05 34 12. You should experiment with other formats.

---

**Programming Exercise 6.1:** Modify the keyboard buffer routine KEYBUFF so that the user may not enter more than 128 keys. KEYBUFF should automatically return if 128 keys are entered.

---

**Programming Exercise 6.2:** Modify KEYBUFF to allow for two simple editing features. If a backspace key is entered (ASCII code 08H), the last key entered should be deleted. (What problem occurs, though, when backspace is the first key entered?) The second editing feature is used to cancel an entire line. If the user enters a Control-C (ASCII code 03H), the contents of the entire buffer are deleted.

---

**Programming Exercise 6.3:** Modify KEYBUFF to include a count of the number of keys entered, including the final return key. This number should be stored in COUNT on return from KEYBUFF.

---

**Programming Exercise 6.4:** Modify KEYBUFF so that all lowercase letters (a-z) are converted to uppercase letters (A-Z) before being placed in the buffer. All other ASCII codes should remain unchanged.

---

**Programming Exercise 6.5:** Modify KEYBUFF so that the contents of the buffer are displayed (using display string from INT 21H) if Control-R is entered, and the buffer is cleared if Control-C is entered.

---

**Programming Exercise 6.6:** Modify the PACKBCD routine so that a maximum of 12 digits may be entered. PACKBCD should automatically return after processing the 12th digit.

---

**Programming Exercise 6.7:** Modify the PACKBCD routine to scan the buffer after the entire number has been entered and eliminate leading zeros.

---

## 6.5 SEARCHING DATA TABLES

In this section, we will see a few examples of how a block of data may be searched for single or multibyte items. This technique is a valuable tool that has many applications. In a large database, information about many individuals may be stored. Their names, addresses, social security numbers, phone numbers, and many other items of importance may be saved. Finding out if a person is in the database by searching for any of the items just mentioned requires an extensive search of the database. In an operating system, information about users may be stored in a special access table. Their user names, account numbers, and passwords might be included in this table. When a user desires to gain access to the system, his or her entry in the table must be located by account number or name and the password checked and verified. Once on the system, the user will begin entering commands. The commands entered must be checked against an internal list to see if they exist before processing can take place. In a word processing program, a special feature might exist that allows a search of the entire document for a desired string. Every occurrence of this string must be replaced by a second string. For example, the author may notice that every occurrence of "apples" must be changed to "oranges." If only one or two of these strings exist, the author will edit them accordingly. But if "apples" occurs in 50 different places, it becomes very time consuming and inefficient to do this manually. Let us now look at a few examples of how a data table may be searched.

### Searching for a Single Item

The first search technique we will examine involves searching for a single item. This item might be a byte or a word value. The following subroutine searches a 100-element data table for a particular byte value. Upon entry to the subroutine, the byte to be searched for is stored in ITEM. The item may or may not exist within the data table. To account for these two conditions, we will need to return an indication of the result of the search. The carry flag is used to do this. If the search is successful, we will return with the carry flag set. If the search fails, we return with the carry flag cleared. Figure 6.3 shows a flowchart for the search process.

In current data segment...

VALUES DB 100 DUP(?)

ITEM DB ?

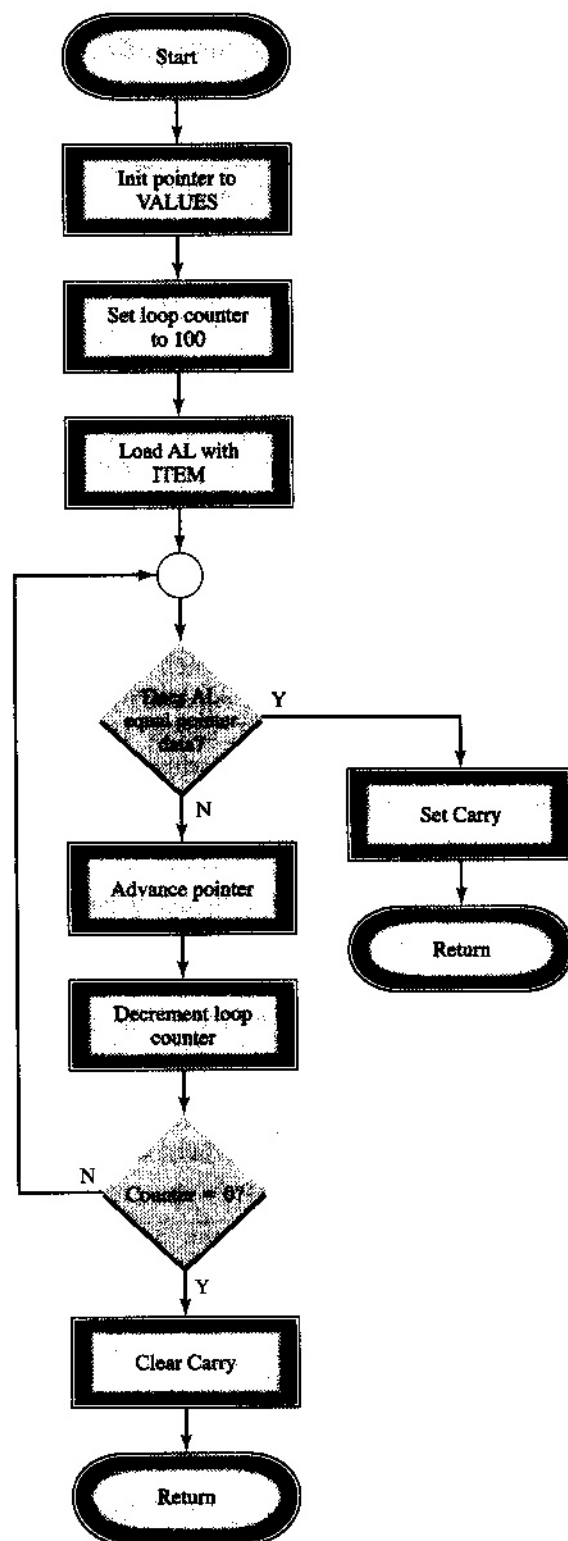
.

.

.

```

FINDBYTE PROC FAR
          LEA SI,VALUES ;init data pointer
          MOV CX,100    ;init loop counter
          MOV AL,ITEM    ;load AL with search item
COMPARE:  CMP AL,[SI]    ;compare item with data in table
          JZ FOUND
          INC SI         ;point to next item
          LOOP COMPARE   ;continue comparisons
  
```

**FIGURE 6.3** Flowchart used to search for a single data item



```

                CLC                ;clear carry flag, search failed
                RET
FOUND:         STC                ;set carry flag, item found
                RET

```

Notice how STC and CLC have been used to directly modify the carry flag, depending on the results of the search. Using the carry flag in this manner allows the programmer to write much simpler code. For example, only two instructions are needed to determine the result of the search:

```

CALL  FINDBYTE
JC    SUCCESS

```

Of course, other techniques may be used to indicate the results. The nice thing about using the flags is that they require no external storage and can be used whenever a binary condition (true/false) is the result.

### Searching for the Highest Integer

When working with data, it often becomes necessary to find the largest value in a given set of numbers. This is useful for finding the range of the given set and also has an application in sorting. MAXVAL is a subroutine that will search an array called NUMBERS for the largest positive byte integer. No negative numbers are allowed at this time. The result of the search is passed back to the caller in the lower byte of BX.

```

In current data segment...
NUMBERS    DB    128 DUP(?)
.
.
.
MAXVAL     PROC    FAR
            LEA    SI, NUMBERS    ;init data pointer
            MOV    BL, 0          ;assume 0 is largest to begin with
            MOV    CX, 128        ;init loop counter
CHECKIT:    CMP    [SI], BL        ;compare current value with new data
            JC     NOCHANGE        ;jump if new value is not larger
            MOV    BL, [SI]        ;load new maximum value
NOCHANGE:   INC    SI             ;point to next byte
            LOOP   CHECKIT        ;continue until all bytes checked
            RET
MAXVAL     ENDP

```

Using JC after the compare operation treats all bytes as unsigned integers. Other forms of the conditional jump will allow signed numbers to be detected as well.

---

**Programming Exercise 6.8:** Modify the FINDBYTE data search subroutine so that the length, in bytes, of the data table is passed via LENGTH. The maximum length of the data is 1,024 items.

---

**Programming Exercise 6.9:** Modify FINDBYTE so that the position of ITEM within VALUES is returned in POSITION, if the search is successful. For example, if ITEM is the first element, POSITION should be 0. If ITEM is the 11th element, POSITION should be 000A.

---

**Programming Exercise 6.10:** Modify the MAXVAL subroutine so that negative numbers (represented in 2's complement notation) may be included in the data.

---

**Programming Exercise 6.11:** Modify the MAXVAL subroutine in two ways: (1) The memory address of the maximum value is returned in DI, and (2) comparison of positive *word* values is performed if AL equals 00 upon entry to MAXVAL, and comparison of byte values is performed otherwise.

---

## 6.6 STRING OPERATIONS

As previously defined, a string is a collection of bytes or words that represent information. For example, the display string function (AH = 9) of DOS's INT 21H requires strings of the form:

```
ANYSTRING DB 'This is a text string.$'
```

in which the end of the string is indicated by the "\$" character. What is required to process a display string? Assume that register DX has been loaded with the starting address of ANYSTRING (via LEA DX,ANYSTRING). The SENDOUT routine shown here reads string characters one at a time and outputs them to the display until the "\$" character is seen.

```
SENDOUT PROC FAR
MOV SI,DX ;use SI as string pointer
GETCHAR: MOV DL,[SI] ;read a string character
CMP DL,'$' ;end of string?
JZ EXIT ;jump if match
MOV AH,2 ;display character function
INT 21H ;DOS call
INC SI ;point to next string character
JMP GETCHAR ;and repeat
EXIT: RET
SENDOUT ENDP
```

One disadvantage of the SENDOUT routine is that it is not possible to output the "\$" character to the display, because it is the end-of-string marker. One way to fix this would be to use a byte value of 0 to terminate the string, as in:

```
NEWSTRING DB 'This string ends differently.',0
```

The CMP instruction must be changed to CMP DL,0 to use this new format.

There are many other uses for text strings. They can specify a DOS file name (and path), as in:

```
RUNFILE DB 'C:\PROGRAMS\RUNME.COM'
```

or a list of abbreviated days of the week:

```
DAYS DB 'MonTueWedThuFriSatSun'
```

In the remainder of this section we will examine a number of techniques that use text strings to perform useful operations.

## Comparing Strings

A very important part of any program that deals with input from a user involves recognizing the input data. Consider the password required by most users of large computing systems. The user must enter a correct password or be denied access to the system. Because the password may be thought of as a string of ASCII characters, some kind of string comparison operation is needed to see if the user's password matches the one expected by the system. The following subroutine compares two strings of 10 characters each, returning with the carry flag set if the strings are exactly the same. If you think of one string as the password entered by the user and the other as the password stored within the system, you will see how they are compared.

```
In current data segment...
STRINGA      DB      'alphabetic'
STRINGB      DB      'alphabet '
.
.
.
CHKSTRING    PROC     FAR
              MOV      SI,0           ;init character pointer
              MOV      CX,10          ;init loop counter
CHECKCHAR:    MOV      AL,STRINGA[SI] ;get character from STRINGA
              CMP      AL,STRINGB[SI] ;compare with STRINGB character
              JNZ      NOMATCH        ;even one difference causes failure
              INC      SI             ;point to next character
              LOOP     CHECKCHAR      ;check all elements
              STC                     ;strings match
              RET
NOMATCH:      CLC                     ;strings are different
              RET
CHKSTRING     ENDP
```

The two strings used in the example are not identical because the last two characters are different.

## A Command Recognizer

Consider a small single-board computer system that allows you to do all of the following:

1. Examine/alter memory (EXAM)
2. Display memory (DUMP)
3. Execute a program (RUN)
4. Terminate program execution (STOP)
5. Load a program into memory (LOAD)

Each of the five example commands has a specific routine address within the memory map of the system. For example, the DUMP command is processed by the code beginning at address 04A2C. The **command recognizer** within the operating system of the small computer must recognize that the user has entered the DUMP command and jump to address 04A2C. This requires that both a string-compare operation and a table lookup be performed. The following routine is one way this may be accomplished:

```
In current data segment...
COMMANDS     DB      'EXAM'
              DB      'DUMP'
```

```

                                DB      'RUN '
                                DB      'STOP'
                                DB      'LOAD'
JUMPTABLE DW      DOEXAM
                                DW      DODUMP
                                DW      DORUN
                                DW      DOSTOP
                                DW      DOLOAD
COMBUFF  DB      4 DUP(?)
.
.
.
RECOGNIZE PROC FAR
                                LEA     BX,COMMANDS      ;point to command table
                                MOV     DI,0              ;init index within JUMPTABLE
                                MOV     CX,5              ;init loop counter
NEXTCOM:  PUSH     CX              ;save loop counter
                                MOV     CX,4              ;prepare for command matching
                                MOV     SI,0
CHKMATCH: MOV     AL,[BX + SI]    ;get a table character
                                CMP     COMBUFF[SI],AL    ;and compare it with command
                                JNZ     NOMATCH
                                INC     SI                ;point to next character
                                LOOP    CHKMATCH          ;continue comparison
                                POP     CX                ;match found, fix stack
                                JMP     JUMPTABLE[DI]     ;jump to command routine
NOMATCH:  POP     CX              ;get loop counter back
                                ADD     BX,4              ;point to next command text
                                ADD     DI,2              ;and next routine address
                                LOOP    NEXTCOM           ;go check next command
                                JMP     COMERROR          ;command not found
RECOGNIZE ENDP

```

The set of valid commands begins at **COMMANDS**. The addresses for each command routine begin at **JUMPTABLE**. The command entered by the user is saved in the 4 bytes beginning at **COMBUFF**. The purpose of **RECOGNIZE** is to compare entries in **COMMANDS** with **COMBUFF**. Every time a match is not found, a pointer (**DI**) is advanced to point to the next routine address in **JUMPTABLE**. When a match is found, **DI** will point to the start of the routine address saved in memory. This routine address is then used by **JMP**. If none of the commands match the user's, a jump is made to **COMERROR** (possibly a routine that will output an error message saying "Illegal command").

## Accessing a Simple Database

In this example we will see how a simple database is defined and accessed through the use of string operations. The database is composed of predefined *records* that contain useful information about employees at a fictitious business. A sample record from the database looks like this:

```
DB      'Jennifer Indigo, CS/AT, 2676, A7',0DH
```

which corresponds to the following record format:

```
<First Name>blank<Last Name>comma<dept>comma<phone>comma<room><cr>
```

where each *<item>* is referred to as a *field* and is terminated by a specific character (such as blank or comma). As you can see, a record is terminated with the code for carriage

return. Any number of records may be strung together to make up the database. In this example, the following database is used:

```
DBASE  DB      'James Antonakos, EET, 2356, B20',0DH
        DB      'Mike Fisher, RWA, 2376, A19',0DH
        DB      'Dave Guza, MPC, 2389, B26', 0DH
        DB      'Jennifer Indigo, CS/AT, 2676, A7',0DH
        DB      'William Robinson, LIS, 2300, J2',0DH
        DB      'Michele Tanner, ILY, 2143, B45',0DH
        DB      0          ;end of database
```

Notice that the length of each record is different. This is due to the varying length of each first name, last name, department, and room number. For simplicity, records are limited to a maximum length of 64 characters, which allows for lengthy names. Even so, as the database shows, the different length of each record prevents us from assuming that the starting character in each field is in a known position. For this reason we must *search* the database for information when we need it. The procedure shown here, LASTNAME, scans the current database record (pointed to by DI) for the individual's last name and displays it. The processor's SCAS instruction is used to find the first blank character in the record, which indicates the end of the first name. SCAS automatically adjusts DI during the scan, so that it points to the first letter of the last name when SCAS completes. A similar search method is used to find the end of the record after the last name is printed out.

Upon entry to LASTNAME, register DI must point to the beginning of a record in the database. Upon exit, DI will point to the beginning of the next record. The last name is displayed on the screen as it is read. It is assumed that the DS and ES registers have been set up accordingly.

```
LASTNAME  PROC      NEAR
            MOV      AL,20H      ;character to find
            MOV      CX,64      ;max length of record
            CLD                ;set auto increment
            REPNZ    SCASB      ;skip to beginning of last name
DISPNAM:   MOV      DL,[DI]     ;load string character
            CMP      DL,', '    ;past end of last name?
            JZ       FIXEND     ;jump if so
            MOV      AH,2      ;otherwise, output character to display
            INT      21H       ;DOS call
            INC      DI        ;advance to next character
            JMP      DISPNAM    ;and repeat
FIXEND:    MOV      AL,0DH      ;character to find
            MOV      CX,64      ;max length of record
            REPNZ    SCASB      ;skip to end of record
            RET
LASTNAME  ENDP
```

If LASTNAME is called repeatedly, each last name in the database will be found. A tester program to do this, and to format the output with carriage return and line feed codes, might display these results:

```
Antonakos
Fisher
Guza
Indigo
Robinson
Tanner
```

Similar routines can be written to access and display any group of items from each record. These routines can also be used to create other databases. For example, LASTNAME can be modified to copy just the last name into a new area of memory where a database of last names is being constructed. In this case, other string instructions, such as MOVS, might be useful.

---

**Programming Exercise 6.12:** Modify the SENDOUT routine so that a carriage return and a line feed are output when the "\$" character is encountered at the end of the string.

---

**Programming Exercise 6.13:** Rewrite the CHKSTRING procedure using the CMPSB instruction. Remember that both strings are located in the current data segment and adjust the ES register accordingly.

---

**Programming Exercise 6.14:** The CHECKSTR subroutine is limited for two reasons. First, the starting addresses of the two strings are set when the subroutine is entered. Second, the length of the two strings is fixed at 10 characters apiece. Modify CHECKSTR so that registers SI and DI are loaded from the addresses stored in locations STRING1 and STRING2 and the string length is loaded from LENGTH.

---

**Programming Exercise 6.15:** Using a modified version of CHECKSTR, write a routine that will count the number of occurrences of the word "the" in the block of text. The text block begins at address 3000 and ends at address 37FF.

---

**Programming Exercise 6.16:** Rewrite the RECOGNIZE procedure using the CMPSW instruction. Remember that both strings are located in the current data segment and adjust the ES register accordingly.

---

**Programming Exercise 6.17:** The command recognizer RECOGNIZE works only with uppercase commands. Rewrite the code so that uppercase and lowercase commands may be recognized. For example, "DUMP" and "dump" should be identical in comparison.

---

**Programming Exercise 6.18:** Write a command recognizer that will recognize single-letter commands. The commands may be either uppercase or lowercase, and have the following addresses associated with them:

- A: 20BE
  - B: 3000
  - C: 589C
  - D: 2900
- 

**Programming Exercise 6.19:** Modify the LASTNAME procedure so that the last name is copied into a buffer, defined as:

```
LNAME    DB    33 DUP(?)
```

The last name written into the LNAME buffer should be terminated with a "\$" character.

---

**Programming Exercise 6.20:** Write a routine called PHONELIST that scans the current database record for the first name and phone fields and displays them on the screen, as in:

```
James    2356
Mike     2376
etc.
```

Be sure that each phone number begins in the same column.

---

## 6.7 SORTING

It is often necessary to sort a group of data items into ascending (increasing) or descending order. On average, the search time for a sorted list of numbers is smaller than that of an unsorted list. Many different sorting algorithms exist, with some more efficient than others. The sorting algorithm covered here is called a **bubble sort**. A bubble sort consists of many passes over the elements being sorted, with comparisons and swaps of numbers being made during each pass. A short example should serve to introduce you to the technique of the bubble sort. Consider this group of numbers:

7 10 6 3 9

It is necessary to perform only four comparisons to determine the highest number in the group. We will repeatedly compare one element in the group with the next element, starting with the first. If the second element is smaller than the first, the two numbers will be swapped. This technique is illustrated in Figure 6.4. By this method, we guarantee that after four comparisons the largest number is at the end of the array. Check this for yourself. Initially, 7 and 10 are compared and not swapped. Then 10 and 6 are compared and swapped because 10 is greater than 6. The new array looks like this:

7 6 10 3 9

Next, 10 and 3 are compared and swapped. Then 10 and 9 are compared and swapped. At the end of the first pass, the array is:

7 6 3 9 10

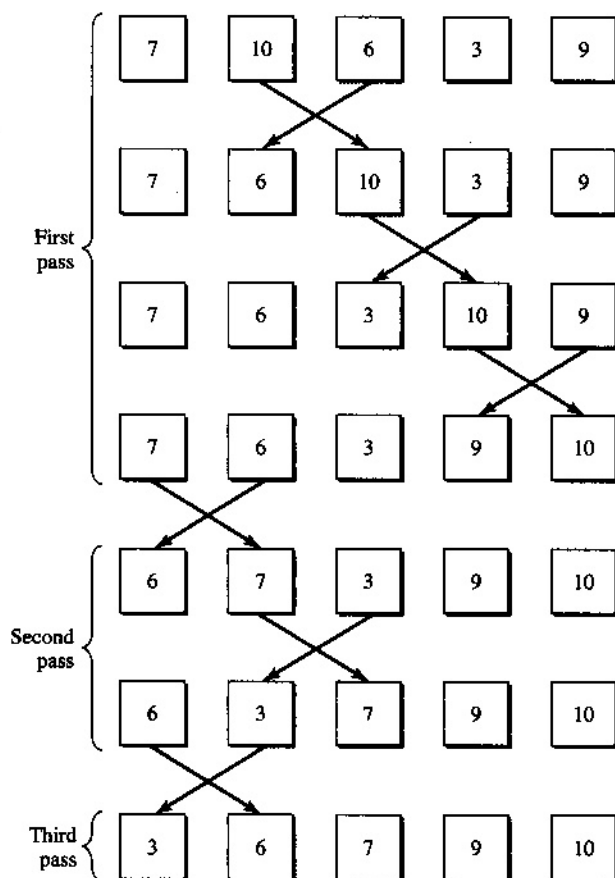
It is not necessary now to ever compare any of the elements in the array with the last one, because we know it to be the largest. The next pass will only compare the first four numbers, giving this array at the end of the second pass:

6 3 7 9 10

The third pass will produce:

3 6 7 9 10

and you may notice now that the array is sorted. However, this is due to the original arrangement of the numbers; for completeness, a final pass must be performed on the first two numbers. Note that the five numbers being sorted required four passes. In general,  $N$  numbers will require  $N - 1$  passes. The subroutine SORT presented here implements a bubble sort. DX is used as the pass counter, registers AL and BL are used for swapping elements, and CX is used as a loop counter. The number of elements to be sorted is saved as a word count in NVAL. The appropriately sized DUP statement is needed for VALUES,

**FIGURE 6.4** Bubble sorting a set of 5 integers

with only sixteen locations reserved in this example. Also, only positive integers may be sorted (because of the use of JNC in the comparison).

In current data segment...

VALUES DB 16 DUP(?)

NVALS DW ?

.

.

.

```

SORT      PROC    FAR
           MOV     DX,NVALS                ;get number of data items
           DEC     DX                      ;subtract 1 to start
DOPASS:    MOV     CX,DX                   ;init loop counter
           MOV     SI,0                    ;init data pointer
CHECK:     MOV     AL,VALUES[SI]           ;get first element
           CMP     VALUES[SI + 1],AL      ;compare with second element
           JNC     NOSWAP
           MOV     BL,VALUES[SI + 1]       ;swap elements
           MOV     VALUES[SI + 1],AL
           MOV     VALUES[SI],BL
NOSWAP:    INC     SI                      ;point to next element
           LOOP    CHECK                   ;continue with pass
           DEC     DX                      ;decrement pass counter
           JNZ     DOPASS                  ;until finished
           RET
SORT      ENDP

```



By advancing SI in steps of one, memory references VALUES[SI] and VALUES[SI+1] always access the next two elements in the VALUES array. When it is necessary to swap them, temporary variable BL is used to hold the contents of the second location while it is being replaced by the contents of the first location. The stack may be used for this purpose also, but at the expense of additional execution time. The use of different conditional jump instructions will allow for negative numbers to be sorted as well.

---

**Programming Exercise 6.21:** Modify the SORT routine so that signed and unsigned numbers are allowed.

---

**Programming Exercise 6.22:** Modify the SORT routine so that it exits as soon as an entire pass fails to produce a single swap.

---

**Programming Exercise 6.23:** Modify the SORT routine so that it sorts only array elements that lie between the addresses contained within SI (starting address) and DI (ending address). For example, if SI contains 03A7 and DI contains 03B2, only the twelve numbers in the address range 03A7 to 03B2 are sorted.

---

## 6.8 COMPUTATIONAL ROUTINES

This section covers examples of how the 80x86 performs standard mathematical functions. Because the processor has specific instructions for both binary and BCD operations, we will examine sample routines written around those instructions. Math processing is a major part of most high-level languages and the backbone of specialized application programs, such as spreadsheets and statistical analysis packages. Most processors, however, are limited in their ability to perform complicated math. When complex functions such as SIN(X) or LOG(Y) are needed, the programmer is faced with a very difficult task of writing the code to support them. Even after the code is written and judged to be correct, it will most likely be very lengthy and slow in execution speed. For this reason, some systems are designed with math coprocessor chips. These chips are actually microprocessors themselves whose instruction sets contain only mathematical instructions. Adding a coprocessor eliminates the need to write code to perform the math function. SIN(X) is now an instruction executed by the coprocessor. The main CPU simply reads the result from the coprocessor. The coprocessor available for the 80x86 is the 80x87 floating-point coprocessor, which we will examine in Chapter 11.

The examples we will see in this section deal only with addition, subtraction, multiplication, and division. We will, however, also look at a few ways these simple operations can be applied to simulate more complex ones.

### Binary Addition

Binary addition is accomplished with ADD and ADC. Both perform addition on registers and/or memory locations. The example presented here is used to find the signed sum of a set of data. The data consists of signed 8-bit numbers. Because it is possible for the sum to exceed 127, we use 16 bits to represent the result.

In current data segment...

```
SCORES DB 200 DUP(?)
SUM DW ?
```

```

TOTAL PROC FAR
        LEA SI,SCORES ;init pointer to data
        MOV CX,200    ;init loop counter
        MOV BX,0      ;clear result
ADDEM:  MOV AL,[SI]    ;load AL with value
        CBW           ;sign extend into 16 bits
        ADD BX,AX      ;add new value to result
        INC SI         ;point to next data item
        LOOP ADDEM     ;do all values
        MOV SUM,BX     ;save result in memory
        RET
TOTAL ENDP
```

Even though the data consists of signed 8-bit numbers, we can perform 16-bit additions if we first use CBW to extend the signs of the input numbers (from 8 to 16 bits).

## Binary Subtraction

Binary subtraction is implemented by SUB and SBB. Both instructions work with memory locations and/or registers. The example presented here shows how two blocks of memory may be subtracted from each other. One application in which this technique is useful involves digitally encoded waveforms. Suppose that two analog signals, sampled at an identical rate, must be compared. If the difference is computed by subtracting the binary representation of each waveform and the resultant waveform is displayed by sending the new data to a digital-to-analog converter, we will see a straight line at the output if the waveforms are identical. WAVE1 and WAVE2 are labels associated with the 2K word blocks of memory that must be subtracted. Because of the addressing mode used, the resulting data will overwrite the data saved in WAVE2's area.

In current data segment...

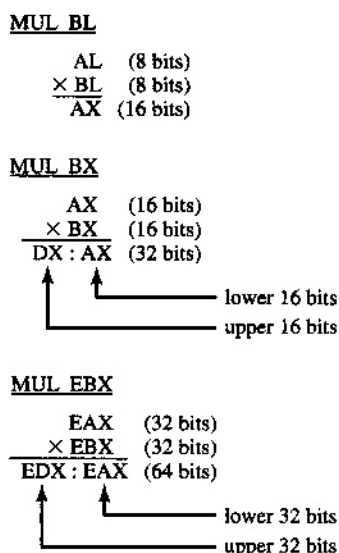
```
WAVE1 DW 2048 DUP(?)
WAVE2 DW 2048 DUP(?)
```

```

SUBWAVE PROC FAR
        LEA SI,WAVE1 ;init pointer to beginning of WAVE1
        LEA DI,WAVE2 ;init pointer for WAVE2
        MOV CX,2048  ;init loop counter
SUBEM:  MOV AX,[SI]   ;get sample from WAVE1
        SUB [DI],AX   ;subtract and replace WAVE2 sample
        ADD SI,2      ;advance WAVE1 pointer
        ADD DI,2      ;advance WAVE2 pointer
        LOOP SUBEM    ;do all samples
        RET
SUBWAVE ENDP
```

## Binary Multiplication

Two instructions are available for performing binary multiplication. MUL (unsigned multiply) is used to multiply 8-, 16-, or 32-bit operands, one of them contained in the accumulator.

**FIGURE 6.5** 8-, 16-, and 32-bit multiplication

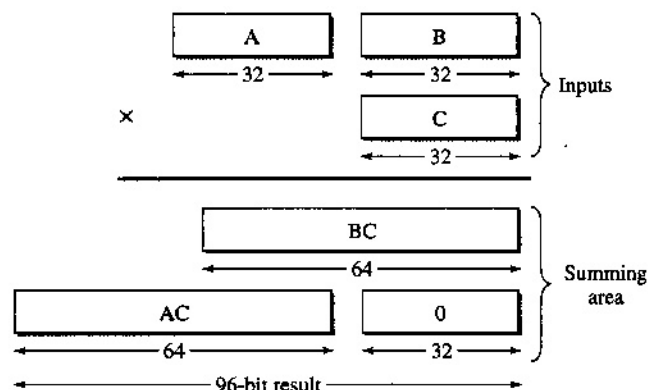
IMUL (signed multiply) generates a signed result using signed operands of 8, 16, or 32 bits each. In both cases, the results are stored in AX/EAX or AX/EAX and DX/EDX, as illustrated in Figure 6.5. When 64-bit precision is not enough, we must turn to an alternate method to perform the math.

The example presented here is used to perform 64- by 32-bit multiplication on unsigned integers. The 96-bit result represents a significant increase over the 64 bits the processor is limited to. The method used to perform the multiplication is diagrammed in Figure 6.6. The 64-bit operand is represented by two 32-bit halves, A and B. The 32-bit operand is represented by C. Multiplying B by C will yield a 64-bit result. The same is true for A and C, except that A is effectively shifted 32 bits to the left, making its actual value much larger. To accommodate this, 32 zeros are placed into the summing area in such a way that they shift the result of A times C the same number of positions to the left. This is analogous to writing down a 0 during decimal multiplication by hand. The lower 32 bits of the result are the same as the lower 32 bits of the BC product. The middle 32 bits of the result are found by adding the upper 32 bits of the BC product to the lower 32 bits of the AC product. The upper 32 bits of the result equal the upper 32 bits of the AC product, plus any carry out of the middle 32 bits. In the following routine, registers EDX and EAX contain the 64-bit value we know as AB (with EDX holding the upper 32 bits). Register EBX contains the 32-bit multiplier C.

In current data segment...

```
LOWER    DD    ?
MIDDLE   DD    ?
UPPER    DD    ?
```

```
MULTIPLY  PROC    FAR
          PUSH    EDX                ;save a copy of EDX (A) on stack
          MUL     EBX                ;do B times C
          MOV     LOWER,EAX          ;save partial results
          MOV     MIDDLE,EDX
          POP     EAX                ;pop stack (A) into EAX
          MUL     EBX                ;do A times C
          ADD     MIDDLE,EAX         ;generate middle 32-bits of result
          ADC     EDX,0              ;increment EDX if carry present
```

**FIGURE 6.6** Diagram of 64- by 32-bit multiplication

```

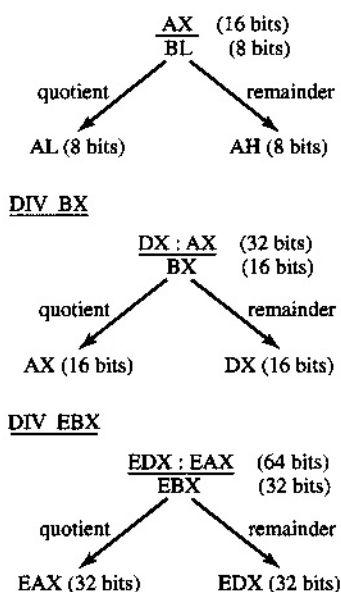
MOV     UPPER,EDX    ;save upper 32-bits of result
RET
MULTIPLY    ENDP

```

It should be possible to relate the code of this example to Figure 6.6. Generating the individual AC and BC products is easily done via MUL. Adding the upper 32 bits of the BC product to the lower 32 bits of the AC product is accomplished by using the ADD instruction. Any overflow out of the middle 32 bits will be placed into the carry flag. This carry is then added to the upper 32 bits of the AC product to complete the operation.

## Binary Division

The 80x86 supports binary division with its DIV and IDIV (unsigned and signed division) instructions. Both instructions can divide a 64-bit quantity by a 32-bit quantity as well as other operand sizes, as shown in Figure 6.7. The 64-bit result is composed of a 32-bit

**FIGURE 6.7** 8-, 16-, and 32-bit DIV BL division

quotient and a 32-bit remainder. When the quotient is too large for the destination register, a type-0 interrupt will be generated on completion of the instruction. Many applications exist for the division operation. It can be used to find averages, probabilities, factors, and many other items that are useful when we are working with sets of data. The following subroutine is used to find a factor of a given number when supplied with another factor. For example, FACTOR will return 50 as a factor, when 6 and 300 are supplied as input (because 300 divided by 6 equals 50 exactly). FACTOR will return 0 if no factor exists (for example, 300 divided by 7 gives 42.857143, which is not an integer; thus, the two numbers cannot be factors).

In current data segment...

```

NUMBER      DQ      ?           ;64-bit input number
FACTOR1     DD      ?           ;32-bit input factor
FACTOR2     DD      ?           ;32-bit output factor
.
.
.
FACTOR      PROC     FAR
             LEA      ESI,NUMBER   ;point to input number
             MOV      EAX,[ESI]    ;load input number into EDX:EAX
             MOV      EDX,[ESI+4]
             DIV      FACTOR1      ;divide by input factor
             CMP      EDX,0        ;was division even?
             JNZ      NOFACTOR
             MOV      FACTOR2,EAX  ;save output factor
             JMP      EXIT
NOFACTOR:    MOV      FACTOR2,0    ;clear output factor
EXIT:        RET
FACTOR      ENDP

```

Because the remainder appears in EDX, we examine it for 00000000 to see if the division was even.

## BCD Addition

In the binary number system, we use 8 bits to represent integer numbers in the range 0 to 255 (00 to FF hexadecimal). When the same 8 bits are used to store a binary coded decimal (BCD) number, the range changes. Integers from 0 to 99 may now be represented, with the 10s and 1s digits using 4 bits each. If we expand this reasoning to 16 bits, we get a 0 to 65,535 binary integer range, and a 0 to 9999 BCD range. Notice that the binary range has increased significantly. This is always the case and represents one of the major differences between binary and BCD numbers. Even so, we use BCD to solve a nasty problem encountered when we try to represent some numbers using binary. Consider the fractional value 0.7. It is impossible to exactly represent this number using a binary string. We end up with 0.101100110011. . . . The last four bits (0011) keep repeating. So, we can get very close to 0.7 this way (0.699999 . . .), but never actually get 0.7. When we use this binary representation in a calculation, we will automatically generate a roundoff error. The purpose of BCD is to eliminate the roundoff error (at the cost of a slower computational routine). This feature of BCD makes it attractive for applications involving decimal numbers, such as calculators, cash registers, and automotive electronics.

For the purposes of this discussion, we will use a BCD representation that consists of 4 bytes stored in consecutive memory locations. The first byte is the most significant byte.

The fourth byte is least significant. All BCD numbers stored this way (0 to 99999999) will be right justified. Examine the following two numbers and their memory representations to see what is meant by right justification:

```
34298: 00 03 42 98
7571364: 07 57 13 64
```

We can increase the range of numbers by adding more bytes of storage per number. Each new byte gives two additional BCD digits. Furthermore, we could also add an additional byte to store the exponent of the number. A single byte could represent exponents from 127 to -128 if we used signed binary numbers. Standards exist that define the format of a BCD number (and of binary numbers as well, for use with floating-point units), but we will not cover them at this time.

The example presented here shows how two BCD numbers (each stored in memory at NUMA and NUMB) can be added together. The DAA (decimal adjust for addition) instruction is used together with ADC to perform the BCD addition. ADC will add 2 bytes together, each containing two BCD digits. The result will be corrected by DAA, with the carry flag containing any carry out of the most significant digit. For example, if 37 and 85 are added, the carry will equal 1 and the result operand will contain 22. Because we have defined the 4-byte storage array for a BCD number to be right justified, it is necessary to begin adding with the least significant byte in the array. The result is stored in NUMB, overwriting the BCD number already saved.

In current data segment...

```
NUMA    DB    4 DUP(?)
NUMB    DB    4 DUP(?)
```

```
ADDBCD  PROC  FAR
        MOV   SI,3           ;init pointer to LSB
        MOV   CX,4           ;init loop counter
        CLC                ;clear carry to start
DECIADD: MOV   AL,NUMA[SI]    ;get first BCD number
        ADC   AL,NUMB[SI]    ;add second BCD number
        DAA                ;correct result into BCD
        MOV   NUMB[SI],AL    ;save result
        DEC   SI             ;point to next pair to digits
        LOOP  DECIADD        ;do all digits
        RET
ADDBCD  ENDP
```

Upon return from the subroutine, the carry flag will contain any carry out of the MSB.

## BCD Subtraction

BCD subtraction is implemented in much the same way as BCD addition, and the subroutine presented here uses the same 4-byte BCD number definition covered in the previous section. The difference in this routine is that the addresses of the two BCD numbers are assumed to be contained in registers SI and DI upon entry. Assuming that SI points to NUMA and DI to NUMB, two different subtractions are possible.

In current data segment...

```
NUMA      DB      4 DUP (?)
NUMB      DB      4 DUP (?)
.
```

```
AMINUSB   PROC     FAR
          XCHG     SI,DI          ;swap pointers
BMINUSA:  MOV      CX,4          ;init loop counter
          ADD      SI,3          ;adjust pointer to end of BCD number
          ADD      DI,3
          CLC                     ;clear carry flag to start
DECISUB:  MOV      AL,[DI]       ;get first number
          SBB      AL,[SI]       ;subtract second number
          DAS                     ;adjust result into BCD
          MOV      [DI],AL       ;store result
          DEC      SI           ;adjust pointers
          DEC      DI
          LOOP     DECISUB       ;do all digits
          RET
AMINUSB   ENDP
```

Upon return, the carry flag will indicate any borrow from the MSB. If the carry flag is set upon return, the result of the subtraction is negative. The result will replace the contents of NUMB when BMINUSA is the entry point to the subroutine. Entering at AMINUSB will cause the result to replace NUMA.

## BCD Multiplication

Because BCD multiplication is not directly implemented on the 80x86, there are at least two ways it can be simulated. One method is to convert both BCD numbers into their binary equivalents and then use IMUL to find the result. Of course, the binary result will have to be converted back into BCD. A procedure to accomplish this method is as follows:

```
BCDMUL1   PROC     FAR
          LEA      SI,NUM1       ;point to first BCD number
          CALL     TOBINARY      ;convert into binary
          MOV      BX,AX         ;save result here
          LEA      SI,NUM2       ;point to second BCD number
          CALL     TOBINARY      ;and convert it to binary
          IMUL     BX            ;find the signed product
          CALL     TOBCD         ;convert result into BCD
          RET
BCDMUL1   ENDP
```

The two conversion routines, TOBINARY and TOBCD, operate as follows: TOBINARY converts the BCD number pointed to by SI into a signed binary number and returns it in AX. TOBCD converts the signed binary number in DX:AX into a BCD number and saves it in memory at BCDNUM.

A second approach is to do all the math in BCD. This will require a number of repetitive additions to generate the answer. The need for this looping will, unfortunately, slow down the execution speed. This disadvantage is overcome by the ability to multiply large BCD numbers. BCDMUL2 will multiply two 2-digit BCD numbers stored in the lower

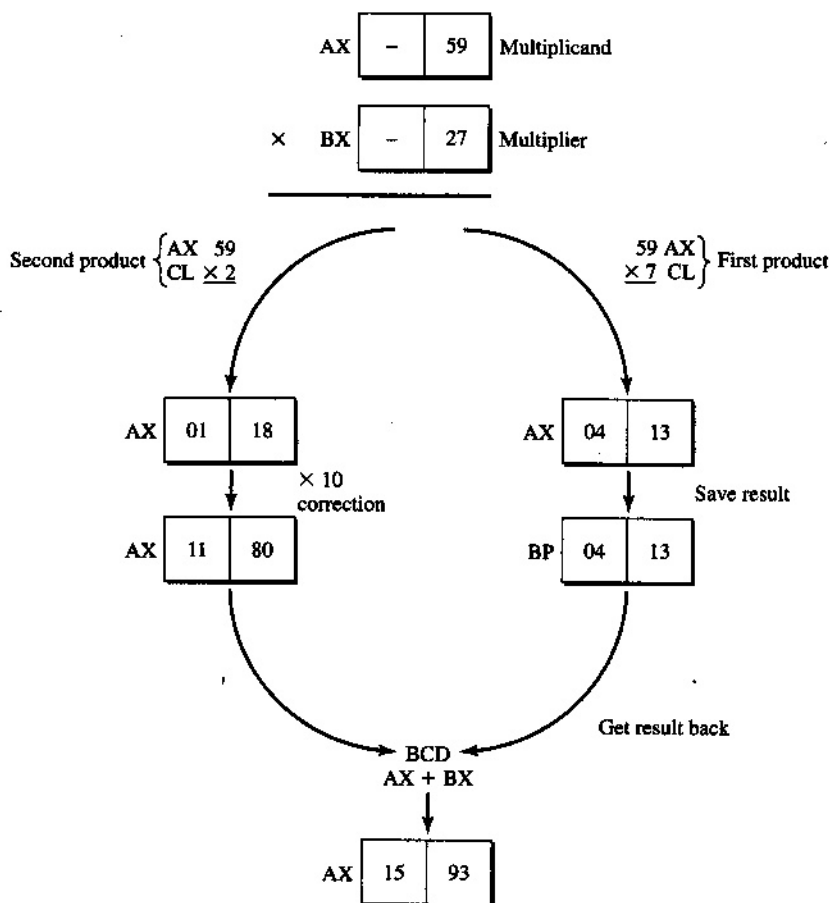


FIGURE 6.8 Multiplying two BCD numbers

byte of registers AX and BX. The BCD result will be placed in AX. Further programming easily extends the input numbers into additional digits.

The multiplication performed by BCDMUL2 is detailed in Figure 6.8. As the figure shows, the product resulting from the 10s digits of the multiplier is shifted left one BCD digit, to simulate the result of multiplying by 10.

```
BCDMUL2  PROC    FAR
MOV      AH,0    ;fix AX
PUSH     AX      ;save copy of AL on stack
MOV      CL,BL   ;find the 1s product
AND      CL,0FH
CALL     FAR PTR ALTIMESCL
MOV      BP,AX   ;save 1s product for later
POP      AX      ;get AL back
MOV      CL,4    ;prepare for shift
SHR      BL,CL   ;move upper BCD digit into lower half
MOV      CL,BL   ;find the 10s product
CALL     FAR PTR ALTIMESCL
MOV      CL,4    ;prepare for shift
SHL      AX,CL   ;multiply BCD value by 10
MOV      BX,BP   ;get 1s product back
```



```

                ADD     AL,BL      ;form lower half of result
                DAA
                ADC     AH,BH      ;form upper half of result
                XCHG    AL,AH      ;DAA only works on AL
                DAA
                XCHG    AL,AH
                RET
BCDMUL2        ENDP
ALTIMESCL      PROC    FAR
                MOV     CH,0       ;fix loop counter
                MOV     DL,AL      ;save AL
                MOV     AX,0       ;clear result
AGAIN:         ADD     AL,DL      ;perform repetitive additions
                DAA          ;until product is found
                ADC     AH,0
                LOOP    AGAIN
                RET
ALTIMESCL      ENDP

```

Notice how a subroutine is used to make the overall process easier to code and read. Although we have not shown it in this example or in previous examples, it is assumed that a valid stack pointer has been assigned to save the subroutine return address and other information. Note also that neither BCDMUL1 nor BCDMUL2 checks for multiplication by 0.

## BCD Division

All of the BCD operations we have examined so far have ignored treatment of exponents. A collection of subroutines that perform BCD math must have methods of dealing with exponents or be very limited in its applications. As previously mentioned, we can add a single byte to our BCD format to include exponents in the calculations. A single byte gives a signed integer range from -128 to 127. This slightly changes the format of the BCD numbers represented and requires **normalization** of the numbers before conversion. Normalization is necessary because we have no way of storing a decimal point within the binary data we use to represent a number. Through normalization, we end up with a standard representation by altering the mantissa and adjusting the exponent accordingly. For example, 576.4 and 5.764E2 are equal, as are 23497.28 and 2.349728E4. In these examples, both numbers have been normalized so that the first digit of the mantissa is always between 1 and 9. This method works for fractional numbers as well. Here, we have 0.0035 equaling 3.5E-3. The addition of an exponent byte to our format, together with the new technique of normalization, will require that we now left justify our BCD numbers. Representing these numbers in our standard format gives

|           |    |    |    |    |    |
|-----------|----|----|----|----|----|
| 576.4:    | 02 | 57 | 64 | 00 | 00 |
| 23497.28: | 04 | 23 | 49 | 72 | 80 |
| 0.0035:   | FD | 35 | 00 | 00 | 00 |

where the first byte is used to represent the signed binary exponent. Notice the 2's complement representation of the exponent -3 in the third set of data bytes.

Adding exponent capability to our BCD format complicates the routines we have already seen. The addition routine (as well as subtraction) will give valid results only when we are adding two numbers whose exponents are equal. Because this is rarely the case, we need to adjust the exponent of one number before doing the addition. For instance, if we

wish to add 5027 and 394, we must first normalize both numbers:

5027: 03 50 27 00 00

394: 02 39 40 00 00

Because the exponents are different, we have to adjust one of the numbers to correctly add them. If we adjust the number with the higher exponent, we may lose accuracy in our answer. It is much safer to adjust the smaller number. This gives us

5027: 03 50 27 00 00

394: 03 03 94 00 00

It is clear now that BCD addition of the 4 trailing bytes will give the correct answer. Notice that we have not changed the value of the second number, only its representation.

BCD multiplication and division also require the use of exponents for best results. Unfortunately, it is not a simple matter of adding exponents for multiplication and subtracting them for division. Special rules are invoked when we multiply or divide two negative numbers. In any case, we must take all rules into account when writing a routine that will handle exponents.

The BCD division routine presented here keeps track of exponents during its calculations. The subroutine NUMALIGN adjusts the dividend so that it is always 1 to 9 times greater than the divisor. NUMALIGN modifies the exponent of the dividend as well. Subroutine MSUB performs multiple subtractions. The number of times (0 to 9) the divisor is subtracted from the dividend is returned in the lower 4 bits of AL. Both routines use SI and DI as pointers to the memory locations containing the BCD representations of the dividend and divisor. Register BX accumulates the individual results from MSUB into a 4-digit BCD result. The exponent is generated by the EXPONENT subroutine, which uses the initial exponent values plus the results of ALIGN to calculate the final exponent, which is returned in the lower byte of AX.

```

In current data segment...
DIVIDEND DB 5 DUP (?) ;reserve 5 bytes (one for exponent)
DIVISOR DB 5 DUP (?)
.
.
.
BCDDIV PROC FAR
    LEA SI, DIVIDEND ;init pointer to dividend
    LEA DI, DIVISOR ;init pointer to divisor
    MOV AL, 0 ;clear exponent accumulator
    MOV CX, 4 ;init loop counter
DIVIDE: CALL FAR PTR NUMALIGN ;align numbers
        CALL FAR PTR MSUB ;perform multiple subtractions
        PUSH CX ;save loop counter
        MOV CL, 4
        SHL BX, CL ;shift result one digit left
        POP CX ;get loop counter back
        AND AL, 0FH ;mask out result from MSUB
        OR BL, AL ;save result in BL
        LOOP DIVIDE ;continue for more precision
        CALL FAR PTR EXPONENT ;generate final exponent
        RET
BCDDIV ENDP

```

BCDDIV does not check for division by 0, but this test could be added easily with a few instructions.

## Deriving Other Mathematical Functions

Once subroutines exist for performing the basic mathematical functions (addition, subtraction, multiplication, and division), these subroutines may be used to derive more complex functions. The examples presented here show how existing routines can be combined to simulate higher level operations. All of the examples to be presented assume that the following multiprecision subroutines exist:

| <i>Routine</i> | <i>Operation</i>          |
|----------------|---------------------------|
| ADD            | $(BP) = (SI) + (DI)$      |
| SUBTRACT       | $(BP) = (SI) - (DI)$      |
| MULTIPLY       | $(BP) = (SI) \times (DI)$ |
| DIVIDE         | $(BP) = (SI) / (DI)$      |

In all cases, SI and DI point to the two input numbers upon entry to the subroutine and BP points to the result! Thus, (SI) means the number pointed to by SI, not the contents of SI. By defining the routines in this way, we can avoid discussion about whether the numbers are binary or BCD.

The first routine examined is used to raise a number to a specified power (for example, 5 raised to the 3rd power is 125). This routine uses the binary number in AL as the power. The number raised to this power is pointed to by SI. The final result is pointed to by BP.

```

POWER    PROC    FAR
          CALL    FAR PTR COPY      ;make a copy of the input number
MAKEPOW: CALL    FAR PTR MULTIPLY   ;compute next power result
          XCHG    BP,SI              ;use result as next input
          DEC     AL                  ;continue until done
          JNZ     MAKEPOW
          XCHG    BP,SI              ;final result is pointed to by BP
          RET
POWER    ENDP

```

POWER is written such that the power must be 2 or more. Negative powers and powers equal to 0 or 1 are not implemented in this routine (they are left as an exercise). COPY is a subroutine that makes a copy of the input number pointed to by SI. The copied number is pointed to by DI.

The next routine is used to generate factorials. A factorial of a number (for example, 5! or 10! or 37!) is found by multiplying all integers up to and including the input number. For instance, 5! equals  $1 \times 2 \times 3 \times 4 \times 5$ . This results in 5! equaling 120. Do a few factorial calculations yourself, and you will see that the result gets very large, very quickly! FACTORIAL will compute the factorial of the integer value stored in AL. The result is pointed to by BP.

```

FACTORIAL PROC    FAR
          LEA     SI,ONE              ;init sequence counter
          LEA     DI,ONE              ;init first multiplier
NEXTNUM: CALL    FAR PTR MULTIPLY   ;compute partial factorial
          XCHG    BP,DI              ;use result as next input
          CALL    FAR PTR INCREMENT ;increment sequence counter
          DEC     AL                  ;continue until done
          JNZ     NEXTNUM
          XCHG    BP,DI              ;result pointed to by BP
          RET
FACTORIAL ENDP

```

**FIGURE 6.9** Finding square roots by Newton's iteration

|                                                                                       |          |
|---------------------------------------------------------------------------------------|----------|
| $\text{Estimate} = \frac{\frac{\text{number}}{\text{estimate}} + \text{estimate}}{2}$ |          |
| Example: Find square root of 42<br>Initial Estimate: 21                               |          |
| Number of iterations                                                                  | Estimate |
| 0                                                                                     | 21       |
| 1                                                                                     | 11.5     |
| 2                                                                                     | 7.57608  |
| 3                                                                                     | 6.55992  |
| 4                                                                                     | 6.48121  |
| 5                                                                                     | 6.4807   |
| $(6.4807)^2 = 41.999$                                                                 |          |

INCREMENT is a subroutine that performs a specific task: add 1 to the number pointed to by SI. We use INCREMENT to generate the sequence of integers that get multiplied together. The symbol ONE refers to a predefined storage area in memory that contains the value 1 in standard format.

The next routine, ROOT, computes square roots. The formula, and an example of how it works, is presented in Figure 6.9 and is called an **iterative formula**. This means that we must run through the formula a number of times before getting the desired result. Notice in Figure 6.9 how each new application of Newton's square root formula brings the estimate of the answer closer to the correct value. After applying the formula only five times, we have a result that comes very close to the square root. A few more iterations will increase the accuracy of the result even more. Fewer iterations are needed when the initial estimate is close to the desired value. For instance, if the original estimate used in Figure 6.9 was 7 instead of 21, fewer iterations would have been needed to get to 6.4807. The routine presented here implements the formula of Figure 6.9:

In current data segment...

```
NUMBER    DB    5 DUP(?)
ESTIMATE  DB    5 DUP(?)
```

```
.
```

```
.
```

```
.
```

```
ROOT      PROC    FAR
          LEA     SI,NUMBER          ;point to input number
          LEA     DI,TWO             ;predefined constant 2
          CALL    FAR PTR DIVIDE     ;calculate original estimate
          LEA     BX,ESTIMATE        ;save estimate
          CALL    FAR PTR SAVE
          MOV     CX,10              ;prepare for 10 iterations
ITERATE:  LEA     SI,NUMBER
          LEA     DI,ESTIMATE
          CALL    FAR PTR DIVIDE     ;number / estimate
          XCHG    BP,SI              ;use result in following addition
          CALL    FAR PTR ADDER      ;(number / estimate) + estimate
          XCHG    BP,SI              ;use result in following division
          LEA     DI,TWO
          CALL    FAR PTR DIVIDE     ;entire formula implemented now
          LEA     BX,ESTIMATE        ;save new estimate
          CALL    FAR PTR SAVE
          LOOP    ITERATE
          RET
ROOT      ENDP
```

**FIGURE 6.10** Generation of  $e^x$  by infinite series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$= \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

$$= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

Example: Find  $e^1$

| Number of terms | Result  |
|-----------------|---------|
| 1               | 1       |
| 2               | 2       |
| 3               | 2.5     |
| 4               | 2.66666 |
| 5               | 2.70833 |
| 6               | 2.71666 |
| 7               | 2.71805 |

( $e^1 = 2.7182818$ )

The subroutine **SAVE** is used to make a copy of the number pointed to by **BP**. The copy is stored in memory starting at the location pointed to by **BX**. The **XCHG** instruction is used to swap pointers, thus making the results of **ADDER** and **DIVIDE** available for the next operation.

The last example we will examine is used to compute powers of base  $e$ . From calculus, it can be shown that an infinite series of terms can be used to generate the result of raising  $e$  (2.7182818) to any power, as Figure 6.10 illustrates. Notice that only the first seven terms are needed to get a reasonable amount of accuracy. Many complex functions can be represented by an infinite series, which we can then implement in software using a loop operation. The following routine generates the first ten terms of the exponential series, using the **POWER** and **FACTORIAL** routines already discussed. We assume, however, that **POWER** and **FACTORIAL** give valid results for all input values (including 0 and 1).

In current data segment...

```
X      DB      5 DUP (?)
TEMP   DB      5 DUP (?)
ETOX   DB      5 DUP (?)
.
```

```
EPOWER  PROC    FAR
          LEA     SI,ZERO                ;predefined constant 0
          LEA     DI,ETOX
          CALL    FAR PTR COPY          ;clear result
          MOV     CX,10                 ;init loop counter
NEXTTERM: LEA     SI,X                   ;compute numerator
          MOV     AL,CL
          CALL    FAR PTR POWER
          LEA     BX,TEMP                ;save numerator
          CALL    FAR PTR SAVE
          MOV     AL,CL                 ;compute denominator
          CALL    FAR PTR FACTORIAL
          LEA     SI,TEMP                ;divide to generate term
          XCHG    BP,DI
          CALL    FAR PTR DIVIDE
          XCHG    BP,DI                ;add current term to result
```

```

        LEA     SI, ETOX
        CALL    FAR PTR ADDER
        LEA     BX, ETOX           ; save result
        CALL    FAR PTR SAVE
        LOOP    NEXTERM
        RET
EPOWER  ENDP

```

Again, XCHG is used to redirect output results back into the math routines. XCHG is also used to swap pointers for storing results in memory. ETOX contains the final result when EPOWER finishes execution.

These examples should serve to illustrate the point that complex mathematical functions can be implemented with a small amount of software. Once a library of these routines has been defined and tested, even more complex equations and functions may be implemented. All that is needed is a CALL to the appropriate subroutine (or collection of subroutines).

**Programming Exercise 6.24:** Modify the TOTAL routine so that 32-bit numbers are added together. Return the sum in registers DX and AX, with DX containing the upper 16 bits of the sum.

**Programming Exercise 6.25:** Write a subroutine called BIGMUL that will compute the 128-bit result obtained by multiplying two 64-bit integers. The two input numbers should be in registers EAX and EBX on entry to BIGMUL. Use the MULTIPLY subroutine in your code to implement a process similar to that shown in Figure 6.6.

**Programming Exercise 6.26:** Use the FACTOR subroutine to find all factors of the number saved in a new variable, INVALUE. Place the factors into a data array called FACTORS.

**Programming Exercise 6.27:** Write a subroutine called BIGADD that will perform a BCD addition of all 32 bits in registers EAX and EBX. Place the result in ECX.

**Programming Exercise 6.28:** Write a subroutine called TOBINARY that will convert the BCD number pointed to by DI into an unsigned binary number. The result should be returned in AX.

**Programming Exercise 6.29:** Write a subroutine called TOBCD that converts the unsigned 16-bit binary number in DX into a BCD number. The result should be returned in DX also.

**Programming Exercise 6.30:** Write a subroutine that performs BCD division by first converting the BCD numbers to binary. DIV should be used to perform the division. The result should be converted back into BCD. Use TOBCD and TOBINARY in your subroutine.

**Programming Exercise 6.31:** Modify the POWER subroutine so that any integer power can be used, including negative powers and 0.

**Programming Exercise 6.32:** Change FACTORIAL so that factorials of 70 or more are not allowed. Return 0 as the result in these cases.

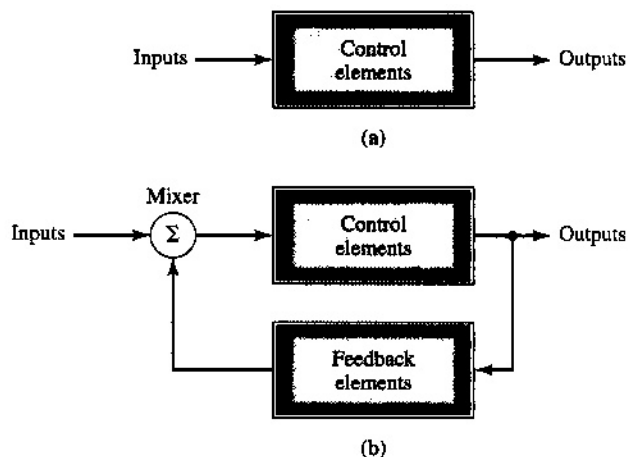
## 6.9 CONTROL APPLICATIONS

In this section, we will examine two examples of how the 80x86 may be used in control applications. Control systems are designed in two different ways: open-loop and closed-loop systems. Figure 6.11 shows two simple block diagrams outlining the main difference between these two types of control systems. An **open-loop control system** uses its input data to affect changes in its outputs and has no feedback. A **closed-loop control system** contains a feedback path where data concerning the present output conditions is sampled and supplied along with the external inputs. A burglar alarm is an example of an open-loop control system. The system may be designed to monitor sensors at various windows and doors. It may also include circuitry to digitize readings from temperature sensors. When any of the sensors detects an abnormal condition (for example, a window opening), the computer may be directed to dial an emergency phone number and play a recorded help message.

A typical application of a closed-loop control system involves the operation of a motor. Suppose we want to control the speed of the motor by making adjustments to an input voltage to the system. The speed of the motor is proportional to the input voltage and increases as the input voltage increases. We cannot simply apply the input voltage to the motor's windings, for it may not be large enough to operate the motor. Usually an amplifier is involved that is capable of driving the motor. But a problem occurs when the motor encounters a load (for example, by connecting the motor shaft to a pump). The increased load on the motor will cause the motor speed to decrease. To maintain a constant speed in the motor at this point, we need an increase in the input voltage. We cannot hope or expect the operator to constantly watch the motor and adjust the input voltage accordingly. For this reason, we add a feedback loop, which is used to sample the motor speed and generate an equivalent voltage. An *error* voltage is generated by comparing the actual speed of the motor (the voltage generated by the feedback circuit) with the desired speed (set by the input voltage). The motor speed voltage may be generated by a tachometer connected to the output of the motor. The error voltage is used to increase or decrease the speed of the motor until it is operating at the proper speed.

Let us look at how the processor might be used to implement the two control systems just described.

**FIGURE 6.11** Control system block diagram: (a) open-loop and (b) closed-loop



## A Computerized Burglar Alarm

In this section, we will use the 80x86 to monitor activity on 100 windows and doors in a small office building. The office building consists of four floors, with fifteen doors and ten windows on each floor. The alarm console consists of an electronic display containing a labeled light-emitting diode for each window and door and a serial data terminal capable of displaying ASCII information. The operation of the system consists of two tasks: (1) illuminating the appropriate LED for all open doors and windows, and (2) sending a message to the terminal whenever a door or window opens or closes. It is necessary to continuously scan all of the windows and doors to detect any changes. The circuitry used to monitor the doors and windows and drive the LED display is connected to the processor's system bus so that all I/O can be done by reading and writing to ports. Figure 6.12 shows the assignments of all input and output devices for the first floor of the office building.

As the figure shows, fifteen door and ten window inputs are assigned for the first floor. Whenever a door or window is open, its associated bit will be low. To sample the bits, the processor must do an I/O read from the indicated port (7000 to 7003). Floors 2, 3, and 4 are assigned the same way, with the following port addresses:

Floor 2: 7004–7007

Floor 3: 7008–700B

Floor 4: 700C–700F

The door and window LEDs for the first floor are illuminated when their respective bits are high. The processor must do an I/O write to ports 7800 through 7803 to activate LEDs for the first floor. The other-floor LEDs work the same way, with these port addresses assigned to them:

Floor 2: 7804–7807

Floor 3: 7808–780B

Floor 4: 780C–780F

The serial device used by the system to communicate with the ASCII terminal is driven by a subroutine called **CONSOLE**. The 7-bit ASCII code in the lower byte of register AL is sent to the terminal when **CONSOLE** is called.

Knowing these definitions, we can design a system to constantly monitor all 100 doors and windows. The technique we will use is called **polling**. Each input port will be read and examined for any changes. If a door or window has changed state since the last time it was read, a message will be sent to the terminal, via **CONSOLE**, indicating the floor and door/window number. Because we need to remember the last state of each door and window, their states must be saved. A block of memory, called **STATUS**, will be used for this purpose. **STATUS** points to a 16-byte block of memory, which we will think of as four blocks of 4 bytes each. Each 4-byte block will store the bits for all doors and windows on a single floor.

When the program first begins operations, the state of each door and window is unknown. For this reason, we initialize **STATUS** by reading all system inputs when the program starts up. The code to perform the initialization is contained in a subroutine called **INIT**, and is as follows:

In current data segment...

STATUS DB 16 DUP (?)

|      |      |            |                                   |
|------|------|------------|-----------------------------------|
| INIT | PROC | FAR        |                                   |
|      | LEA  | DI, STATUS | ;init pointer to STATUS           |
|      | MOV  | DX, 7000H  | ;init pointer to first input port |
|      | MOV  | CX, 16     | ;init loop counter                |

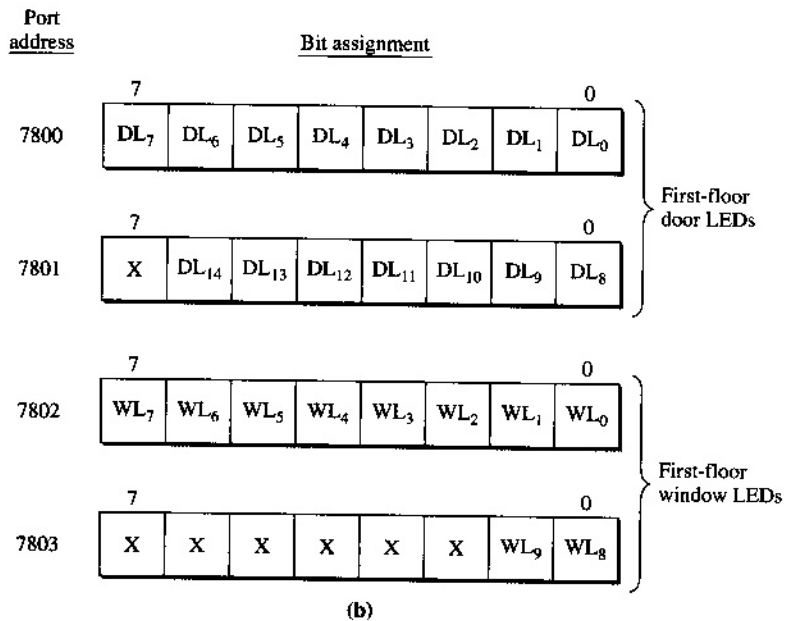
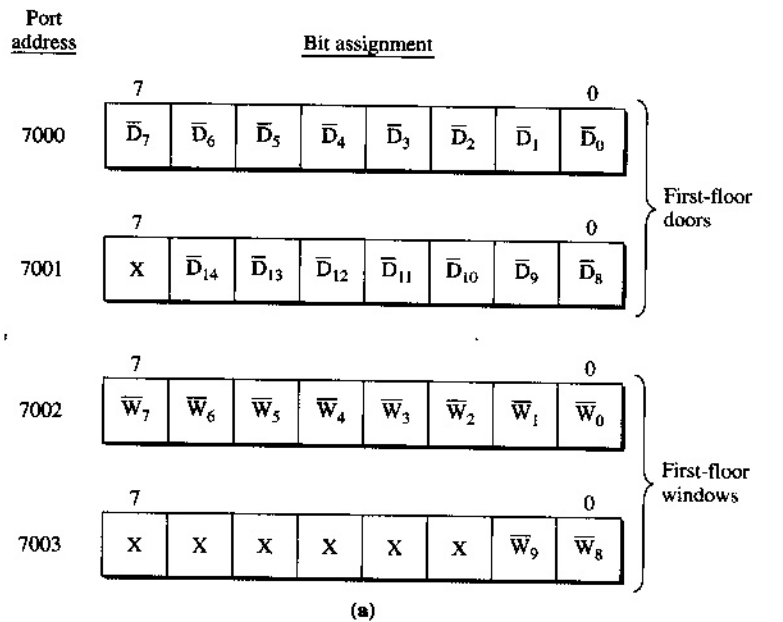


```

SYSREAD: IN      AL,DX           ;read system information
          MOV     [DI],AL        ;save it in memory
          NOT     AL             ;complement input data
          ADD     DX,800H        ;set up output port address
          OUT     DX,AL          ;update display
          SUB     DX,800H        ;generate next input port address
          INC     DX
          INC     DI             ;point to next STATUS location
          LOOP    SYSREAD
          RET
INIT      ENDP

```

**FIGURE 6.12** Burglar alarm I/O assignments: (a) system inputs and (b) system outputs



**FIGURE 6.13** Detecting state changes with XOR

| Previous door state |   | Next door state |   |   |
|---------------------|---|-----------------|---|---|
| A                   | B | A               | B | F |
| 0                   | 0 | 0               | 0 | 0 |
| 0                   | 1 | 0               | 1 | 1 |
| 1                   | 0 | 1               | 0 | 1 |
| 1                   | 1 | 1               | 1 | 0 |

0 ~ Door open  
1 ~ Door closed

Door stayed open.  
Door closed.  
Door opened.  
Door stayed closed.

When INIT completes execution, the display has been updated to show the state of all 100 doors and windows, and STATUS has been loaded with the same information.

Once the initial states are known, future changes can be detected by using an Exclusive OR operation. Remember that Exclusive OR produces a 1 when both inputs are different. Figure 6.13 shows how state changes can be detected with Exclusive OR. To incorporate this into the program, XOR is used during updates to detect changes. Note that up to sixteen changes at once can be detected by XORing entire words. It is then a matter of scanning the individual bits to determine if any state changes occurred. A subroutine called DETECT will do this for us. DETECT will sense any state changes and send the appropriate message (for example, first floor: door 12 opened) to the terminal. When DETECT is called, data registers AX, BX, and DX will be interpreted as follows:

AX: Contains current door or window states.

BX: Contains door or window state changes.

DX: Bit 8 cleared means AX contains door information.

Bit 8 set means AX contains window information.

Bits 0 and 1 contain the floor number (0—first, 1—second, 2—third, 3—fourth)

We will not cover the code involved in getting DETECT to do its job. You are encouraged to write this routine yourself, preferably using a rotate or shift instruction to do the bit testing. Because DETECT will have to output ASCII text strings (for example, "First floor," "Second floor," "opened"), the following code may come in handy:

In current data segment...

```
MSG1      DB      'First floor $'
MSG2      DB      'Second floor $'
MSG3      DB      'Third floor $'
MSG4      DB      'Fourth floor $'
MSG5      DB      'door $'
MSG6      DB      'window $'
MSG7      DB      'opened $'
MSG8      DB      'closed $'
```

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

```
SEND      PROC      FAR
          MOV      AL,[SI]                ;get a message character
          CMP      AL,'$'                ;end of message character?
          JZ       EXIT
          CALL     FAR PTR CONSOLE      ;send character to terminal
          INC      SI                    ;point to next character
          JMP      SEND
EXIT:     RET
SEND      ENDP
```

The SEND subroutine must be entered with register SI pointing to the address of the first character in the text string to be sent. SEND could be a subroutine called by DETECT during its analysis of the door and window states.

Using DETECT, the code to poll all doors and windows in the office building becomes:

```

BEGIN:      CALL  FAR PTR INIT      ;get initial states and update display
            MOV   CX,0              ;start with first floor doors
            MOV   DX,7000H          ;point to first input port
            LEA   SI,STATUS          ;point to STATUS information
NEWFLOOR:   IN    AL,DX             ;get door data
            NOT   AL
            ADD   DX,800H           ;update display
            OUT   DX,AL
            MOV   AH,AL             ;save first eight door states
            SUB   DX,800H           ;point to next door
            INC   DX
            IN    AL,DX             ;get remaining door data
            NOT   AL
            ADD   DX,800H           ;update display
            OUT   DX,AL
            XCHG  AL,AH             ;correct AX for DETECT
            NOT   AX
            MOV   BL,[SI]           ;get past door status
            MOV   BH,[SI + 1]
            ADD   SI,2              ;advance to next status group
            XOR   BX,AX             ;compute stage changes
            XCHG  CX,DX             ;get floor number into DX
            CALL  FAR PTR DETECT    ;find doors that have changed
            XCHG  CX,DX             ;get port address back
            SUB   DX,800H           ;point to window data
            INC   DX
            IN    AL,DX             ;get window data
            NOT   AL
            ADD   DX,800H           ;update display
            OUT   DX,AL
            MOV   AH,AL             ;save first eight window states
            SUB   DX,800H           ;point to next window
            INC   DX
            IN    AL,DX             ;get remaining window data
            NOT   AL
            ADD   DX,800H           ;update display
            OUT   DX,AL
            XCHG  AL,AH             ;correct AX for DETECT
            NOT   AX
            MOV   BL,[SI]           ;get past window status
            MOV   BH,[SI + 1]
            ADD   SI,2              ;advance to next floor
            XOR   BX,AX             ;compute state changes
            OR    CX,100H           ;set bit-8 in CX
            XCHG  CX,DX             ;get floor number into DX
            CALL  FAR PTR DETECT    ;find windows that have changed
            XCHG  CX,DX             ;get port address back
            AND   CH,0              ;clear bit-8
            SUB   DX,800H           ;point to next floor
            INC   DX

```

```

INC    CX                ;increment floor counter
CMP    CX,4              ;done?
JZ     REPEAT
JMP    NEWFLOOR          ;both JMPs are needed since
REPEAT: JMP BEGIN        ;relative range has been exceeded

```

While you write DETECT, do not forget that the main routine uses a number of registers and that these registers should not be altered. The stack would be a good place to store them for safekeeping.

### A Constant-Speed Motor Controller

In this section, we will see how the 80x86 may be used in a closed-loop control system to maintain constant speed in a motor. The schematic of the system is shown in Figure 6.14. The speed control is a potentiometer whose output voltage varies from 0 to some positive voltage. This voltage is digitized by an 8-bit analog-to-digital converter, such that 0 volts is 00H and the most positive voltage is FF. The processor reads this data from port 8000. For a purely digital speed control system, this circuitry is eliminated and the speed is set directly by software.

The motor speed is controlled by the output of an 8-bit digital-to-analog converter (with appropriate output amplifier, capable of driving the motor). The motor's minimum speed, 0 RPM, occurs when the computer outputs 00 to the D/A (by writing to port 8020). The motor's maximum speed occurs when FF is sent to the D/A. A tachometer is connected to the motor shaft through a mechanical coupling. The output of the tachometer is digitized also. Again, the minimum and maximum tachometer readings correspond to 00 and FF. A purely digital system would use a digital shaft encoder instead of a tachometer and A/D. The tach is read from input port 8010H.

Each converter is calibrated with respect to a common reference. In theory, a 17 from the SPEED A/D causes a 17 to be sent to the MOTOR D/A, which in turn causes the TACH A/D to read 17 when at the proper speed. In practice the relationship is not so linear, due to external effects of deadband, friction, and other losses in the motor.

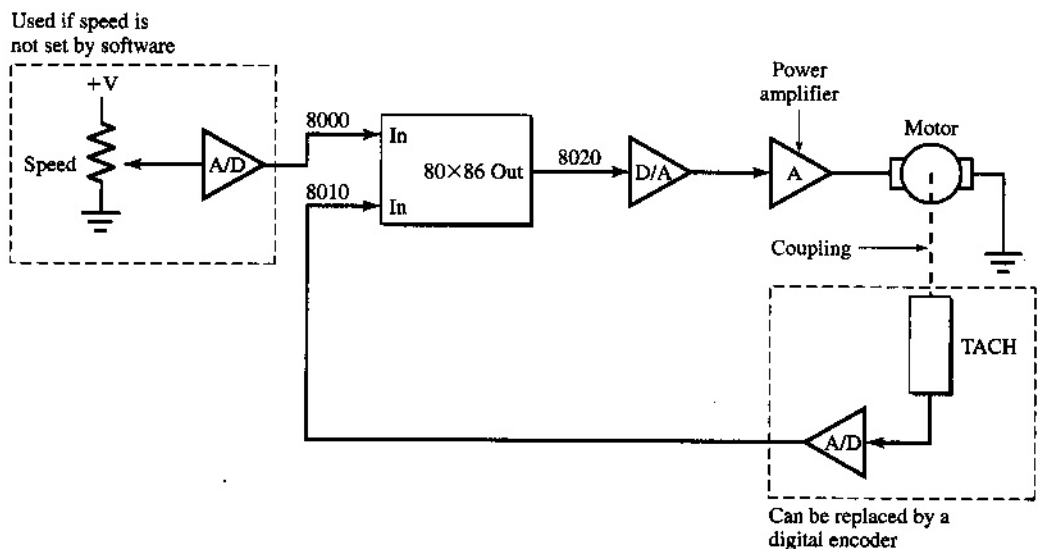


FIGURE 6.14 Constant-speed motor controller

The purpose of the program is to operate the motor at a constant speed by comparing the SPEED data with the TACH data. When SPEED equals TACH, the motor is turning at the desired speed. When SPEED is less than TACH, the motor is spinning too fast. When SPEED is greater than TACH, the motor is rotating too slowly. The idea is to subtract the TACH value from the SPEED value. The difference determines how much the motor speed should be increased or decreased.

```

SERVO:      MOV     AL,0           ;initial motor speed is 0 RPM
            MOV     DX,8020H
            OUT     DX,AL
GETSPEED:    MOV     DX,8000H      ;read new speed value
            IN      AL,DX
            MOV     AH,AL          ;save speed here
            ADD     DX,10H         ;and new tachometer value
            IN      AL,DX
            CMP     AH,AL          ;SPEED minus TACH
            JG      INCREASE
            JZ      GETSPEED       ;no change
            XCHG    AL,AH
INCREASE:    SUB     AH,AL          ;compute error value
            MOV     AL,AH
            CALL    FAR PTR GAIN
            MOV     DX,8020H      ;output new motor speed
            OUT     DX,AL
            JMP     GETSPEED

```

Because the motor's speed will not change instantly from very slow to very fast, or vice versa, the program will loop many times before the motor gets to the proper speed. For safety or functional reasons, it may not be desirable to try to change the motor speed from slow to fast instantly. Instead, the program should *ramp up* to speed gradually by limiting the size of the error voltage presented to the D/A during speed increases. Subroutine GAIN is used for this purpose, to alter the contents of AL before AL is output to the motor D/A. The ramp up/down speed of the motor, and therefore the response of the closed-loop system, will be a function of the operation of GAIN.

---

**Programming Exercise 6.33:** Write the DETECT subroutine used by the computerized burglar alarm.

---

**Programming Exercise 6.34:** An office complex consisting of sixty-four offices and sixteen hallways is to have its lighting controlled by a computer. Each office has one switch to control its light. Hallways have a switch at each end. Each switch is assigned a bit position in a particular memory location that can be read by the computer, and a closed switch represents a 0. Each light (think of all lights in a hallway as a single light) is also assigned a certain bit in a memory location that the computer can write to. A logic 1 is needed to turn on any light. How many byte locations are needed for all I/O? Write a program that will constantly monitor all switches and adjust the complex lighting as necessary.

---

**Programming Exercise 6.35:** Modify the SERVO program so that the motor's speed will ramp up during periods where a large speed increase is desired.

## 6.10 NUMBER CONVERSIONS

When information is exchanged between a human and a computer, it is often necessary to perform a conversion from one number system to another. For example, when numeric digits are entered on a keyboard, their ASCII codes must first be converted into decimal values and then the individual values must be combined into a single, equivalent binary number. In general, this process is referred to as **decimal-to-binary conversion**. Likewise when the computer generates a binary result, the result is often converted into the corresponding ASCII codes for output to the display screen or for storage in a text file. The technique used to perform this transformation is called **binary-to-decimal-conversion**. In this section, we will look at examples of each type of conversion.

### Decimal to Binary

This conversion takes a three-digit sequence of ASCII digits and converts them into an equivalent unsigned 8-bit number. For example, the ASCII digits "1" — "7" — "2" are converted into the hexadecimal number ACH (10101100 binary). The three-digit ASCII sequence can specify any number from 0 to 255 decimal. The DTOB procedure presented here uses the ASCII digits stored in memory locations HUN, TEN, and ONE as the decimal input number. The ASCII bias of 30H is subtracted from each number to get an actual decimal value. The HUN value is multiplied by 100 and the TEN value is multiplied by 10. These values are combined with the value from ONE to get the final unsigned 8-bit result, which is stored in BINVAL. Note that negative numbers are not allowed.

In current data segment...

```

BINVAL DB ?
HUN     DB ?
TEN     DB ?
ONE     DB ?
.
.
.
DTOB    PROC FAR
        MOV     AL,HUN           ;get hundreds digit
        SUB     AL,30H          ;remove ASCII bias
        MOV     BL,100          ;multiply by 100
        MUL     BL
        MOV     CX,AX           ;save temp result
        MOV     AL,TEN          ;get tens digit
        SUB     AL,30H          ;remove ASCII bias
        MOV     BL,10           ;multiply by 10
        MUL     BL
        ADD     CX,AX           ;add to temp result
        MOV     AL,ONE          ;get ones digit
        SUB     AL,30H          ;remove ASCII bias
        ADD     CL,AL           ;add to get final result
        MOV     BINVAL,CL       ;save conversion value
        RET
DTOB    ENDP

```

Input values that exceed 255 (256 to 999) will not produce a correct result because only the lower byte of CX is saved.

A different approach to this conversion process is shown in the following C code:

```
result = 0;
do
{
    gotdigit = getdigit();
    if(gotdigit)
    {
        result = result * 10;
        result = result + digit;
    }
} while(gotdigit);
```

In this code, the `getdigit()` function gets the next digit from the user (if any) and puts its value into the "digit" variable. A sample execution may go like this:

1. result = 0
2. `getdigit()` returns with digit = 1 (`gotdigit` is true)
3. result =  $10 * 0 = 0$
4. result =  $0 + 1 = 1$
5. `getdigit()` returns with digit = 7 (`gotdigit` is true)
6. result =  $10 * 1 = 10$
7. result =  $10 + 7 = 17$
8. `getdigit()` returns with digit = 2 (`gotdigit` is true)
9. result =  $10 * 17 = 170$
10. result =  $170 + 2 = 172$
11. `getdigit()` returns without a new digit (`gotdigit` is false)

The size of the input number that can be converted depends on the size of the "result" variable. Assuming the size is 16 bits, here is one way to translate the C code into assembly language:

```

        MOV     CX, 0
BAK:    CALL    GETDIGIT
        JZ      FIN
        MOV     AX, 10
        MUL     CX
        ADD     AL, BL
        ADC     AH, 0
        MOV     CX, AX
        JMP     BAK
FIN:    RET
```

The `GETDIGIT` subroutine returns with the zero flag set if there is no new digit from the user. Otherwise, the digit value is returned in `BL`. The result is kept in `CX`, which is initialized to 0 and multiplied by 10 each time a new digit is returned.

## Binary to Decimal

This conversion technique takes an unsigned 8-bit quantity and converts it into a three-digit sequence of ASCII digits. For example, the hexadecimal number 81H converts to the ASCII codes 31H, 32H, and 39H, which represent the decimal number 129. The unsigned 8-bit input number can specify any decimal value from 0 to 255.

Examine the `BTOD` procedure. The unsigned 8-bit input value is stored in `BINVAL`. This input number is converted into decimal in two steps. First, it is divided by 100 to get

the hundreds digit. Then, the remainder is divided by 10 to get the tens and ones digits. Each digit is converted into ASCII by adding 30H to it. The three-digit ASCII result is stored in three memory locations: HUN, TEN, and ONE. HUN will take on only the ASCII codes for "0," "1," or "2." TEN and ONE can be any ASCII code from "0" to "9." Placing the result in memory allows further processing (possibly leading 0 suppression) before the result is sent to the display or output in some other way.

In current data segment...

```

BINVAL DB ?
HUN     DB ?
TEN     DB ?
ONE     DB ?
.
.
.
BTOD    PROC FAR
        MOV     AL,BINVAL    ;load binary input value
        SUB     AH,AH        ;prepare for division by 100
        MOV     BL,100
        DIV     BL           ;get hundreds digit
        ADD     AL,30H       ;convert into ASCII digit
        MOV     HUN,AL       ;and save
        XCHG    AL,AH        ;get remainder
        SUB     AH,AH        ;prepare for division by 10
        MOV     BL,10
        DIV     BL           ;get tens digit
        ADD     AL,30H       ;convert into ASCII digit
        MOV     TEN,AL       ;and save
        ADD     AH,30H       ;convert ones digit into ASCII
        MOV     ONE,AH       ;and save
        RET
BTOD    ENDP

```

What other types of conversion might be useful? Ideally, we would like to have all of the common bases represented: binary, octal, decimal, and hexadecimal. Conversions between any two of these bases are straightforward. Additionally, we might want to allow negative integers as well and increase the range of integers to 65,535 or higher. Furthermore, you should consider how a loop may be used (dividing by 10 each time through the loop), so that the same code works for all sizes of input numbers.

---

**Programming Exercise 6.36:** Modify the DTOB procedure so that negative integers are included. The range of *signed* integers represented with 8 bits is -128 to +127.

---

**Programming Exercise 6.37:** Modify the DTOB procedure so that the ASCII digits are entered from the keyboard. Allow a positive integer range from 0 to 65,535.

---

**Programming Exercise 6.38:** Modify the BTOD procedure so that leading zeros are replaced by blanks. For example, if the result is "0" — "5" — "9," the leading 0 in the hundreds position gets replaced by a blank, giving " " — "5" — "9."



**Programming Exercise 6.39:** Modify the BTOD procedure so that the ASCII digits are output to the display screen. Allow a signed range of integers from  $-32,768$  to  $+32,767$ .

## 6.11 DATA STRUCTURES

In this section we will examine a number of different data structures. Data structures are organized groups of data that must be accessed in a certain way. The organization of the data within a structure is up to the programmer. The data structures covered here (linked-lists, stacks, and queues) are often used by programmers to solve many different types of programming problems.

### Linked-Lists

A **linked-list** is a collection of data elements called **nodes** that is created dynamically. Dynamic creation means that the size of the linked-list is not fixed when it is first created. As a matter of fact, it is empty when first created. As an example, if we want to reserve enough room in memory for 100 integer bytes, we use

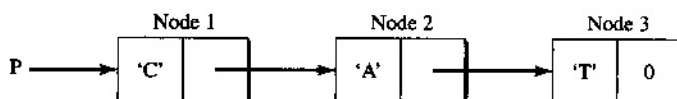
```
DATA DB 100 DUP (?)
```

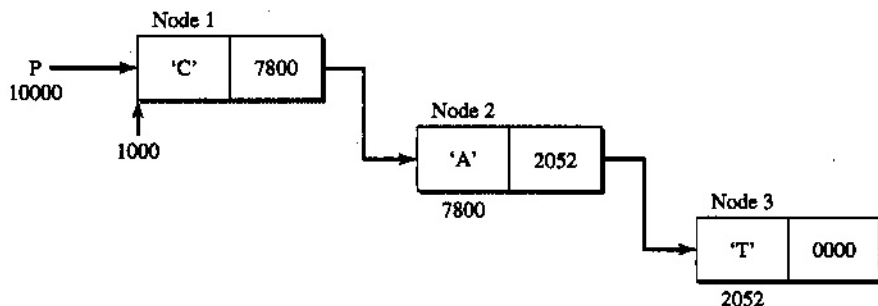
This assembler directive is used because we already know how many numbers are going to be used. The beauty of a linked-list is that its size can be changed as necessary, either increased or decreased, with a maximum size limited only by the amount of free memory available in the system. This method actually saves space in memory, because it does not dedicate entire blocks of RAM for storing numbers. Rather, a small piece of memory is allocated each time a node is added to the linked-list. A node is most commonly represented by a pair of items. The first item is usually used for storing a piece of data. The second item is a pointer; it is used to point to the next node in the linked-list. Figure 6.15 contains an example of a three-node linked-list. Each node in the list stores a single ASCII character. The beginning of the linked-list, the first node, is pointed to by P. The nodes are linked together via pointers from one node to another. The last node in the list, node 3, contains 0 in its pointer field. We will interpret this as a pointer to nowhere (and thus the last node). The empty pointer is commonly called **null**.

The actual representation of the node on a particular system can take many forms. Because the linked-list must reside in memory, it makes sense to assign one or more locations for the data part (or data field), and two locations for the pointer part (also called the pointer field). Why two locations for the pointer field? Because all nodes reside in memory. To point to a certain node, we must know its address, which occupies 16 bits.

For this discussion, assume that all nodes consist of 4 data bytes and 2 address bytes. Consider a subroutine called GETNODE that can be called every time a new node is added to the linked-list. GETNODE must find 6 bytes of contiguous (sequential) memory

**FIGURE 6.15** A sample linked-list





**FIGURE 6.16** A linked-list with address assignments

to allocate the node. When it finds them, it will return the starting address of the 6-byte block in register SI. Let us take another look at our example linked-list, only this time addresses have been added to each node. Figure 6.16 shows how the pointer field of each node contains the address of the next node in the list. The address in the pointer field of node 3 indicates the end of the list. The pointer P may be a register containing 1000, the address of the first node in the list. To generate this list, GETNODE has been called three times. GETNODE returned different addresses each time it was called. First came 1000, then 7800, and finally 2052. Linked-lists do not have to occupy a single area of memory. Rather, they may be spread out all over the processor's address space and still be connected by the various pointer fields.

To add a node to the linked-list, a simple procedure is followed. First, a new node is allocated by calling GETNODE. Register SI holds the address of this new node. The pointer field address of this new node, which will have to be modified to add it to the list, starts at SI plus 4 (because the data field occupies the first 4 bytes). To add the new node to the existing list, a copy of the pointer P is written into the pointer field of the new node. Then, to make the new node the first node in the list, P is changed to the address of the new node. Figure 6.17 shows this step-by-step process, assuming that register DI is used to store P. Once the new node has been inserted, its data field, now pointed to by DI, may be loaded with new data. Assume the new data comes from register AL. Note from Figure 6.17 that insertion of the new node into the beginning of the linked-list has changed its contents from "CAT" to "SCAT." The code to perform the insertion described in Figure 6.17 is as follows:

```

INSERT PROC FAR
    CALL FAR PTR GETNODE      ;get a new node from storage pool
    MOV [SI + 4],DI           ;load pointer field with P
    MOV DI,SI                 ;update pointer P to new node
    MOV [DI],AL               ;load data field with AL
    MOV [DI + 1],20H          ;pad rest of data field with blanks
    MOV [DI + 2],20H
    MOV [DI + 3],20H
    RET
INSERT ENDP
  
```

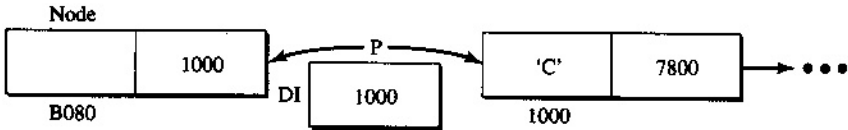
ASCII blank codes (20H) are used to fill the remaining three data bytes in the data field.

Linked-lists are ordinarily used to represent dynamic data structures, such as trees, in memory. The data field may be used to store ASCII characters (as in this example), integers, Boolean data, and even pointers to other linked-lists. Linked-lists are very useful

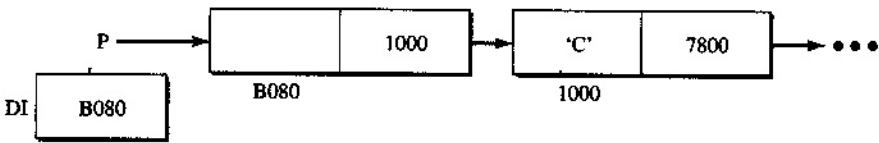
1. GETNODE returns new node.



2. Pointer field of new node is loaded with P.



3. Pointer P to list is changed.



4. Data field of new node is loaded.

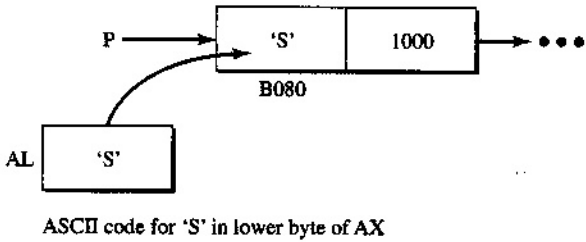


FIGURE 6.17 Adding a node to a linked-list

tools employed in the functions of operating systems. They are also supported by computer languages such as C.

Many computations are greatly simplified by the use of a software-controlled stack or queue. Expression evaluation and round-robin selection algorithms are just two examples of where a stack and a queue are used. The method of implementation is not critical; either may be designed as a special form of a linked-list or simply as fixed-size memory structures. The latter approach will be used here, with address registers pointing to the stack and queue memory structures.

### Stacks

A stack is an area in memory reserved for reading and writing special data items, such as return addresses and register values. For example, a CALL instruction automatically pushes a return address onto the stack (using SP). Registers may be pushed onto the stack (written into stack memory) with a PUSH instruction, as in:

PUSH AX

where the entire contents of AX are written into the stack area pointed to by SP. SP is automatically decremented by 2 during execution.

Items previously pushed onto the stack can be popped off the stack (read out of memory) in a similar fashion; for example, when executing

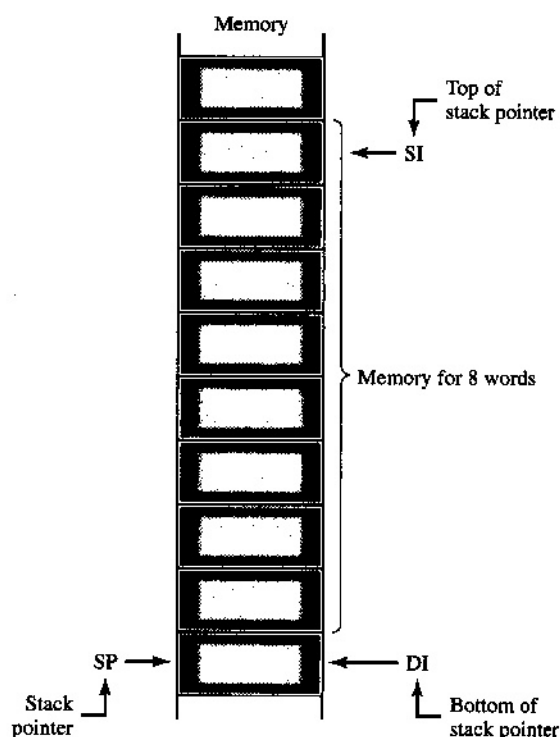
```
POP    AX
```

stack memory is read out into AX, and SP is automatically incremented by 2 during execution. Thus, we see that using a stack requires manipulation of a stack-pointer register.

One characteristic of a stack is that the last item pushed is always the first item popped. For this reason, stacks are commonly referred to as a **LIFO (last in first out) structure**.

It is possible, and often necessary, for a programmer to design a custom stack area for use within a program. For instance, suppose that a programmer requires a stack that allows only eight words to be pushed onto it. The 80x86 has no mechanism for limiting the amount of pushes (or pops) onto a stack. If this is necessary, a set of stack procedures must be written. The following routines implement a stack that allows a maximum of eight pushes. The SPUSH routine is used to place data onto the stack. The data pushed must be in AX. If the push is successful, a success code of 00H will be returned in the lower byte of BX. If more than eight pushes are attempted, the routine returns with error code 80H in the lower byte of BX, without pushing any data. The SPOP routine is used to remove an item from the stack. If a pop is attempted on an empty stack, error code 0FFH is returned in BX. Successful pops return data in AX. The stack-pointer register is SP and is assigned the address of a free block of memory from the storage pool by a routine called MAKESTACK. MAKESTACK must be called before the stack can be used. MAKESTACK also assigns addresses to SI and DI, which are used by SPUSH and SPOP to determine when a stack operation is possible. The structure of the stack is indicated in Figure 6.18. DI points to the bottom of the stack structure and SI points to the top. SP points to the stack location that will be used for the next push or pop.

**FIGURE 6.18** Software-controlled stack structure



Examine the following routines to see how SI and DI are used to check for legal pushes and pops.

```

SPUSH:    CMP    SP,SI          ;ok to push?
          JZ     STKFULL        ;no,go return error code
          PUSH  AX              ;push AX onto stack
          SUB   BL,BL           ;indicate a successful push
          JMP   NEXT

STKFULL:   MOV    BL,80H        ;stack full error code
          JMP   NEXT

SPOP:     CMP    SP,DI          ;ok to pop?
          JZ     STKEMPTY       ;no,go return error code
          POP   AX              ;pop AX off stack
          SUB   BL,BL           ;indicate a successful pop
          JMP   NEXT

STKEMPTY:  MOV    BL,0FFH       ;stack empty error code
NEXT:     ---

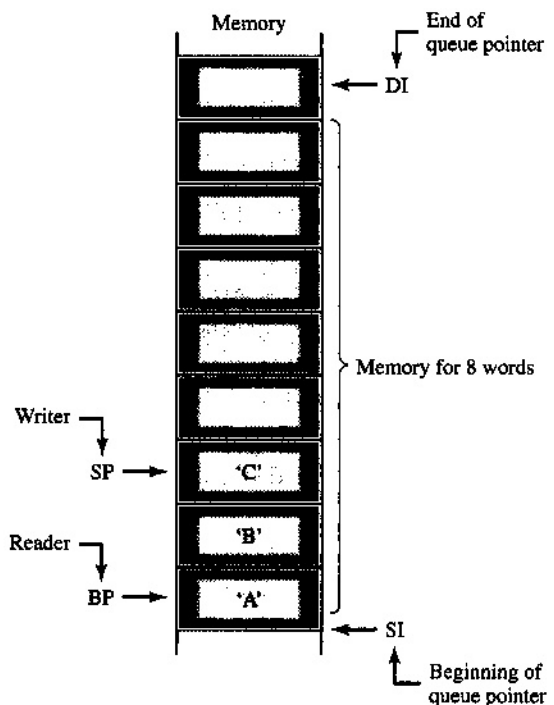
```

Note that multiple stacks can be maintained by saving the contents of SP, SI, and DI and loading new addresses into each register.

## Queues

Queues are also memory-based structures, but their operation is functionally different from that of a stack. In a queue, the first item loaded is the first item to be removed. For this reason, queues are referred to as **FIFO (first in first out) structures**. Figure 6.19 shows a diagram of a queue that has had the data items "A," "B," and "C" loaded into it. "A" was loaded first, "C" last. When we begin removing items from the queue, the "A" will come out first (unlike the stack structure, which would have popped "C" first).

**FIGURE 6.19** Software-controlled queue structure



A routine called MAKEQUEUE is used to assign a block of memory from the storage pool. MAKEQUEUE initializes four registers whose use and meanings are as follows:

| Register | Use/Meaning                         |
|----------|-------------------------------------|
| SP       | Pointer for write operation         |
| BP       | Pointer for read operation          |
| DI       | Contains end-of-queue address       |
| SI       | Contains beginning-of-queue address |

Initially, SP, SI, and BP are all loaded with the same value. The address placed into DI is determined by the desired size of the queue.

Before an item can be written into the queue, SP must be compared with DI. If SP equals DI, it is necessary to reload SP with SI's address. This allows SP to *wrap around* the end of the queue (a similar technique is used to wrap BP around when reading). Next, SP is used to write the data item into memory. Then SP is decremented by 2 to prepare for the next write.

Data is removed from the queue by the read-pointer register BP. After the data is read, BP is decremented by 2. Note that serious data errors result when BP reads a location that has not been written into by SP yet. For this reason it is necessary to pay special attention to the positions of SP and BP within the queue. This part of the queue software is left for you to devise on your own. The routines presented here perform write and read operations with wraparound, but with no error checking.

```

INQUEUE:  CMP     SP,DI        ;need to wrap around?
           JNZ     NOADJSP      ;no
           MOV     SP,SI        ;yes, reload SP
NOADJSP:  PUSH    AX           ;write data into queue memory
           JMP     NEXT
OUTQUEUE:  CMP     BP,DI        ;need to wrap around?
           JNZ     NOADJBP      ;no
           MOV     BP,SI        ;yes, reload BP
NOADJBP:  MOV     AX,[BP]       ;read data out of queue memory
           SUB     BP,2         ;adjust read pointer
NEXT:     ---

```

In both routines, AX is used as the queue data register.

---

**Programming Exercise 6.40:** Write a subroutine called SEARCH that will search the data fields of a linked-list for a certain piece of data. The data item to be located is saved in DL.

---

**Programming Exercise 6.41:** Modify the INSERT subroutine so that new nodes are added to the end of the linked-list, not the beginning.

---

**Programming Exercise 6.42:** Show how two stacks can be maintained by saving copies of all stack-related registers.

---

**Programming Exercise 6.43:** Show how a queue can be controlled without the use of PUSH and POP instructions.

## 6.12 TROUBLESHOOTING TECHNIQUES

This chapter dealt mainly with individual programming modules that performed a single chore. In each case, the subroutine was already developed and tested for you; limitations of the routines were also mentioned.

How is a subroutine created from scratch? The answer is: lots of different ways. Every programmer will use his or her own individual techniques to create a new set of instructions to solve a problem. Some may prefer to use flowcharts or pseudocode, others feel comfortable writing the instructions from scratch. Let us look at one method to create a new subroutine.

First comes the problem. It may be something like this:

"Write a FAR subroutine called FIND7 that counts the number of times the value 7 appears in an array of 100 word-size integers. The starting address of the array is the label VALUES. Return the result in AL."

This specification is very clear about what needs to be done. We begin by framing the subroutine:

```
FIND7      PROC    FAR
.
.
.
                RET
FIND7      ENDP
```

Next, we add the instructions that will allow 100 passes through the loop.

```
FIND7      PROC    FAR
                MOV    CX,100
AGAIN:
.
.
.
                LOOP   AGAIN
                RET
FIND7      ENDP
```

Using register CX does not interfere with the requirement that we return the result in register AL and has the added advantage of being automatically used by the LOOP instruction to control the number of passes.

Now, in addition to initializing CX, we must initialize AL and a pointer to the VALUES array. AL must be initialized to zero, because it represents a count (there really may be zero 7s in the data).

```
FIND7      PROC    FAR
                SUB     AL,AL
                LEA     SI,VALUES
                MOV     CX,100
AGAIN:
.
.
.
                LOOP   AGAIN
                RET
FIND7      ENDP
```

All that remains is the actual loop code. What we want to do, 100 times, is examine a data item from the VALUES array, compare it with 7, and, if equal, increment the AL register. This is done as follows:

```

FIND7      PROC      FAR
           SUB       AL, AL
           LEA       SI, VALUES
           MOV       CX, 100
AGAIN:     CMP       BYTE PTR [SI], 7
           JNZ       NOTEQ
           INC       AL
NOTEQ:
           .
           .
           .
           LOOP      AGAIN
           RET
FIND7      ENDP

```

Is there anything left to do? Yes, it is very important to adjust the pointer register SI with each pass through the loop, so that the CMP instruction always reads a new data item from the VALUES array. Because the size of each data item in VALUES is a word, SI must be incremented by two each pass.

The final FIND7 subroutine looks like this:

```

FIND7      PROC      FAR
           SUB       AL, AL
           LEA       SI, VALUES
           MOV       CX, 100
AGAIN:     CMP       BYTE PTR [SI], 7
           JNZ       NOTEQ
           INC       AL
NOTEQ:     ADD       SI, 2
           LOOP      AGAIN
           RET
FIND7      ENDP

```

FIND7 is now ready to be combined with a tester program for evaluation.

This step-by-step approach, or one you develop on your own, will become very automatic with practice. It requires familiarity with the most basic programming chores: initialization, counting, looping, and comparing. It is worthwhile to invent your own programming problems as well, and then try to write the instructions to solve them.

---

## SUMMARY

In this chapter we examined a number of different applications the 80x86 is capable of performing. These applications find widespread use in industrial and commercial settings. In addition, we covered many different techniques, such as code conversion, table lookup, sorting, and mathematical processing with both binary and BCD numbers. The overall idea is to get a sense of how the instructions can be combined to perform any task that we imagine. Many more applications are possible and we have only scratched the surface here, but the routines presented in this chapter should serve as a foundation on which to build when you try to write an application of your own.



---

**STUDY QUESTIONS**

1. What is an advantage of writing programs with pseudocode?
2. Is it possible for a software tester to fully test a new piece of code?
3. How must the DS and ES segment registers be initialized for the routines presented in Section 6.6?
4. Suppose that the number of integers to sort is fixed to a maximum of 32. Is there a different sorting technique that will sort the input numbers with fewer loops/passes than a bubble sort? If so, explain your technique.
5. How must the format for storing BCD digits be changed to allow for fractional numbers like 0.783 or 457.05?
6. How might exponents be processed in numbers like 35E3 or 2.6E-7?
7. How are round-off errors eliminated by using BCD?
8. Consider the normalization of two numbers, one positive and one negative. Are the exponents adjusted in the same way for each number?
9. In the treatment of BCD numbers, the proposed format contained no provision for representing negative numbers. How might the format be changed to include them?
10. The 5-byte BCD format discussed in this chapter uses 1 byte for a signed exponent and 4 bytes for the mantissa. What is the largest positive integer that can be represented?
11. Why are field terminator characters in a database record useful?
12. When converting from one base to another, do you find any similarities in technique?
13. What does a specification for a subroutine contain?
14. Describe how to develop a subroutine.
15. What is polling?

---

**ADDITIONAL PROGRAMMING EXERCISES**

1. Modify the TESTQUAD driver program so that ten values are used to test the QUAD routine. Use a data table in your new program.
2. Write a subroutine called VALDIGIT, which will determine if the ASCII code contained in the lower byte of AL is a digit from "0" to "9." If so, simply return. If not, jump to ERROR.
3. Modify PACKBCD to include signed BCD numbers. If the first character entered by the user is a minus sign, place 80H into SIGN. If the first character is a number or a plus sign, place 0 into SIGN.
4. Write a tester program for the MAXVAL procedure. Use the following data to test the routine:

DB 5,50,20,77,6,3,22

5. Write a subroutine that will convert the BCD number stored in BCDNUM into a binary number. Return the result in AX.
6. Write a subroutine called STRSIZE that returns the number of characters in a text string. The text string is terminated with an ASCII return character (ODH). The string begins at address TEXTLINE. Return the character count in BX.
7. Write a subroutine to find the average of a block of words that starts at location SAMPLES and whose length, in words, is saved in SIZE. Place the average in AVERAGE.

8. Write a tester program for the SORT routine. Use the following data to test the procedure:

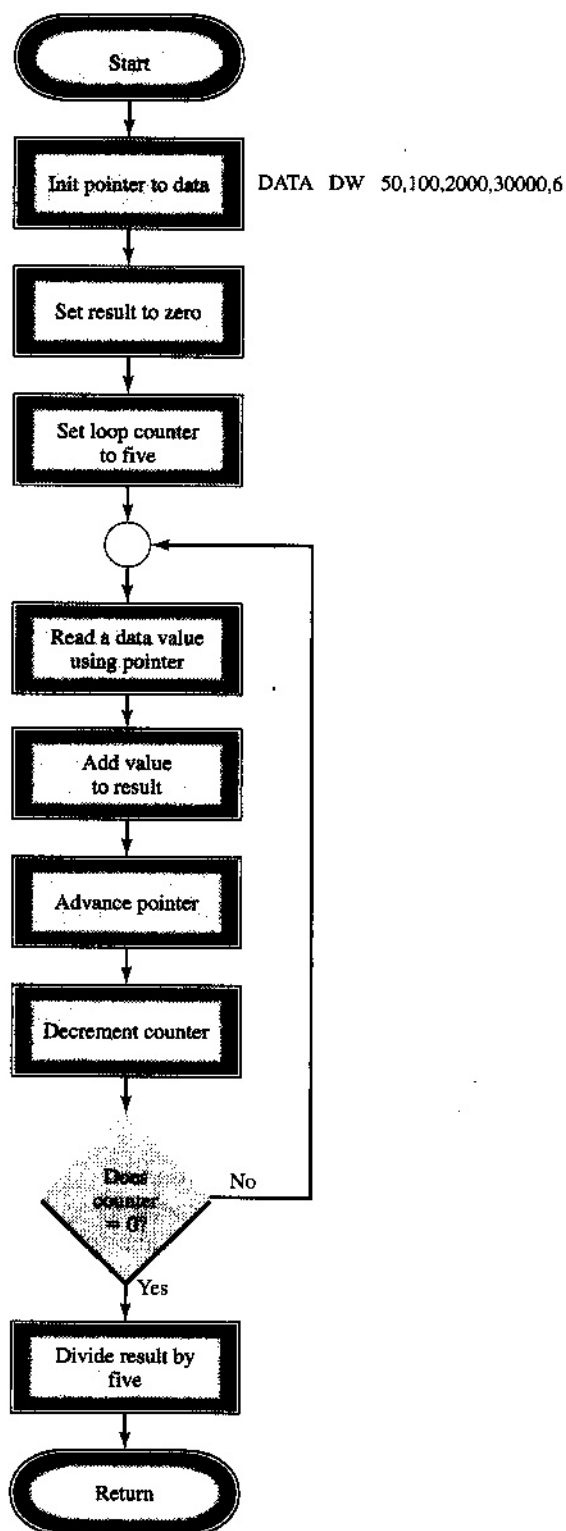
DB 9,1,8,2,7,3,6,4,5

9. Write a subroutine called JUSTIFY that will left justify any BCD number stored in the 4-byte array called BCDIN. For example, if BCDIN contains 00 05 37 19, JUSTIFY must replace BCDIN with 53 71 90 00.
10. Two sample routines for performing BCD multiplication were presented in this chapter. Both routines had limited precision. Build on these two routines by extending their precision. For example, BCDMUL2 was limited to multiplying two-digit numbers. Write a routine that will multiply four-digit BCD numbers, using BCDMUL2 as a subroutine.
11. Write a subroutine called EXPONENT that will return a signed 8-bit exponent in register BL. Inputs to EXPONENT are SI, DI, and AX. SI and DI both point to the exponent byte of the two numbers being divided (with DI pointing to the divisor exponent). AX initially contains an exponent adjustment value (in signed 8-bit format) that must always be added to the generated exponent value.
12. Write a subroutine called ZEROCHECK that examines the BCD number pointed to by register BP and returns with the zero flag set if the BCD number is equal to 0.
13. Write a subroutine called HYPOT that computes the hypotenuse of a triangle whose sides have lengths pointed to by SI and DI. Return the result address in BP. The lengths are stored as words.
14. Find the infinite series for SIN(X) in a calculus book and implement it in a subroutine called SIN. Use EPOWER as an example of how to do this.
15. Implement COS(X) via subroutine COS. Use the following formula as a guide and solve it for COS(X) before writing any code:

$$\sin^2(X) + \cos^2(X) = 1$$

Make use of ROOT and SIN in your subroutine.

16. Consider the office complex for Programming Exercise 6.34 and its associated definitions. Suppose for reasons of efficiency that no light may be on continuously for more than 30 minutes after 5 P.M. For example, if an office light is turned on at 6:17 P.M., the computer automatically shuts it off at 6:47. Use the PC's system clock interrupt function to control an automatic shutoff routine for the lights. *Note:* The light may be immediately turned on again (for another 30 minutes) if someone in the office hits the switch again. The automatic shutoff feature ends at 6 A.M.
17. Write a routine called BTOB that takes the 8-bit number in AL and displays its binary equivalent on the screen. For example, if AL contains 4EH, "01001110" is output to the display.
18. Write a routine that converts a Roman-numeral input string (such as "MCMLXXI") and determines the decimal equivalent. Return the result in AX.
19. Repeat Exercise 18 for the opposite conversion, decimal-to-Roman numeral.
20. Write a subroutine that returns with the zero flag set if the input coordinates in AX are inside the rectangle defined by coordinates in BX and CX. Register BX indicates the upper left corner (BH = row, BL = column), and CX the lower right corner.
21. Determine the instructions necessary to implement the flowchart shown in Figure 6.20.

**FIGURE 6.20** For Programming Exercise 21

---

# CHAPTER 7

---

## Advanced Programming Applications

---

### OBJECTIVES

In this chapter you will learn about:

- Linking separate object files together
- Creating and using an object code library
- Creating and using source code macros
- Predicting the execution time of a block of code
- How interrupts are supported and used by DOS
- How two programs can execute “simultaneously”
- How memory is managed by DOS
- Using information from the PC’s mouse
- Writing a memory-resident program
- Beating your computer at tic-tac-toe
- Testing for protected-mode operation
- How assembly language interfaces with C
- How assembly language is used in Windows and Linux.

### KEY TERMS

|                       |                               |                                   |
|-----------------------|-------------------------------|-----------------------------------|
| C calling convention  | Nested macro                  | Syscall                           |
| Conditional statement | Overhead                      | TSR (terminate and stay resident) |
| Console application   | Paragraph                     | Windowed application              |
| Interrupt hook        | Parameter                     |                                   |
| Local variable        | Portable programming language |                                   |
| Macro                 | Stall                         |                                   |
| Memory management     |                               |                                   |

---

### 7.1 INTRODUCTION

The material presented in this chapter is not a complete treatment of the many truly advanced things that can be accomplished by a good programmer and a PC. Instead, a good subset of the more interesting and useful advanced topics has been chosen. For some,

memory-resident programming might be very useful; for others, it could be instruction execution time. You are encouraged to cover as many of the sections as you can to round out your programming experience.

Section 7.2 shows how separate source files can be assembled and linked. Section 7.3 discusses the use of macros to simplify source code. Section 7.4 details the process used to determine execution time of a group of instructions. This is followed by a second treatment of interrupts in Section 7.5. Section 7.6 shows how a specific DOS interrupt can be used to run more than one program at a time. Memory management is the subject of Section 7.7. Section 7.8 shows how mouse movements can be tracked with a simple program. In Section 7.9, a memory-resident program is discussed to illustrate how a hot key is created. An easy-to-beat tic-tac-toe game is presented in Section 7.10. The next two sections, 7.11 and 7.12, cover protected-mode detection and C language interfacing. The chapter concludes with a look at assembly language programming in the Windows and Linux environments in Sections 7.13 and 7.14.

---

## 7.2 USING THE EXTRN AND PUBLIC ASSEMBLER DIRECTIVES

Recall that the usual process of creating an executable program (.EXE or .COM) requires a number of steps. These steps are as follows:

1. Write source program (.ASM file)
2. Assemble program (via ML)
3. Link program (via LINK)

In a large organization, such as a computer software company, the programs being created are the work of many individual programmers, each working on a different portion of the main program. Their individual procedures can communicate through the use of memory locations and registers while the program is running, but how is the main program created from the many individual pieces written by each programmer? In this section, we will see how separate source files can be assembled and then linked into a single executable program.

### Creating Separate Source Modules

Let us examine two source modules. The first module, from the source file DISPBIN.ASM, looks like this:

```
;Procedure DISPBIN.ASM: Display value of AL in binary on the screen.
;
    .MODEL SMALL
    .CODE

    PUBLIC DISPBIN           ;for linking

DISPBIN PROC FAR
    MOV     CX,8             ;set up loop counter
NEXT:  SHL     AL,1          ;move bit into carry flag
    PUSH    AX              ;save number
    JC      ITIS1           ;was the LSB a 1?
    MOV     DL,30H          ;load '0' character
    JMP     SAY01           ;go display it
```

```

ITIS1:  MOV    DL,31H           ;load '1' character
SAY01:  MOV    AH,2             ;display character function
        INT    21H             ;DOS call
        POP    AX              ;get number back
        LOOP   NEXT            ;and repeat
        RET
DISPBIN ENDP
        END

```

Notice the use of the statement:

```
PUBLIC DISPBIN
```

This assembler directive informs the linker that the value of DISPBIN must be made available at link time.

The second source module, from the source file TESTBIN.ASM, looks like this:

```

;Program TESTBIN.ASM: Test the DISPBIN display procedure.
;
        .MODEL SMALL
        .CODE

        EXTRN DISPBIN:FAR      ;for linking

        .STARTUP
        SUB    AL,AL           ;clear counter
AGAIN:  PUSH   AX              ;save counter
        CALL   DISPBIN         ;display counter in binary
        MOV    DL,20H          ;load blank character
        MOV    AH,2            ;display character function
        INT    21H             ;DOS call
        MOV    AH,2            ;output a second blank
        INT    21H
        POP    AX              ;get counter back
        INC    AL              ;increment it
        JNZ    AGAIN           ;and repeat until counter equals zero
        .EXIT
        END

```

The corresponding link statement in this source module is:

```
EXTRN DISPBIN:FAR
```

which informs the linker that DISPBIN is an external *far* label. The value of DISPBIN will be used by the assembler and the linker to create the correct code for the CALL DISPBIN instruction.

When the first source module is assembled (via ML /c DISPBIN.ASM), the object file DISPBIN.OBJ is created. TESTBIN.OBJ is created by assembling the second source module (via ML /c TESTBIN.ASM).

The directory information for each file is as follows:

```

DISPBIN  OBJ          135 03-20-97   9:51a
TESTBIN  OBJ          168 03-20-97   9:55a

```

You might agree that the size of both object files is larger than that required by the machine code for each module's instructions. This is due to the fact that ML has placed additional information into each object file to indicate what public or external variables have been used.

Note that neither program can exist by itself. The only way to make an executable program out of them is to *combine* them. This is done with a simple linker statement:

```
LINK TESTBIN + DISPBIN, , ;
```

which creates TESTBIN.EXE.

If TESTBIN is linked by itself (via LINK TESTBIN,,:), we get the following error message:

```
TESTBIN.OBJ(A) : error L2029 : 'DISPBIN' : unresolved external
```

So, the linker is able to determine if all necessary variable requirements have been met by matching every EXTRN reference it encounters with a PUBLIC directive for the same variable.

## Building and Using an Object Code Library

An object code library is a file that contains a collection of many individually linked code modules. For example, an object code library called NUMOUT.LIB contains the code for each of the following routines:

| <i>Routine</i> | <i>Function.</i>                         |
|----------------|------------------------------------------|
| DISPBIN        | Display AL in binary on screen           |
| DISPHEX        | Display AL in hex on screen              |
| DISPHEX_16     | Display AX in hex on screen              |
| DISPBCD        | Display AL in BCD on screen              |
| DISPINT        | Display AL as unsigned integer on screen |
| DISPINT_16     | Display AX as unsigned integer on screen |

These six routines provide many different ways for outputting numerical values. Having them already written and contained within a code library is convenient, because they need only be linked with another module (a driver or some other main program).

To create a code library, use the LIB utility supplied with DOS. The command:

```
C:\> LIB NUMOUT.LIB
```

will cause the following action by LIB:

```
Library does not exist. Create?
```

Answer **y** for yes to create the new library. LIB will then prompt for other inputs, but these can be ignored by hitting return:

```
Operations: <cr>
```

```
List file: <cr>
```

To verify that NUMOUT.LIB was created, we can use the DIR command. The result is as follows:

```
NUMOUT LIB 1,033 03-20-97 11:29a
```

The reason NUMOUT.LIB is not empty is that LIB requires library files to have a certain internal format that makes it possible to support multiple object code modules.

To add an object code module to an existing library, we use the following command:

```
C\> LIB NUMOUT + DISPBIN;
```

The LIB utility will add the code for DISPBIN to the NUMOUT.LIB library. This command is used for each of the six DISP modules.

To use a module contained within a library file, we use the linker in a different way. For example, to create TESTBIN.EXE using the NUMOUT library, we use this LINK command:

```
C\> LINK TESTBIN, , NUMOUT
```

which causes the linker to search the NUMOUT library file for external references.

What happens when you decide to rewrite a procedure contained within a code library? The new procedure must be assembled to create an object file. Then the existing object file must be replaced with the new one. For example, to replace NUMOUT's DISPINT procedure with a new version of DISPINT, use the command:

```
C\> LIB -DISPINT +DISPINT
```

This allows the library file to be updated as the need arises.

There are many other useful output routines that could be added to NUMOUT.LIB. What kind of procedures might be placed into a NUMIN library?

**Programming Exercise 7.1:** Write a DISPOCT procedure to display the value of AL in octal on the screen.

**Programming Exercise 7.2:** Write a DISPBCD\_16 procedure that outputs the BCD number in register AX on the screen.

**Programming Exercise 7.3:** Write a tester program for each of the other five display routines. What do they all have in common?

## 7.3 USING MACROS

**Macros** provide the programmer with a powerful new way to write programs. In this section we will examine a number of standard macro forms and uses.

### A Simple Macro

The following group of instructions is used to define a simple macro:

```
DISP_MSG  MACRO
           MOV    AH,9           ;display string function
           INT    21H           ;DOS call
           ENDM
```

The name of the macro is DISP\_MSG, and it is defined on the first line of the macro definition (much like a procedure name is defined when used with PROC). The instructions enclosed between MACRO and ENDM (end macro) are the contents of the macro.



The operation of the macro during assembly is to replace all occurrences of `DISP_MSG` within the source file with the `MOV` and `INT` instructions. For example, the source statements:

```
LEA DX,ABC
DISP_MSG
LEA DX,DEF
DISP_MSG
```

specify two *calls* to the `DISP_MSG` macro. A macro call triggers the assembler into *expanding* a macro. The resulting source code is:

```
LEA DX,ABC
MOV AH,9      ;display string function
INT 21H       ;DOS call
LEA DX,DEF
MOV AH,9      ;display string function
INT 21H       ;DOS call
```

prior to assembly. The macro provides a convenient way for the programmer to replace groups of instructions with a single macro name. This saves the programmer time typing in the source file and leads to smaller source file sizes (although not in a smaller executable file).

#### ■ EXAMPLE 7.1

Suppose that a programmer is busy with a program requiring frequent `SHL` operations on register `AX`. The statements:

```
SHL AX,1
SHL AX,1
SHL AX,1
SHL AX,1
```

can be replaced by a single macro named `SHL_AX` everywhere they occur. The macro definition will be as follows:

```
SHL_AX    MACRO
           SHL AX,1
           SHL AX,1
           SHL AX,1
           SHL AX,1
           ENDM
```

A different macro definition uses the special `REPT` directive to repeat the four `SHL` statements:

```
SHL_AX    MACRO
           REPT 4
             SHL AX,1
           ENDM
           ENDM
```

Note that the two instructions:

```
MOV CL,4
SHL AX,CL
```

will work just as well. ■

## Adding a Parameter to a Macro

A macro becomes more useful when we are able to use it for more than one thing. The `DISP_MSG` macro is useful from the standpoint that we are freed from typing:

```
MOV  AH,9
INT  21H
```

over and over again. This is improved by allowing the macro to expand into all three instructions required to display a string. Consider the `SEND_MSG` macro shown here:

```
SEND_MSG  MACRO  ADDR
           LEA    DX,ADDR      ;set up pointer to string
           MOV    AH,9        ;display string function
           INT    21H         ;DOS call
           ENDM
```

In this macro definition, the symbol `ADDR` is a **parameter** passed to the macro for expansion purposes. The source statement:

```
SEND_MSG ABC
```

causes the `ADDR` symbol to become `ABC`. The resulting expansion looks like this:

```
LEA  DX,ABC      ;set up pointer to string
MOV  AH,9        ;display string function
INT  21H         ;DOS call
```

The single source statement `SEND_MSG ABC` is certainly preferable to typing in all three instructions.

Because I/O is a big part of many programs, other macros designed to assist with I/O would be very useful. Two more character-based macros are as follows:

```
DISP_CHAR  MACRO  DATA
           MOV    DL,DATA      ;load ASCII character
           MOV    AH,2        ;display output function
           INT    21H         ;DOS call
           ENDM

CRLF       MACRO
           DISP_CHAR 0DH      ;output a cr
           DISP_CHAR 0AH      ;output a lf
           ENDM
```

The second macro, `CRLF`, actually is an example of **nested macro**. A nested macro is a macro that contains a macro call within its definition. So, macros can make it easier to write other macros.

More than one parameter may be defined within a macro. In this case, parameter names must be separated by a comma in the macro definition, as in:

```
<Name>  MACRO  <parameter 1>,  <parameter 2>
```

### ■ EXAMPLE 7.2

The macro `SAXPY` defined here multiplies the `AX` register by `XVAL` and adds `YVAL` to the product:

```
SAXPY  MACRO  XVAL, YVAL
       MUL    XVAL
       ADD    AX,YVAL
       ENDM
```

When called with the following parameters;

SAXPY BX, 25

we get:

MUL BX

ADD AX, 25 ■

## Using Labels within Macros

Suppose that we need to replace the following group of instructions with a macro:

```

                CMP  AL, 10
                JNC  NOADD
                ADD  AL, 7
NOADD:         ADD  AL, 30H
                MOV  DL, AL
                MOV  AH, 2
                INT  21H

```

The NOADD label will cause an assembly error if the macro is called more than once because the assembler will try to define a second NOADD symbol.

The solution to this problem is to define the label NOADD as a **local variable**, which means a variable that has meaning only *inside* the current macro expansion. The macro definition then becomes:

```

HEXOUT  MACRO
        LOCAL  NOADD
        CMP  AL, 10
        JNC  NOADD
        ADD  AL, 7
NOADD:   ADD  AL, 30H
        MOV  DL, AL
        MOV  AH, 2
        INT  21H
        ENDM

```

Every time HEXOUT is called, the assembler will define a new name for the NOADD label.

## Conditional Macro Expansion

There are times when a program is written in such a way that it can be assembled in more than one way. For example, the program might be a numeric display application. One assembly might create an executable module for decimal numbers, another for hexadecimal numbers.

To allow a choice during assembly, a **conditional statement** must be used. Consider the following macro:

```

SHIFT  MACRO  SIZE
        IF  SIZE EQ 8
            SHR  AL, 1
        ELSE
            SHR  AX, 1
        ENDIF
        ENDM

```

This macro uses the value of the `SIZE` parameter to determine which `SHR` instruction to generate. The `SHR AL,1` instruction is produced by `SHIFT 8`. `SHR AX,1` is generated when `SHIFT 16` is used. The programmer may place as many statements as needed between `IF` and `ELSE`, and between `ELSE` and `ENDIF`.

The types of conditions that may be tested within the `IF . . . ELSE` statement are as follows:

| Test | Meaning               |
|------|-----------------------|
| EQ   | Equal                 |
| NE   | Not equal             |
| LT   | Less than             |
| GT   | Greater than          |
| LE   | Less than or equal    |
| GE   | Greater than or equal |

### ■ EXAMPLE 7.3

The `MULTIPLY` macro shown here moves the `NUMBER` value into either register `BL` or `BX`, depending on the size of `NUMBER`. The appropriate `MUL` instruction is also generated.

```

MULTIPLY    MACRO    NUMBER
             IF NUMBER GT 255
                 MOV    BX,NUMBER
                 MUL    BX
             ELSE
                 MOV    BL,NUMBER
                 MUL    BL
             ENDIF
             ENDM

```

## Macro Operators

A number of operators are defined for use within a macro definition. Four of these operators, and their meanings, are:

|       |                          |
|-------|--------------------------|
| %     | Expression               |
| <...> | Literal character string |
| ::    | Macro comment            |
| &     | Substitute               |

The expression operator (%) is used to interpret the *value* of a parameter, and use the value in the expansion. For example, the macro:

```

MOV_AL    MACRO    VAL
MOV    AL,VAL
ENDM

```

can be called with `MOV_AL 2*30`, or with `MOV_AL %2*30`, resulting in two different expansions, as shown here:

```

MOV_AL    2*30    -->    MOV    AL,2*30
MOV_AL    %2*30   -->    MOV    AL,60

```

It may be necessary in some cases to work with the value of a parameter instead of the actual parameter text.

The literal character string operators (<...>) are used in conjunction with the substitute operator (&) to place a specific character string into the expanded macro instruction. Consider the following string definitions:

```
STR1    DB    'This is a message',0DH,0AH,'$'
STR2    DB    'This is another message',0DH,0AH,'$'
```

If many messages need to be defined in the indicated format, the literal and substitute operators can be used in the following macro to allow easier string generation:

```
S_MAKE   MACRO NUM, TXT
          STR&NUM    DB    '&TXT',0DH,0AH,'$'
          ENDM
```

The S\_MAKE macro must be called like this:

```
S_MAKE    1, <This is a message>
S_MAKE    2, <This is another message>
```

You can see that the value of NUM was substituted into STR&NUM to generate STR1 and STR2. All characters between the literal operators < and > were substituted for &TXT.

Another example of this type of substitution expansion is as follows:

```
SHIFT    MACRO,    DIR,REG
          SH&DIR    REG,1
          ENDM
```

This macro creates SHL or SHR instructions for *any* register, as in the following types of calls:

```
SHIFT    R, AX    -->    SHR    AX,1
SHIFT    L, BL    -->    SHL    BL,1
```

The macro comment operator (;;) prevents the comments included inside a macro definition from being repeated each time the macro is expanded. This avoids cluttering up the source file with a collection of identical comments.

## TESTMAC: A Macro Expansion Example

The TESTMAC program shown here uses the macro structures previously covered to perform useful I/O with the user.

```
;Program TESTMAC.ASM: Test macro expansion.
;
```

```
        .MODEL SMALL
        .DATA
MSG1    DB    'This message was output by the DISP_MSG macro.',0DH,0AH,'$'
MSG2    DB    'This message was output by the SEND_MSG macro.',0DH,0AH,'$'
MSG3    DB    ' decimal equals $'
MSG4    DB    ' hexadecimal.',0DH,0AH,'$'
WMSG    DB    'Hit <C> to continue...$'

DISP_MSG    MACRO
              MOV     AH,9           ;;display string function
              INT     21H           ;;DOS call
              ENDM
```

```

SEND_MSG    MACRO    ADDR
             LEA      DX,ADDR      ;;set up pointer to string
             MOV      AH,9         ;;display string function
             INT      21H         ;;DOS call
             ENDM

DISP_CHAR    MACRO    DATA
             MOV      DL,DATA      ;;load ASCII character
             MOV      AH,2         ;;display output function
             INT      21H         ;;DOS call
             ENDM

CRLF        MACRO
             DISP_CHAR 0DH
             DISP_CHAR 0AH
             ENDM

NUM_OUT      MACRO    DATA, BASE
             IF BASE EQ 10
                 MOV    AL,DATA      ;;load number
                 MOV    BL,100       ;;load divisor
                 CALL   DEC_OUT      ;;find and display 100's digit
                 MOV    BL,10       ;;load divisor
                 CALL   DEC_OUT      ;;find and display 10's digit
                 ADD    AL,30H       ;;add ASCII bias to 1's digit
                 DISP_CHAR AH       ;;display digit
             ELSE
                 MOV    AL,DATA      ;;load number
                 PUSH   AX           ;;save it
                 MOV    CL,4         ;;load shift counter
                 SHR    AL,CL        ;;shift upper nybble down
                 CALL   HEXOUT       ;;displays MS hex digit
                 POP    AX           ;;get number back
                 CALL   HEX_OUT      ;;display LS hex digit
             ENDIF
             ENDM

HIT_C        MACRO
             LOCAL WAIT_C, EXIT
             SEND_MSG WMSG

WAIT_C:      MOV    AH,1             ;;read keyboard function
             INT    21H             ;;DOS call
             CMP    AL,'c'          ;;is character a 'c'?
             JZ     EXIT
             CMP    AL,'C'          ;; is character a 'C'?
             JNZ    WAIT_C          ;;no, repeat until it is

EXIT:        CRLF                  ;;output cr and lf
             ENDM

.CODE
.STARTUP
LEA    DX,SMSG1                    ;set up pointer to message
DISP_MSG                                ;display message
SEND_MSG SMSG2                    ;display second message
DISP_CHAR 'H'                      ;output 'Hi.'
DISP_CHAR 'i'
DISP_CHAR '.'

```

```

CRLF                ;and cr and lf
NUM_OUT 100, 10     ;display 100 in decimal
SEND_MSG MSG3       ;display decimal message
NUM_OUT 100, 16     ;display 100 in hexadecimal
SEND_MSG MSG4       ;display hex message
HIT_C               ;wait for a c/C
HIT_C               ;wait for another c/C
.EXIT

HEX_OUT PROC NEAR
AND AL,0FH          ;clear upper nybble
ADD AL,30H          ;add ASCII bias
CMP AL,'9'+1        ;do we need alpha fix?
JC NOADD7
ADD AL,7            ;add alpha bias
NOADD7: DISP_CHAR AL ;display hex digit
RET
HEX_OUT ENDP

DEC_OUT PROC NEAR
SUB AH,AH           ;prepare for division
DIV BL             ;find digit
ADD AL,30H         ;add ASCII bias
PUSH AX            ;save remainder
DISP_CHAR AL       ;display digit
POP AX             ;get remainder back
XCHG AH,AL         ;save new number
RET
DEC_OUT ENDP

END

```

Note the use of ;; in the macro definitions. This is done to keep the look of the resulting list file uncluttered.

Let us examine a few sections of output from the list file.

```

0017 8D 16 0000 R    LEA DX,MSG1      ;set up pointer to message
                        DISP_MSG      ;display message
001B B4 09          1    MOV AH,9
001D CD 21          1    INT 21H
                        SEND_MSG MSG2 ;display second message
001F 8D 16 0031 R    1    LEA DX,MSG2
0023 B4 09          1    MOV AH,9
0025 CD 21          1    INT 21H

```

Note that when a macro name is encountered, as in DISP\_MSG and SEND\_MSG, the instructions generated by macro expansion are flagged with a 1. This indicates the nesting level for the current macro.

This is further illustrated during expansion of the CRLF macro, which calls the DISP\_CHAR macro:

```

                                CRLF                ;and cr and lf
0039 B2 0D          2      MOV DL,0DH
003B B4 02          2      MOV AH,2
003D CD 21          2      INT 21H
003F B2 0A          2      MOV DL,0AH
0041 B4 02          2      MOV AH,2
0043 CD 21          2      INT 21H

```

You are encouraged to examine the list file for TESTMAC for other examples of how the macros were expanded.

The resulting output from TESTMAC's execution is as follows:

```
This message was output by the DISP_MSG macro.
This message was output by the SEND_MSG macro.
Hi.
100 decimal equals 64 hexadecimal.
Hit <C> to continue...noC
Hit <C> to continue...c
```

Recall that the HIT\_C macro waits for c or C to be entered on the keyboard.

---

**Programming Exercise 7.4:** Write a new macro called SENDER that has the following calling conventions:

```
SENDER C, X      --> output character 'X' via Function 2
SENDER S, <Hello> --> output Hello string via Function 9
```

Conditional macro expansion should be used to generate the appropriate instructions.

---

**Programming Exercise 7.5:** Modify the NUM\_OUT macro to include binary and octal (base 8) output numbers.

---

## 7.4 INSTRUCTION EXECUTION TIMES

An important topic in the study of any microprocessor involves analysis of the execution time of programs, subroutines, or short sections of code. The most direct application of this study is in the design of programs that function under a time constraint. For example, high-resolution graphics operations, such as image rotation, filtering, and motion simulation, require all processing to be completed within a very short period of time (usually a few milliseconds or less). If analysis of the total instruction execution time for the graphics routine exceeds the allowed time of the system, a loss in image quality will most likely result. We will not get quite so involved with our analysis of the instruction times. Instead, we will look at one example subroutine and how its total execution time may be determined. We will use the 8088 microprocessor instruction times, and then examine the differences in the Pentium's execution of the same code.

### Instruction Cycle Analysis

TOBIN is a subroutine that will convert a 4-digit BCD number in register BX into a binary number. The result is returned by TOBIN in AX. TOBIN is a good example to use for execution time determination because it contains two nested loops. The number of clock cycles for each instruction can be determined by referring to Appendix A. Table 7.1 is an example of how clock cycles are determined. The number of clock cycles an instruction takes to execute depends on a number of factors. Operand size is the first variable. Look at the clock cycles required by the MOV CX instructions. The instruction itself takes four



**TABLE 7.1** Required instruction execution clock cycles in a simple programming loop

|            |                     |           | <b>Clock Cycles</b>    |                          |                          |
|------------|---------------------|-----------|------------------------|--------------------------|--------------------------|
|            | <i>Instructions</i> |           | <i>Overhead Cycles</i> | <i>Outer-Loop Cycles</i> | <i>Inner-Loop Cycles</i> |
| TOBIN      | PROC                | FAR       |                        |                          |                          |
|            | SUB                 | AX,AX     | 3                      |                          |                          |
|            | MOV                 | DX,AX     | 2                      |                          |                          |
|            | MOV                 | CX,4      | 4 + 4                  |                          |                          |
| NEXTDIGIT: | PUSH                | CX        |                        | 11 + 4                   |                          |
|            | SUB                 | BP,BP     |                        | 3                        |                          |
|            | MOV                 | CX,4      |                        | 4 + 4                    |                          |
| GETNUM:    | RCL                 | BX,1      |                        |                          | 2                        |
|            | RCL                 | BP,1      |                        |                          | 2                        |
|            | LOOP                | GETNUM    |                        |                          | 17/5                     |
|            | MOV                 | CX,10     |                        | 4 + 4                    |                          |
|            | MUL                 | CX        |                        | 118-133                  |                          |
|            | ADD                 | AX,BP     |                        | 3                        |                          |
|            | POP                 | CX        |                        | 8 + 4                    |                          |
|            | LOOP                | NEXTDIGIT |                        | 17/5                     |                          |
|            | RET                 |           | 18 + 8                 |                          |                          |
|            | ENDP                |           |                        |                          |                          |
| TOBIN      |                     |           |                        |                          |                          |

cycles, plus another four required by the 8088 to fetch a word operand from memory. The processor's 8-bit data bus requires two memory-read cycles to obtain the word operand, resulting in the additional four clock cycles.

The addressing mode used by an instruction also affects the number of clock cycles required. Register addressing, as in the SUB AX,AX and ADD AX,BP instructions, requires fewer clock cycles than an instruction that must access memory during its execution, such as PUSH and POP. The clock cycles required for each addressing mode are included in Appendix A. You must note that the clock cycles shown for each instruction assume that the instructions have already been fetched and placed in the instruction queue. This, in many cases, results in very few clock cycles for some instructions. An exception is the LOOP instruction, which has two times listed. The smaller time is used when the jump does *not* take place. This makes sense, because the instruction following LOOP can simply be fetched from the instruction queue. Greater time is required when the LOOP does take place, because the queue must be flushed and reloaded. Other instructions have a variable number of cycles as a function of the data on which they operate. MUL is a good example of this, requiring anywhere from 118 to 133 cycles depending on the number of 1s in the operands forming the product. It is best to use the *worst case* execution time. Then you will avoid nasty situations such as having to explain why the routine does not always execute at the same speed.

The RET instruction requires an ample number of clock cycles to execute because it must pop the CS and IP return address off the stack. These operations require accesses to memory, which always increase the execution time.

Table 7.1 has three columns of clock cycles. The **overhead** column is for instructions that execute only once in the subroutine. They do not contribute significantly to the overall time, but they cannot be ignored. The outer-loop and inner-loop cycles are repeated a number of times and, thus, grow into a large number of clock cycles before the routine completes.

Keep in mind that the overall execution time will be only an *estimate*, because other factors are normally present that affect execution time. These factors include the time required to initially load the instruction queue, or the time to reload it after a jump or LOOP instruction; however, our estimate will be within 10 percent of the actual execution time.

### Execution Time Calculation

To compute the execution time, we must first determine the total number of clock cycles needed. The inner loop requires 21 cycles (worst case) for one pass. Because the inner loop is designed to execute four times, this gives 84 cycles for one completion of the inner-loop instructions. But these instructions are contained within the outer loop, which itself requires an additional 199 clock cycles (worst case). So, executing the outer loop once uses 283 clock cycles. The outer loop executes four times also, giving a total of 1,132 cycles so far. Adding in 39 overhead cycles (do you see why overhead is not significant?) results in a grand total of 1,171 clock cycles for the TOBIN subroutine.

How does this number translate into an execution time? Because each clock cycle has a period determined by its frequency, we need to know the clock frequency at which the processor is running. Suppose this frequency is 5 MHz. Each cycle will then have a period of 200 ns. Multiplying 200 ns by 1,171 gives 234.2  $\mu$ s! This is the execution time of TOBIN.

In conclusion, it is interesting to note that TOBIN can convert more than 4,200 BCD numbers to binary in 1 second.

### How the Pentium Does It

Table 7.2 lists the clock cycles required for each of TOBIN's instructions when executed on the Pentium. Note that most instructions on the Pentium require a *single clock cycle* to

**TABLE 7.2** Required instruction execution clock cycles on the Pentium

|            |              |           | Clock Cycles    |                   |                   |
|------------|--------------|-----------|-----------------|-------------------|-------------------|
|            | Instructions |           | Overhead Cycles | Outer-Loop Cycles | Inner-Loop Cycles |
| TOBIN      | PROC         | FAR       |                 |                   |                   |
|            | SUB          | AX,AX     | 1               |                   |                   |
|            | MOV          | DX,AX     | 1               |                   |                   |
|            | MOV          | CX,4      | 1               |                   |                   |
| NEXTDIGIT: | PUSH         | CX        |                 | 1                 |                   |
|            | SUB          | BP,BP     |                 | 1                 |                   |
|            | MOV          | CX,4      |                 | 1                 |                   |
| GETNUM:    | RCL          | BX,1      |                 |                   | 1                 |
|            | RCL          | BP,1      |                 |                   | 1                 |
|            | LOOP         | GETNUM    |                 |                   | 5/6               |
|            | MOV          | CX,10     |                 | 1                 |                   |
|            | MUL          | CX        |                 | 11                |                   |
|            | ADD          | AX,BP     |                 | 1                 |                   |
|            | POP          | CX        |                 | 1                 |                   |
|            | LOOP         | NEXTDIGIT |                 | 5/6               |                   |
|            | RET          |           | 2               |                   |                   |
|            | ENDP         |           |                 |                   |                   |
| TOBIN      |              |           |                 |                   |                   |

execute. This is a cornerstone of the RISC design philosophy, and is possible on the Pentium, thanks to a number of architectural features, such as instruction and data cache, instruction and address pipelining, and branch prediction.

MUL and LOOP are examples of instructions that take more than one instruction to execute. The task of multiplying large binary numbers cannot be performed in a single cycle without placing an enormous amount of logic gates in the instruction pipeline. This is impractical and may even require a slower clock on the processor to ensure that the logic gates function properly. So, instead, the multiplication is done in stages over eleven clock cycles.

As before, LOOP instructions have a *pair* of cycle times associated with their execution. The 5/6 cycle count for LOOP means that five clock cycles are used when the LOOP takes place (CX is not zero) and six when the LOOP does not take place (CX is zero). Other instructions share this feature, most notably the conditional jumps. In general, instructions that change the flow of execution in a program cause the instruction pipeline to **stall** for at least one additional cycle, as the pipeline changes its execution path. The RET instruction at the end of TOBIN falls into this category.

Several other factors contribute to a small percentage of uncertainty about the exact number of clock cycles required on a Pentium architecture, such as cache performance, operating system overhead (page faults, task switching), and the sequence of instructions being executed. It is still possible to get a good estimate of the number of cycles using the techniques developed for the 8088's analysis.

---

## 7.5 WORKING WITH INTERRUPT VECTORS

Recall from Chapter 5 that an interrupt is an event that causes the processor to stop its current program execution and perform a specific task to service the interrupt. In a PC, interrupts are used to keep accurate time, read the keyboard, operate the disk drives, and access the power of the disk operating system. DOS assigns a number of these interrupts (INT 21H, for example) for its own use, so we must be careful when accessing the interrupt vector table. When it is necessary to reassign an interrupt vector, the following functions should be used.

### DOS INT 21H, Function 35H: Get Interrupt Vector

This function is selected when the AH register equals 35H. Upon entry, the interrupt number (0 to 255) must be in register AL. Upon exit, register BX contains the interrupt handler IP address, and the ES register contains the interrupt handler CS address.

---

■ **EXAMPLE 7.4** The following instructions can be used in DEBUG to obtain the vector address of INT 21H:

```
MOV AL, 21
MOV AH, 35
INT 21
```

When these instructions are traced, we get:

```
-p
AX=0021    BX=0000    CX=0000    DX=0000    SP=FFEE    BP=0000    SI=0000    DI=0000
DS=1C46    ES=1C46    SS=1C46    CS=1C46    IP=0102    NV UP EI PL NZ NA PO NC
1C46:0102 B435                MOV     AH, 35
-p
```

```

AX=3521    BX=0000    CX=0000    DX=0000    SP=FFEE    BP=0000    SI=0000    DI=0000
DS=1C46    ES=1C46    SS=1C46    CS=1C46    IP=0104    NV UP EI PL NZ NA PO NC
1C46:0104  CD21                INT      21
-p
AX=3521    BX=16B4    CX=0000    DX=0000    SP=FFEE    BP=0000    SI=0000    DI=0000
DS=1C46    ES=0BF4    SS=1C46    CS=1C46    IP=0106    NV UP EI PL NZ NA PO NC
1C46:0106  AE                SCASB

```

The resulting addresses in registers BX and ES give a vector address of 0BF4:16B4. ■

## DOS INT 21H, Function 25H: Set Interrupt Vector

This function is selected when the AH register is loaded with 25H. Upon entry, the interrupt number (0 to 255) must be loaded into register AL. The CS:IP vector address of the new interrupt handler must be loaded into the DS and DX registers, respectively.

**Attention!** Even experienced programmers may get unexpected results when an interrupt vector is replaced. Usually, if an interrupt vector in a DOS machine is set improperly or does not have a valid handler at the new vector address, the machine is in danger of crashing. Be careful when using this function.

In Sections 7.6 and 7.9 you will examine applications designed to take over certain DOS/BIOS interrupt functions.

## VECTORS: View Entire Interrupt Vector Table

The VECTORS program shown here displays all 256 vector addresses, as initialized by BIOS, DOS, and any other programs using them. A total of sixty-four vector addresses are displayed at a time, in the familiar CS:IP address format.

```

;Program VECTORS.ASM: View entire interrupt vector table.
;
.MODEL SMALL
.DATA
WMSG DB 0DH,0AH, 'Hit any key to continue...',0DH,0AH,0DH,0AH,'$'

.CODE
.STARTUP
SUB AL,AL ;begin with vector 0
NEWV: PUSH AX ;save vector number
CALL DISPHEX ;display it
MOV DL,' ' ;load blank character
MOV AH,2 ;display character function
INT 21H ;DOS call
MOV DL,'=' ;load equal sign
INT 21H ;and output
MOV DL,' ' ;load another blank
INT 21H ;and output
POP AX ;get vector number back
PUSH AX ;and save it again
MOV AH,35H ;get interrupt vector function
INT 21H ;DOS call
MOV AX,ES ;get interrupt segment
CALL DISPHEX_16 ;display it
MOV DL,':' ;load colon character
MOV AH,2 ;display character function
INT 21H ;DOS call

```

```

HEXOUT  PROC NEAR
        CMP  AL,10          ;is AL greater than 10?
        JC   NHA1          ;yes
        ADD  AL,7           ;no, add alpha bias
NHA1:   ADD  AL,30H         ;add ASCII bias
        MOV  DL,AL          ;load output character
        MOV  AH,2          ;display character function
        INT  21H           ;DOS call
        RET
HEXOUT  ENDP
END

```

The program uses DOS INT 21H, Function 35H to read each vector, one at a time, from 0 to 255.

A sample execution showing the first 64 vectors is as follows:

|                |                       |                |                |
|----------------|-----------------------|----------------|----------------|
| 00 = 011C:108A | 01 = 0070:06F4        | 02 = 09B4:0016 | 03 = 0070:06F4 |
| 04 = 0070:06F4 | 05 = 1587:04E4        | 06 = F000:E14C | 07 = F000:EF6F |
| 08 = 0BF4:1875 | 09 = 0BF4:1923        | 0A = F000:EF6F | 0B = 09B4:006F |
| 0C = F000:EF6F | 0D = F000:EF6F        | 0E = 09B4:00B7 | 0F = 0070:06F4 |
| 10 = 0BF4:18B3 | 11 = F000:F84D        | 12 = F000:F841 | 13 = 0BF4:18C5 |
| 14 = F000:E739 | 15 = 0BF4:19A0        | 16 = F000:E82E | 17 = F000:EFD2 |
| 18 = F000:E3D0 | 19 = 0BF4:1990        | 1A = F000:FE6E | 1B = 0070:06EE |
| 1C = F000:FF53 | 1D = F000:F0A4        | 1E = 0000:0522 | 1F = C000:5B5F |
| 20 = 011C:1094 | <b>21 = 0BF4:16B4</b> | 22 = 0A76:02B1 | 23 = 0A76:014A |
| 24 = 0A76:0155 | 25 = 0BF4:19DE        | 26 = 0BF4:1A27 | 27 = 011C:10BC |
| 28 = 1587:02E8 | 29 = 0070:0762        | 2A = 011C:10DA | 2B = 011C:10DA |
| 2C = 011C:10DA | 2D = 011C:10DA        | 2E = 0A76:013F | 2F = 0F3A:306F |
| 30 = 1C10:D0EA | 31 = F000:FF01        | 32 = 011C:10DA | 33 = 011C:10DA |
| 34 = 011C:10DA | 35 = 011C:10DA        | 36 = 011C:10DA | 37 = 011C:10DA |
| 38 = 011C:10DA | 39 = 011C:10DA        | 3A = 011C:10DA | 3B = 011C:10DA |
| 3C = 011C:10DA | 3D = 011C:10DA        | 3E = 011C:10DA | 3F = 011C:10DA |

Hit any key to continue...

The vector address for INT 21H is highlighted in bold. Note that the actual addresses on *your* computer probably will be different. The vector addresses change with each new version of DOS, and with the configuration specified at boot time via CONFIG.SYS and AUTOEXEC.BAT.

Examine the entire output of the VECTORS program. Are there any interrupts that contain vector address 0000:0000? There should be many of them, all indicating unused interrupts in the BIOS/DOS environment.

---

**Programming Exercise 7.6:** Modify VECTORS so that the effective address of each interrupt routine is displayed. For example, a vector address of 011C:10DA has an effective address of 0229A.

---

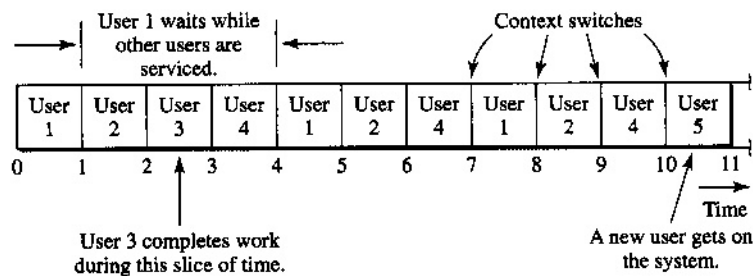
## 7.6 MULTITASKING

In an ever-increasing effort to squeeze the most processing power out of the basic micro-processor, individuals have come up with ingenious techniques for getting a single CPU to do many wonderful things. Consider a standard, single-CPU microcomputing system. One

user sits at a terminal entering commands, thinking, entering more commands, waiting for I/O from the system (as a file is loaded in from disk or tape), and so on and on. When computer specialists discovered that in this situation the CPU was wasting a great deal of time doing I/O, they thought of a way to get more use out of the CPU. Suppose that circuitry could be added to the system to perform the I/O operations under the CPU's control. All the CPU would have to do is issue a command, such as "read the disk," and the disk controller would do the rest of the work. This would free up the processor for other things while the disk controller was busy reading the disk. What other things can the CPU do? The most obvious answer that came to mind was *service another user!* Thus, the age of multitasking was born. A microprocessor system capable of performing multitasking is able to communicate with several users, seemingly at once. Each user believes he or she is the only user on the system. What is actually happening is that one user gets a small slice of the processor's time, then another user gets another time slice, and the same goes for all other users. Figure 7.1 shows a simple diagram of this operation. In this figure, we see that up to four users are executing their programs, seemingly at the same time. A single CPU can support more than one person at a time because of the high processing speed of the processor versus the slow thinking speed of the users. It is not difficult to see that the user will spend long periods of time thinking about what to do next or waiting for I/O to appear on the terminal. The time slice allocated to each user is designed so that it is long enough to perform a significant number of instructions without being so long as to be noticed by the user. For example, suppose that as many as sixteen users may be on the system at once. If the CPU must service each user once every 1/10th of a second, the time slice required is 6.25 ms, possibly enough to complete a user's current program. So, even though all users are forced to wait while the CPU services each of them (as shown in Figure 7.1), they probably do not even notice.

The software involved in supporting a multitasking system can become very involved, so we will examine only the basic details here. What exactly happens when one user's program is suspended so that the CPU can service a different user? Suppose that both users have written programs that use some of the same processor registers (for example, AX, BX, and SI are used by both programs). It becomes necessary to save one user's registers before letting the other user's program take over. This is referred to as a context switch. A context switch is used to save all registers for one user and load all registers used by the next user. Thus, a context switch is needed every time the CPU switches users. The routine presented here, TSLICE, will handle context switches for four different users in a round-robin fashion. This means that user 1 will not execute again until all other users have had their chance. To get the processor to do a context switch, we have to inform it that the user's time slice has expired. The easiest way to do this is to periodically interrupt the

**FIGURE 7.1** Multitasking with a single CPU



processor (by connecting a timer circuit to the processor's interrupt input). TSLICE is then actually an interrupt handler that is executed when a time slice is used up. TSLICE first saves all processor registers on the stack (the program counter and flags are already there, thanks to the processor's interrupt handling scheme), and then finds the next user who should execute. The registers for this user are then loaded from its stack area and execution resumes. This discussion assumes that all user programs remain in memory when not executing.

```

USER1      DW      64 DUP (?)      ;user stack areas
USER2      DW      64 DUP (?)
USER3      DW      64 DUP (?)
USER4      DW      64 DUP (?)
WHO        DB      ?                ;current user number (1-4)
STACK1     DW      ?                ;storage for user stack pointers
STACK2     DW      ?
STACK3     DW      ?
STACK4     DW      ?
.
.
.
TSLICE:    PUSH     AX                ;save all registers
           PUSH     BX
           PUSH     CX
           PUSH     DX
           PUSH     SI
           PUSH     DI
           PUSH     BP
           PUSH     DS
           PUSH     ES
           MOV      AX,DATA           ;load system data segment
           MOV      DS,AX
           MOV      AL,WHO           ;get user number
           DEC      AL               ;need 0-3 user number for indexing
           CBW                      ;extend into 16 bits
           ADD      AX,AX             ;double accumulator
           MOV      DI,AX             ;load index
           LEA      BP,STACK1         ;point to stack pointer table
           MOV      (BP + DI),SP      ;save user stack pointer
           INC      WHO               ;increment user number
           ADD      DI,2              ;point to next stack pointer
           CMP      DI,8              ;wrap around to user 1?
           JNZ      GETSTACK
           MOV      WHO,1             ;reset user number
           MOV      DI,0              ;and index register
GETSTACK:  MOV      SP,[BP + DI]      ;load new stack pointer
           POP      ES                ;restore new user's registers
           POP      DS
           POP      BP
           POP      DI
           POP      SI
           POP      DX
           POP      CX
           POP      BX
           POP      AX
           IRET                      ;go service new user

```

One requirement of TSLICE is that all users use the same stack segment. This guarantees that all PUSHes, CALLs, and interrupts generated by any user will place data within memory governed by the system software. The complexity of this code results from the need to wrap around the buffer containing the stack pointers, when the context switch is from user 4 to user 1. In Chapter 14, we will see how multitasking is built into protected mode, and how multiple tasks are managed.

## BIOS INT 1CH: Timer Tick

This interrupt is called by BIOS 18.2 times each second, the rate at which the computer's time-of-day clock is serviced. To run a program in *background* (such as PRINT or DOSKEY) or to switch between programs in an orderly fashion, this interrupt must be *hooked*. DOS interrupts 25H and 35H can be used to hook an interrupt. Any application that wants to run in background may use INT 1CH to install itself in the interrupt chain for Timer Tick. One note of caution: because INT 1CH is issued 18.2 times per second, only 1/18.2, or 54.94 milliseconds, is allowed for all programs sharing INT 1CH. Thus, the tasks performed by each program hooked into the Timer Tick interrupt should execute quickly. The specific details of hooking an interrupt are covered in Section 7.9.

It is also very important to preserve the state of all processor registers to ensure that the program interrupted by INT 1CH resumes normally.

## SWITCHER: A Simple Task-Switching Application

The SWITCHER program presented here allows two individual procedures (tasks) to run at different rates of execution. There is a significant difference between the SWITCHER program and the TSLICE procedure. TSLICE assumes that programs have been interrupted *during* execution, and thus saves all information on the current stack and then switches stack areas to restart a suspended process. SWITCHER operates differently by assuming that the individual tasks *complete* execution between successive interrupts. This allows the complicated stack switching to be eliminated.

; Program SWITCHER.ASM: Automatically switch between two simple tasks.

```
;
        .MODEL SMALL
        .CODE
TASK     DB      0           ;task number
COUNT  DB      '0'         ;initial count
T1KNT    DB      4           ;count rate value
ALPHA    DB      'A'         ;initial alpha
T2KNT    DB      12          ;alpha rate value
OLDIPCS  DD      ?           ;old interrupt address

        .STARTUP
SUB      AH,AH               ;set video mode function
MOV      AL,3                ;25 by 80 color
INT      10H                 ;BIOS call
MOV      AH,35H              ;get interrupt vector function
MOV      AL,1CH               ;timer interrupt number
INT      21H                 ;DOS call
MOV      WORD PTR CS:OLDIPCS,BX ;save old IP
MOV      WORD PTR CS:OLDIPCS[2],ES ;save old CS
MOV      AX,CS                ;load current CS value
MOV      DS,AX
```



```

        LEA DX,CS:SWAP      ;load address of new ISR
        MOV AH,25H         ;set interrupt vector function
        MOV AL,1CH         ;timer interrupt number
        INT 21H            ;DOS call
WAIT4KY: MOV AH,1          ;read keyboard status function
        INT 16H            ;BIOS call
        JZ WAIT4KY         ;loop until any key is pressed
        MOV DX,WORD PTR CS:OLDIPCS ;load old interrupt IP
        MOV DS,WORD PTR CS:OLDIPCS[2] ;load old interrupt CS
        MOV AL,1CH         ;timer interrupt number
        MOV AH,25H         ;set interrupt vector function
        INT 21H            ;DOS call
        MOV AH,1          ;read keyboard function
        INT 21H            ;DOS call
        .EXIT

SWAP:   PUSHF              ;save registers used here
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH DX
        NOT CS:TASK        ;switch tasks
        CMP CS: TASK,0     ;task 2's turn now?
        JNZ ET2
        CALL TASK1         ;perform task 1
        JMP BYE            ;go resume interrupt chain
ET2:    CALL TASK2         ;perform task 2
BYE:    POP DX             ;restore registers
        POP CX
        POP BX
        POP AX
        POPF
        JMP CS:OLDIPCS     ;go execute old ISR

TASK1   PROC NEAR
        DEC CS:T1KNT       ;decrement rate value
        JNZ ER1            ;exit if task is still asleep
        MOV CS:T1KNT,4     ;else, reload rate value
        MOV AH,2           ;set cursor position function
        MOV BH,0           ;display page 0
        MOV DH,12          ;row 12
        MOV DL,37          ;column 37
        INT 10H            ;BIOS call
        MOV AL,CS:COUNT  ;load count value
        MOV BL,2           ;color is green
        MOV CX,1           ;write one character
        MOV AH,9           ;write character/attribute function
        INT 10H            ;BIOS call
        INC CS:COUNT      ;increment count value
        CMP CS:COUNT,'9'+1 ;check for wrap around
        JNZ ER1
        MOV CS:COUNT,'0'  ;set initial count value
ER1:    RET
TASK1   ENDP

```

```

TASK2    PROC NEAR
        DEC CS:T2KNT          ;decrement rate value
        JNZ ER2              ;exit if task is still asleep
        MOV CS:T2KNT,12      ;else, reload rate value
        MOV AH,2             ;set cursor position function
        MOV BH,0             ;display page 0
        MOV DH,12            ;row 12
        MOV DL,43            ;column 43
        INT 10H              ;BIOS call
        MOV AL,CS:ALPHA      ;load alpha value
        MOV BL,1             ;color is blue
        MOV CX,1             ;write one character
        MOV AH,9             ;write character/attribute function
        INT 10H              ;BIOS call
        INC CS:ALPHA         ;increment alpha value
        CMP CS:ALPHA,'Z'+1   ;check for wrap around
        JNZ ER2
        MOV CS:ALPHA,'A'     ;set initial alpha value
ER2:     RET
TASK2    ENDP

        END

```

The first task in SWITCHER displays a running count from 0 to 9, in green, on the screen. A short delay is added between outputs.

The second task displays the alphabet, from A to Z, in blue on the screen also, with a much longer time delay between outputs. The time between outputs in both tasks is controlled by counting the number of Timer Tick interrupts that have occurred. Task 1 executes its display code every 4 ticks. Task 2 executes its display code every 12 ticks. The interrupt service routine SWAP uses the TASK byte to determine which task should execute. SWAP complements the TASK byte with each Timer Tick, causing alternate calls to TASK1 and TASK2. SWAP preserves the state of all registers used in the TASK procedures and jumps to the address of the old interrupt service routine for INT 1CH when it completes execution.

Any keystroke will cause the program to exit to DOS. SWITCHER will restore the old interrupt vector and return to DOS.

---

**Programming Exercise 7.7:** Modify the SWITCHER program so that four tasks are switched. The two new tasks should count *down* from 9 to 0 and from Z to A.

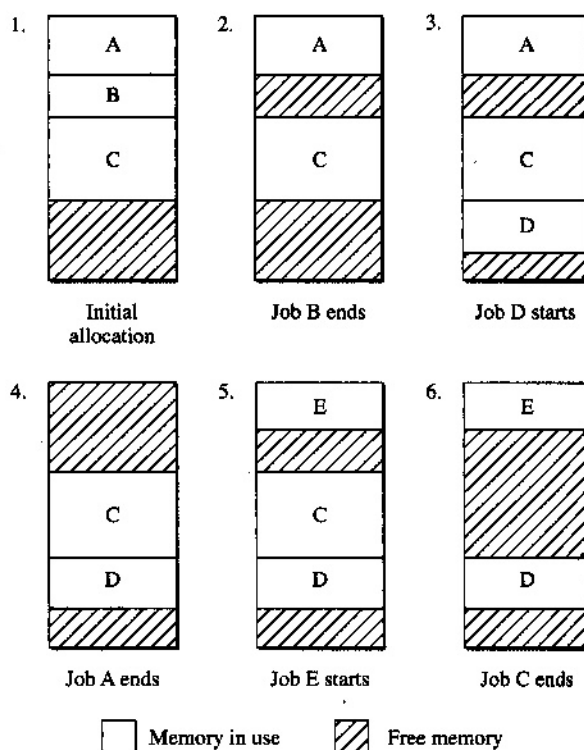
---

**Programming Exercise 7.8:** Modify the SWITCHER program so that both TASK procedures are installed in the chain for INT 1CH. That is, eliminate the SWAP code and have INT 1CH process both TASK procedures for every Timer Tick. It will be necessary to set the interrupt vector twice to accomplish this.

---

## 7.7 MEMORY MANAGEMENT

In this section, we will see how entire blocks of memory can be assigned through the use of a **memory management** routine. This is a required feature in all operating systems that load multiple programs into memory for shared execution (as in multitasking). We can

**FIGURE 7.2** Memory allocation in an operating system

easily see why memory management is needed by examining Figure 7.2. In this figure, the memory space of a typical system is examined. Initially, three jobs (A, B, and C) are running (1). Because they were the first three to begin execution, all were assigned consecutive blocks of memory. In (2), we see that job B (or program B) terminates. The memory allocated to job B is returned to the storage pool. In (3), job D begins execution. Because the memory required by this job exceeded what was available in the area vacated by job B, this job is loaded into the first available space that is big enough. When job A ends in (4), its space is also returned to the storage pool. Job E quickly takes this space over in (5). Finally, job C terminates in (6).

The purpose of the memory manager is to keep track of all free blocks of memory. When the operating system needs to load a new job into memory, it will inform the memory manager of how much memory is needed by the new job. The memory manager will either find a big enough space and return the starting address of the block, or indicate that not enough memory is available for the job to execute at the present time. In this case, the job will have to wait until more memory becomes available.

There are three built-in DOS functions explicitly designed to help programs manage memory in the ways just described.

### DOS INT 21H, Function 48H: Allocate Memory

This function is selected when register AH equals 48H. Upon entry to the interrupt, the BX register must contain the number of *paragraphs* requested for allocation. DOS allocates memory in chunks of sixteen locations called paragraphs, so a 1,024-byte storage block is composed of 1,024/16, or 64 paragraphs.

If DOS is able to allocate the requested number of paragraphs, it will return the starting segment address of the memory block in register AX. If DOS cannot allocate enough memory, the carry flag will be set and register AX will contain error code information. The BX register will contain the number of paragraphs in the largest block of available memory.

---

**■ EXAMPLE 7.5**

Show the code required to attempt allocation of a 32KB block of storage. If successful, load the ES register with the allocation block segment address. If unsuccessful, jump to NO\_MEM.

A 32KB block of memory requires 2,048 paragraphs. Thus, we have the following request code:

```
MOV  AH, 48H
MOV  BX, 2048
INT  21H
JC   NO_MEM
MOV  ES, AX
```

Offset 0 in the extra segment is the first location in the newly allocated 32KB block. ■

---

### DOS INT 21H; Function 49H: Free Allocated Memory

This function is selected when the AH register is loaded with 49H. Upon entry, the ES register must contain the segment address of the block being freed. If the carry flag is cleared upon return, the memory was freed successfully. If the carry flag is set, register AX will contain the error code.

This function *must* be used with a segment address previously obtained by a call to function 48H.

### DOS INT 21H; Function 4AH: Modify Allocated Memory Blocks

This function is selected when the AH register equals 4AH. Upon entry, the ES register must contain the segment address of the block being modified. The BX register must contain the number of paragraphs that will *remain* after reallocation.

As with the other functions, the operation is successful if the carry flag is clear upon return. Otherwise, register AX contains an error code, and register BX contains the maximum number of paragraphs available for an increase in allocation.

---

**■ EXAMPLE 7.6**

A modify request is generated as follows:

```
MOV  AH, 4AH
MOV  BX, 200H
INT  21H
```

Upon return, the carry flag is set and the BX register equals 118H. What does this mean?

Because the carry flag is set, we know that DOS was not able to reallocate memory. The value in register BX indicates that more memory was requested (200H paragraphs) than is available (118H paragraphs). ■

---

## MEMMAN: A Memory Usage Application

The MEMMAN program presented here is structured as a .COM file for allocation purposes. When DOS executes a .COM program, it loads the program into memory (at some initial allocation segment address), and then allocates *all available* memory to the program. If it is necessary for the .COM program to allocate and free memory blocks, it must first modify its own memory allocation to return memory blocks to DOS. Notice the EOP label at the end of the MEMMAN program. The value of this label is used to calculate the length of the program's required memory space.

```
;Program MEMMAN.ASM: A memory-management demonstrator.
;Note: Program is written in .COM format (ORG at 100H) and
;must use the .TINY model.
;
        .MODEL TINY
        .CODE
        ORG     100H           ;code must be relative to 100H
START:   JMP     GO           ;jump over data area

ERR0     DB      'Error! Could not allocate memory.', 0DH,0AH,'$'
OK0      DB      '50K-bytes of memory has been allocated at segment $'
ERR1     DB      'Error! Could not modify allocated memory.', 0DH,0AH,'$'
OK1      DB      'Allocated memory has been modified.', 0DH,0AH,'$'
CRLF     DB      0DH,0AH,'$'
AMB      DW      64 DUP(?)

GO:      LEA     BX,EOP        ;get program size in bytes
        MOV     CL,4          ;shift count
        SHR     BX,CL         ;compute paragraph size
        INC     BX            ;round up to next paragraph
        MOV     AH,4AH        ;modify allocated memory function
        INT     21H           ;DOS call
        JNC     MM1          ;jump if modification successful
        LEA     DX,ERR1       ;set up pointer to not modified message
        MOV     AH,9          ;display string function
        INT     21H           ;DOS call
        JMP     EXIT

MM1:     LEA     DX,OK1        ;set up pointer to memory modified message
        MOV     AH,9          ;display string function
        INT     21H           ;DOS call
        SUB     DI,DI         ;clear index

MM3:     MOV     BX,3200       ;request 3200 paragraphs (50K bytes)
        MOV     AH,48H        ;allocate memory function
        INT     21H           ;DOS call
        JNC     MM2          ;jump if allocated
        LEA     DX,ERR0       ;set up pointer to not allocated message
        MOV     AH,9          ;display string function
        INT     21H           ;DOS call
        JMP     EXIT

MM2:     MOV     AX,AMB[DI],AX ;save segment address of new memory block
        LEA     DX,OK0        ;set up pointer to segment address message
        MOV     AH,9          ;display string function
        INT     21H           ;DOS call
        MOV     AX,AMB[DI]    ;load allocated segment address
```

```

        ADD     DI,2           ;advance pointer to allocation table
        CALL    DHEX16        ;display segment address of new block
        LEA     DX,CRLF       ;set up pointer to crlf string
        MOV     AH,9          ;display string function
        INT     21H           ;DOS call
        JMP     MM3           ;go try for another 50k-byte block
EXIT:    .EXIT

DHEX16 PROC NEAR
        PUSH    AX            ;save number
        MOV     AL,AH         ;get upper byte
        CALL    DHEX         ;convert and display
        POP     AX            ;get number back
        CALL    DHEX         ;convert and display lower byte
        RET
DHEX16 ENDP

DHEX PROC NEAR
        PUSH    AX            ;save value
        SHR     AL,1          ;shift down upper nybble
        SHR     AL,1
        SHR     AL,1
        SHR     AL,1
        CALL    DDIG         ;display hex digit
        POP     AX            ;get value back
        AND     AL,0FH        ;set lower nybble value
        CALL    DDIG         ;display hex digit
        RET
DHEX ENDP

DDIG     PROC NEAR
        CMP     AL,10         ;is AL less than 10?
        JC      NHA1          ;jump if yes
        ADD     AL,7           ;otherwise, add alpha bias
NHA1:    ADD     AL,30H        ;add ASCII bias
        MOV     DL,AL          ;load character for output
        MOV     AH,2          ;display character function
        INT     21H           ;DOS call
        RET
DDIG     ENDP

EOP:     NOP                  ;end of program code, used
                                ;for length calculations
        END     START         ;must specify starting address

```

When DOS launches a .COM file, the values of all four segment registers are identical and equal to the load address of the program segment prefix. An example DEBUG session shows MEMMAN's initial segment register addresses:

```

C:\>DEBUG MEMMAN.COM
-r
AX=0000 BX=0000 CX=01B9 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=1C64 ES=1C64 SS=1C64 CS=1C64 IP=0100 NV UP EI PL NZ NA PO NC
1C64:0100 E92C01 JMP 022F
-q

```

Notice that all segment registers contain 1C64. This is the segment of the initial allocation block for the MEMMAN.COM program. This segment address must be used when MEMMAN attempts to change its allocation through function 4AH. When making the current allocation smaller, MEMMAN must be careful to save enough memory (in paragraphs) for its code requirements. By using the address value of the EOP label, MEMMAN is able to determine the correct number of required paragraphs.

Assuming that MEMMAN is able to return memory, it then attempts to request as many 50KB blocks as it can. The segment address of each successful allocation is displayed, as shown in the following sample execution:

```
C:\>MEMMAN
Allocated memory has been modified.
50K-bytes of memory has been allocated at segment 16CB
50K-bytes of memory has been allocated at segment 234C
50K-bytes of memory has been allocated at segment 2FCD
50K-bytes of memory has been allocated at segment 3C4E
50K-bytes of memory has been allocated at segment 48CF
50K-bytes of memory has been allocated at segment 5550
50K-bytes of memory has been allocated at segment 61D1
50K-bytes of memory has been allocated at segment 6E52
50K-bytes of memory has been allocated at segment 7AD3
50K-bytes of memory has been allocated at segment 8754
Error! Could not allocate memory.
```

The program was able to allocate ten 50KB blocks before DOS ran out of memory. To get a handle on the segment addresses returned, notice that they increase in value as the allocations progress. If we subtract any two consecutive segment addresses, we will get the block size in paragraphs. To check,  $6E52 - 61D1 = 0C81$ , which corresponds to 3,201 paragraphs. A 50KB block requires 3,200 paragraphs. The extra paragraph allocated is used by DOS to maintain allocation information.

It may be interesting to watch what happens when MEMMAN is executed when you have *shelled to DOS* from another application. For instance, after starting up WordPerfect® 5.1 and shelling to DOS via Control-F1, MEMMAN gives the following result:

```
C:\WP51\>MEMMAN
Allocated memory has been modified.
50K-bytes of memory has been allocated at segment 79A6
50K-bytes of memory has been allocated at segment 8627
50K-bytes of memory has been allocated at segment 92A8
Error! Could not allocate memory.
```

The program is now only able to allocate three 50KB blocks, because the rest of memory is already allocated to the suspended WordPerfect program.

---

**Programming Exercise 7.9:** Modify the MEMMAN program to allocate blocks of 4,095 paragraphs. What do you notice about the segment addresses displayed?

---

**Programming Exercise 7.10:** Write a memory allocation program that alternately requests 16KB, 32KB, and 64KB blocks and saves their segment addresses. When no more blocks can be allocated, return all 32KB blocks allocated, then request as many 8KB blocks as possible.

---

## 7.8 USING THE MOUSE

Anyone who has used a mouse to point and click has probably discovered that it is easy to handle. With a software mouse driver installed, the PC keyboard has some electronic competition for input services. It is often more convenient to use a mouse to navigate through a program than a keyboard. In this section, we will see how to incorporate mouse functions in our own programs.

Assuming that a MOUSE.SYS driver was loaded during boot time (or that MOUSE.COM has been executed), the mouse functions should be available through INT 33H. More than fifty individual mouse functions are provided through INT 33H. We will examine just a few of them in this section.

### MOUSE INT 33H, Function 00H: Mouse Reset and Status

This function is selected by loading 00H into register AH. The interrupt returns status information in registers AX and BX. If AX contains 0FFH upon return, the mouse was initialized. If AX contains 0, no mouse functions are available. Register BX contains the status of the mouse buttons. If the left button is pressed, bit-0 of BX will be set. If the right button is pressed, bit-1 will be set.

When the mouse is reset, a number of parameters are initialized. A few of these mouse parameters are as follows:

|                      |                      |
|----------------------|----------------------|
| Cursor position:     | Center of the screen |
| Cursor:              | Hidden               |
| Text Cursor:         | Inverse video box    |
| Graphics Cursor:     | An arrow             |
| Display page number: | 0                    |

If no mouse driver has been loaded, INT 33H usually will have a vector value of 0000:0000 or, if a vector exists, the only instruction in the interrupt service routine might be IRET. So, it is necessary to examine register AX after a mouse reset attempt.

---

#### ■ EXAMPLE 7.7

How can the presence of a mouse be tested? After performing a mouse reset function, register AX can be tested for 0 like so:

```
OR    AX,AX      ;set zero flag if AX = 0
JZ    NO_MOUSE   ;jump if no mouse found ■
```

---

### MOUSE INT 33H, Function 01H: Show Mouse Cursor

This function is selected when AH equals 01H. The mouse cursor is turned on and will move when the mouse is used. In text mode, the mouse cursor is an inverse video box the size of a single character. In graphics mode, the mouse cursor is a white arrow with a black outline.

### MOUSE INT 33H, Function 02H: Hide Mouse Cursor

This function is selected when AH is loaded with 02H. The mouse cursor is turned off. Because DOS does not automatically turn the mouse cursor off when a .COM or .EXE file terminates, this function must be used prior to exit.



**MOUSE INT 33H, Function 03H: Get Button Status and Mouse Position**

This function is selected when AH equals 03H. Upon return, the button status of the mouse is saved in register BX. The horizontal and vertical positions of the mouse are saved in registers CX and DX, respectively.

The lower 2 bits of register BX are used to represent the state of the left and right mouse buttons. If either button is pressed, its associated bit in register BX will be set. Bit-0 indicates left-button status and bit-1 indicates right-button status. Register CX will equal 0 when the mouse is on the left side of the screen. Register DX will equal 0 when the mouse is at the top of the screen. The maximum values of registers CX and DX depend on the video mode selected.

**■ EXAMPLE 7.8**

In this example, we see one way the mouse status can be read and analyzed. The instructions place a 1 in registers AL and AH, depending on the button status returned by INT 33H.

```
MOV AH,3      ;get mouse status/position function
INT 33H       ;MOUSE call
SUB AX,AX     ;clear result
SHR BX,1      ;shift left button bit into carry flag
ADC AL,0      ;adjust AL for left-button status
SHR BX,1      ;repeat for right-button status
ADC AH,0
```

The button status registers can be tested elsewhere with code like this:

```
OR AL,AL      ;is button pushed?
JZ NOT_PUSHED ■
```

**MOUSETST: A Mouse Application**

There are many more mouse functions available, but the four presented here should enable us to write a simple mouse-interactive program. The MOUSETST program shown here uses all four functions to control and access the mouse.

```
;Program MOUSETST.ASM: Display mouse's horizontal/vertical position.
;
```

```
        .MODEL SMALL
        .DATA
NODRV    DB      'No mouse driver installed.', 0DH,0AH,'$'
NOMSE    DB      'Mouse not responding.',0DH,0AH,'$'
MEXIT    DB      0DH,0AH, 'Press left mouse button to exit...', 0DH,0AH,'$'
SPOS     EQU     12*160

        .CODE
        .STARTUP
MOV      AH,35H      ;get interrupt vector function
MOV      AL,33H      ;INT 33H is for the mouse
INT      21H         ;DOS call
MOV      AX,ES       ;is vector 0000:0000?
OR       AX,BX
JNZ      GO
LEA      DX,NODRV    ;set up pointer to no-driver message
```

```

ERRD:  MOV  AH,9           ;display string function
        INT  21H          ;DOS call
        JMP  EXIT         ;exit to DOS
GO:     SUB  AX,AX         ;initialize mouse function
        INT  33H          ;MOUSE call
        OR   AX,AX        ;does mouse exist?
        JNZ  NEXT
        LEA  DX,NOMSE     ;set up pointer to no-mouse message
        JMP  ERRD         ;go process error
NEXT:   MOV  CX,25         ;set up loop counter
CLRSC:  MOV  DL,0AH        ;load line-feed code
        MOV  AH,2         ;display character function
        INT  21H          ;DOS call
        LOOP CLRSC
        LEA  DX,MEXIT     ;set up pointer to exit message
        MOV  AH,9         ;display string function
        INT  21H          ;DOS call
        MOV  AX,1         ;show mouse cursor function
        INT  33H          ;MOUSE call
        MOV  AX,0B800H    ;load address of video RAM segment
        MOV  DS,AX
        MOV  SI,SPOS      ;set up pointer to screen position
        MOV  BYTE PTR [SI+54],'X' ;write X: to screen
        MOV  BYTE PTR [SI+56],':'
        MOV  BYTE PTR [SI+94],'Y' ;write Y: to screen
        MOV  BYTE PTR [SI+96],':'
RDMSE:  MOV  AX,3          ;get status/position function
        INT  33H          ;MOUSE call
        MOV  AX,CX        ;read X position
        MOV  DI,SPOS+60   ;set up pointer to X's screen position
        CALL DISP         ;display number on screen
        MOV  AX,DX        ;read Y position
        MOV  DI,SPOS+100  ;set up pointer to Y's screen position
        CALL DISP         ;display number on screen
        CMP  BL,1         ;was left button pressed?
        JNZ  RDMSE       ;continue if not
        MOV  AX,2         ;hide mouse cursor function
        INT  33H          ;MOUSE call
EXIT:   .EXIT

DISP    PROC  NEAR
        PUSH CX           ;save CX
        MOV  CX,4         ;set up loop counter
NYBL:   ROL  AX,1          ;get upper nybble of AX
        ROL  AX,1
        ROL  AX,1
        ROL  AX,1
        PUSH AX           ;save AX
        AND  AL,0FH       ;mask out upper nybble
        ADD  AL,30H       ;add ASCII bias
        CMP  AL,3AH       ;are we greater than 9?
        JC   NO7          ;no, go save digit
        ADD  AL,7         ;correct to hex-alpha
NO7:    MOV  [DI],AL       ;save digit in memory
        POP  AX           ;get AX back

```

```

ADD    DI,2           ;advance memory pointer
LOOP   NYBL           ;repeat until done
POP    CX             ;get CX back
RET
DISP   ENDP
END

```

The initial group of instructions attempts to determine if a mouse driver is present. If so, the screen is cleared and a help message displayed, along with the mouse's horizontal (X) and vertical (Y) position, as in:

```
X: 0140 Y: 0060
```

The X and Y numbers will automatically be updated as the mouse is moved around. This is accomplished by converting the horizontal and vertical positions (in CX and DX) into four-digit integers and writing them directly to the screen's video RAM. The program exits to DOS when the left button is pressed.

---

**Programming Exercise 7.11:** Modify the MOUSEST program so that the keyboard can be used to show and hide the mouse. Hide the mouse if "H" is hit, and show the mouse if "S" is pressed.

---

**Programming Exercise 7.12:** Modify the MOUSEST program so that the program may also exit if "Q" is pressed on the keyboard.

---

**Programming Exercise 7.13:** Modify the MOUSEST program so that a *target* is displayed on the screen, such as [ \* ]. Exit to DOS only if the mouse button is pushed while the cursor is over the \*.

---

**Programming Exercise 7.14:** Write a program that will make the speaker beep whenever the mouse gets near the center of the screen.

---

## 7.9 WRITING A MEMORY-RESIDENT PROGRAM

A memory-resident program is a program that remains in memory after it returns control to DOS. Normally, when an .EXE or a .COM program completes execution, control is returned to DOS with the instructions:

```

MOV    AH,4CH         ;terminate program function
INT    21H            ;DOS call

```

which are generated by the .EXIT directive. This is, in fact, a DOS call that never returns. When this interrupt function is issued, DOS reclaims all memory allocated to the terminating program and starts up the command processor (COMMAND.COM) again. For a program to remain resident in memory (referred to as a TSR for **terminate and stay resident** program), a different function is needed.

### DOS INT 21H, Function 31H: Terminate and Stay Resident

This function is selected when register AH equals 31H. Upon entry, the DX register must contain the number of **paragraphs** that will remain resident. The number of paragraphs

must be large enough to hold the program segment prefix and all necessary code and data segments of the TSR.

■ **EXAMPLE 7.9** What is the memory size of the resident program after these instructions execute?

```
MOV  DX, 60
MOV  AH, 31H
INT  21H
```

The DX register specifies 60 paragraphs, which equals  $60 * 16 = 960$  bytes. ■

A second method used to terminate and stay resident, which should be used only with .COM files, is INT 27H. This interrupt requires that the DX register be loaded with the size of the program in *bytes*. INT 27H uses the current CS address to preserve the bytes indicated by DX.

Writing a terminate and stay resident program requires more than simply issuing Function 31H (or INT 27H). Some method must be used to give the program control after it terminates. Let us examine one common approach.

### Hooking an Interrupt

When a program terminates and stays resident, it must provide for a method of getting control so that it can execute. Otherwise, it will simply remain in memory, not executing, until the computer is rebooted.

One common method used to give control to memory-resident programs is through the use of an **interrupt hook**. An interrupt hook is accomplished through the use of DOS INT 21H, Functions 25H and 35H. A TSR program may use the following steps to hook an interrupt for itself:

1. Get vector of interrupt to hook (with INT 21H, Function 35H).
2. Save old interrupt vector.
3. Make a new vector for the interrupt.
4. Set new vector of hooked interrupt (with DOS INT 21H, Function 25H).

The new interrupt vector usually points to the address of a procedure contained within the memory-resident program. Figure 7.3 shows the result of an interrupt hook for INT 16H. The original vector for INT 16H (F000:2EE8) is read with Function 35H and saved within the data area of the TSR. The CS:IP address (1587:01CD) of the procedure within the TSR that will handle the interrupt replaces the old vector for INT 16H (using Function 25H). Thus, an interrupt *chain* has been created. An INT 16H will now cause the TSR to get control. Because the old vector of INT 16H is saved in the TSR's data area, the TSR can JMP or CALL the original INT 16H handler as required.

■ **EXAMPLE 7.10** Specifically, INT 16H may be hooked through the following instructions:

```
;storage for old interrupt vector
OLDIPCS DD ? ;need a double word for CS:IP
;save current vector address
MOV  AL,16H ;interrupt number
MOV  AH,35H ;get interrupt vector function
INT  21H ;DOS call
MOV  WORD PTR OLDIPCS,BX ;save INT 16H instruction pointer
```

```

MOV     WORD PTR OLDIPCS[2],ES    ;save INT 16H code segment
;set new vector address
MOV     AL,16H                    ;interrupt number
MOV     AH,25H                    ;set interrupt vector function
LEA     DX,NEWISR                  ;load IP of new interrupt service routine
PUSH    CS                        ;put copy of CS into DS
POP     DS
INT     21H                        ;DOS call

```

The code beginning at the address specified by NEWISR will get control whenever INT 16H is issued. ■

The TSR may give control to the old interrupt service routine in two ways: through the use of a FAR JMP or a FAR CALL. Considering the code from Example 7.10, a FAR JMP is performed by the instruction:

```
JMP     OLDIPCS
```

In this case, the IRET at the end of the original ISR will cause a return to the program that issued the interrupt.

To use a FAR CALL, it is necessary to adjust the processor's stack pointer. Remember that a FAR CALL pushes two words onto the stack, the CS:IP return address. The original ISR's IRET instruction will pop *three* words from the stack, the CS:IP return address and the flags. So, to use a FAR CALL within the TSR, we need the following code:

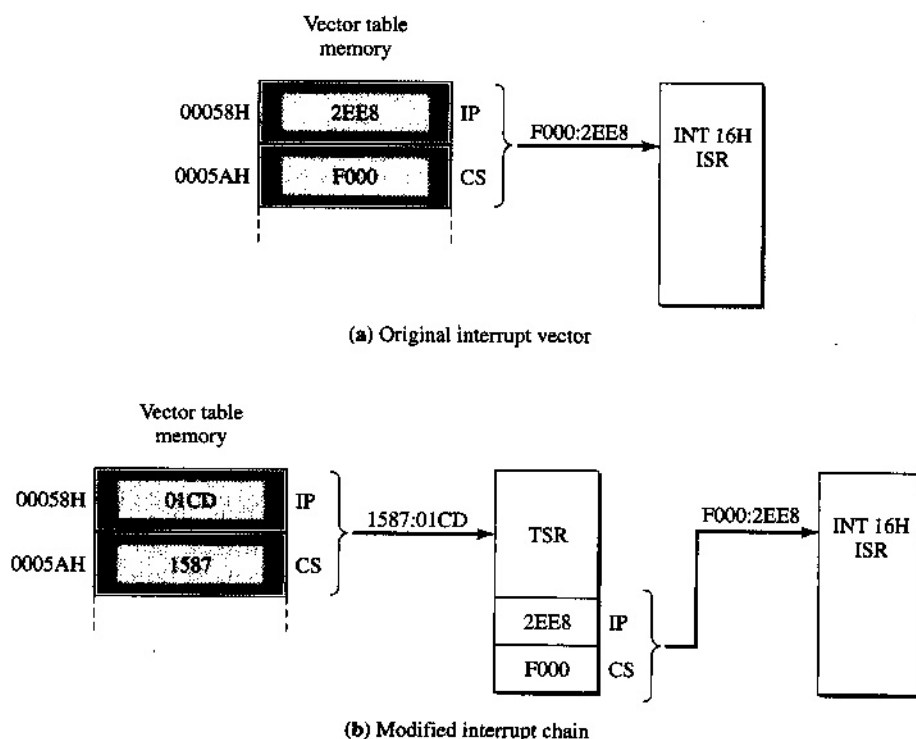
```

PUSHF                                ;adjust stack
CALL    OLDIPCS                       ;execute original ISR

```

Either way, it is necessary for the TSR to pass control to the original ISR at some point, or risk interfering with the normal operation of the computer.

**FIGURE 7.3** Hooking interrupt 16H: (a) original interrupt vector; (b) modified interrupt chain



## PHONES: A Pop-Up Telephone Pad

The PHONES program presented here installs itself as a memory-resident program and hooks INT 16H so that it can watch the keystrokes entered on the keyboard. When a special key is pressed (Alt-P), the TSR performs its intended task and writes a small telephone directory to the display. All other keyboard keys are ignored.

The telephone directory is output as a multiline white character on a blue background message. The screen information overwritten by the telephone directory is lost.

When Alt-P is entered (after PHONES has been installed as a TSR), the screen should look something like this:

```

Alan ..... 555 - 5017
Charlie ..... 555 - 5325
Dave ..... 555 - 7854
Michele..... 555 - 7514
Mike..... 555 - 7015

```

The telephone information will scroll off the screen line by line as Enter is used.

```

;Program PHONES.ASM: Memory-resident telephone pad.
;Using .COM format.
;
        .MODEL TINY
        .CODE
        ORG 100H ;needed for .COM structure
START:  JMP  INIT
OLDIPCS DD  ?
PMSG    DB  'Alan ..... 555 - 5017 ',1
        DB  'Charlie ..... 555 - 5325 ',1
        DB  'Dave ..... 555 - 7854 ',1
        DB  'Michele..... 555 - 7514 ',1
        DB  'Mike..... 555 - 7015 ',1
        DB  0
IPOS    EQU  20*2 ;initial screen position
SPOS    DW  IPOS
HOTKEY   EQU  1900H ;19H for ALT-P, 00H for extended key

POPUP    PROC  FAR
        PUSHF ;save flags
        CMP  AH,0 ;is function 00H requested?
        JNZ  EXTKEY ;no, go look for other function
        JMP  RDKEY ;go read keyboard
EXTKEY:  CMP  AH,10H ;is function 10H requested?
        JNZ  EXIT ;no, go process old ISR
RDKEY:   PUSHF ;fix stack for CALL
        CALL CS:OLDIPCS ;process old ISR (read keyboard)
        CMP  AX,HOTKEY ;does key match popup code?
        JNZ  BYE ;no, just exit now
        PUSH DI ;save registers
        PUSH SI
        PUSH DS
        MOV  AX,0B800H ;load address of video RAM
        MOV  DS,AX
        LEA  SI,CS:PMSG ;set up pointer to popup string
N1:      MOV  DI,CS:SPOS ;set up pointer to screen position
N2:      MOV  AL,CS:[SI] ;read popup character
        INC  SI ;advance to next character

```

```

        CMP     AL,1                ;was it end-of-line code?
        JZ      N3                 ;yes, go update position
        OR      AL,AL              ;was it end-of-string character?
        JZ      N4                 ;yes, go restore registers
        MOV     [DI],AL            ;otherwise, write character to screen
        MOV     BYTE PTR [DI+1],17H ;force white on blue attribute
        ADD     DI,2               ;advance to next screen position
        JMP     N2                 ;and repeat
N3:      ADD     CS:SPOS,160        ;advance screen position to next line
        JMP     N1                 ;and continue
N4:      MOV     CS:SPOS,IPOS       ;reset screen position
        POP     DS                 ;restore registers
        POP     SI
        POP     DI
        MOV     AX,3920H           ;return <space> for <Alt-p>
BYE:     POPF                     ;restore flags
        IRET                      ;return from interrupt
EXIT:    POPF                     ;restore flags
        JMP     CS:ODLIPCS         ;go process old interrupt
POPUP    ENDP

INIT:    MOV     AL,16H            ;keyboard interrupt number
        MOV     AH,35H            ;get interrupt vector function
        INT     21H               ;DOS call
        MOV     WORD PTR CS:OLDIPCS,BX ;save old interrupt IP
        MOV     WORD PTR CS:OLDIPCS[2],ES ;save old interrupt CS
        LEA     DX,CS:POPUP        ;load address of popup ISR
        MOV     AL,16H            ;keyboard interrupt number
        MOV     AH,25H            ;set interrupt vector function
        INT     21H               ;DOS call
        LEA     DX,CS:INIT         ;load required memory size in bytes
        MOV     CL,4              ;set up shift count
        SHR     DX,CL             ;convert size into paragraphs
        INC     DX                ;round up for good measure
        MOV     AH,31H            ;keep program resident function
        INT     21H               ;DOS call

        END     START             ;must specify starting address

```

The PHONES program is designed as a .COM program (via the .TINY directive) with all code *and* data contained within the code segment. This accounts for the operand references containing the segment override prefix CS, as in CS:[SI] and CS:OLDIPCS.

The INIT portion of the program is used to install the TSR and is not required after the interrupt chain has been modified. The instructions:

```

LEA     DX,CS:INIT
MOV     CL,4
SHR     DX,CL
INC     DX
MOV     AH,31H
INT     21H

```

are used to compute the length of the TSR in paragraphs, *up to* the INIT code.

We can examine the effects on allocated memory before and after execution of PHONES with DOS's MEM/C command. For example, before PHONES is executed,

MEM/C shows the following allocation:

Modules using memory below 1 MB:

| Name     | Total   |        | = | Conventional |        | + | Upper Memory |       |
|----------|---------|--------|---|--------------|--------|---|--------------|-------|
| MSDOS    | 17,565  | (17K)  |   | 17,565       | (17K)  |   | 0            | (0K)  |
| HIMEM    | 1,168   | (1K)   |   | 1,168        | (1K)   |   | 0            | (0K)  |
| EMM386   | 3,120   | (3K)   |   | 3,120        | (3K)   |   | 0            | (0K)  |
| COMMAND  | 2,928   | (3K)   |   | 2,928        | (3K)   |   | 0            | (0K)  |
| SMARTDRV | 27,504  | (27K)  |   | 11,104       | (11K)  |   | 16,400       | (16K) |
| MSCDEX   | 36,224  | (35K)  |   | 36,224       | (35K)  |   | 0            | (0K)  |
| SETVER   | 800     | (1K)   |   | 0            | (0K)   |   | 800          | (1K)  |
| MOUSE    | 15,808  | (15K)  |   | 0            | (0K)   |   | 15,808       | (15K) |
| MTMCDE   | 57,664  | (56K)  |   | 0            | (0K)   |   | 57,664       | (56K) |
| DBLSPACE | 39,648  | (39K)  |   | 0            | (0K)   |   | 39,648       | (39K) |
| DOSKEY   | 4,144   | (4K)   |   | 0            | (0K)   |   | 4,144        | (4K)  |
| Free     | 607,168 | (593K) |   | 583,008      | (569K) |   | 24,160       | (24K) |

When PHONES.COM is executed, we see the following change in memory allocation:

Modules using memory below 1 MB:

| Name          | Total      |             | = | Conventional |             | + | Upper Memory |             |
|---------------|------------|-------------|---|--------------|-------------|---|--------------|-------------|
| MSDOS         | 17,565     | (17K)       |   | 17,565       | (17K)       |   | 0            | (0K)        |
| HIMEM         | 1,168      | (1K)        |   | 1,168        | (1K)        |   | 0            | (0K)        |
| EMM386        | 3,120      | (3K)        |   | 3,120        | (3K)        |   | 0            | (0K)        |
| COMMAND       | 2,928      | (3K)        |   | 2,928        | (3K)        |   | 0            | (0K)        |
| SMARTDRV      | 27,504     | (27K)       |   | 11,104       | (11K)       |   | 16,400       | (16K)       |
| <b>PHONES</b> | <b>832</b> | <b>(1K)</b> |   | <b>832</b>   | <b>(1K)</b> |   | <b>0</b>     | <b>(0K)</b> |
| MSCDEX        | 36,224     | (35K)       |   | 36,224       | (35K)       |   | 0            | (0K)        |
| SETVER        | 800        | (1K)        |   | 0            | (0K)        |   | 800          | (1K)        |
| MOUSE         | 15,808     | (15K)       |   | 0            | (0K)        |   | 15,808       | (15K)       |
| MTMCDE        | 57,664     | (56K)       |   | 0            | (0K)        |   | 57,664       | (56K)       |
| DBLSPACE      | 39,648     | (39K)       |   | 0            | (0K)        |   | 39,648       | (39K)       |
| DOSKEY        | 4,144      | (4K)        |   | 0            | (0K)        |   | 4,144        | (4K)        |
| Free          | 606,336    | (592K)      |   | 582,176      | (569K)      |   | 24,160       | (24K)       |

In addition to the change in allocated memory, we should examine the differences in the interrupt vector table for INT 16H. The addresses for INT 16H vector are 00058H through 0005BH. These addresses can be examined using DEBUG, with the command:

```
-D 0:58 L 4
```

When this is done *before* PHONES is executed, we get:

```
E8 2E 00 F0
```

which indicates the address F000:2EE8. This is the original address of the INT 16H handler (somewhere in the system BIOS ROM).

After PHONES is executed, DEBUG will show the vector:

```
CD 01 87 15
```

which indicates the address 1587:01CD, the address of the POPUP code in PHONES.

So, we have proof that PHONES has been installed as a TSR program. Because it hooked into INT 16H, it will gain control frequently (every time a key is hit). Whenever Alt-P is pressed, the telephone information in the PMSG string will be copied to screen RAM. Notice that PMSG is not structured as a character string terminated by a "\$" character



(for use with Function 9: display string), but instead uses the number 1 to represent the end of a line of text, and a 0 to indicate the end of the list. A small group of instructions is used to copy the PMSG data to screen RAM and adjust the screen attributes.

**Attention!** It is extremely important to preserve the state of the processor registers used within the TSR! Only a small amount of stack space is provided by DOS for interrupt handling, so use and save registers sparingly.

If the TSR issues an INT of its own (possibly to an INT 21H function), there may not be enough stack space available to save everything. It may be necessary for the TSR to create its own stack environment while it has control. Then, before exiting, the original stack area must be reassigned.

---

**Programming Exercise 7.15:** Modify the PHONES program so that the screen text overwritten by the telephone directory is saved in memory and then restored when any key is hit after Alt-P.

---

**Programming Exercise 7.16:** Try to rewrite PHONES with a call to INT 21H, Function 09H to display the telephone directory. What problems, if any, do you encounter?

---

## 7.10 TICTAC: A GAME FOR A CHANGE

Writing a game program may not be considered an advanced programming assignment, but the effort still has its benefits. Many games are simple and fun to play and do not require a tremendous amount of programming experience to develop. Also, it is often easy to spot incorrect program operation when you have a good idea of what the program should be doing next. Debugging a game program usually offers many valuable programming lessons.

In this section, we will examine a simple tic-tac-toe program that does not take much effort to beat. In the human versus computer TICTAC.ASM program presented here, the strategy of the computer is straightforward: put an O in the first empty position encountered. It is difficult to lose a game against the computer due to this simple strategy.

Even though the computer's strategy lacks skill, writing the code for the entire program is still a challenge. The game can be broken down into a number of individual procedures. These procedures do the following:

- Display the game board
- Get a valid move from the user
- Determine a valid move for the computer
- Check to see who won

You may notice that other games, including checkers and chess, perform the same operations. It is very satisfying to write new game procedures one by one and watch the game develop into a playable program.

Examine the TICTAC.ASM program.

```
;Program TICTAC.ASM: Play Tic Tac Toe with the computer.
;
.MODEL SMALL
.DATA
MSG DB 'Computer TIC TAC TOE.',0DH,0AH
    DB 'User is X, computer is O',0DH,0AH,0DH,0AH,'$'
```

```

BOARD DB '123456789'
BTXT DB 0DH,0AH
      DB ' | | ',0DH,0AH
      DB '-----',0DH,0AH
      DB ' | | ',0DH,0AH
      DB '-----',0DH,0AH
      DB ' | | ',0DH,0AH,0DH,0AH,'$'
BPOS DB 2,6,10,24,28,32,46,50,54
PMSG DB 'Enter your move (0 to 9): $'
PIM1 DB 0DH,0AH,'That move does not make sense, try again.',0DH,0AH,'$'
PIM2 DB 0DH,0AH,'That square is occupied, try again.',0DH,0AH,'$'
CMMSG DB 'I choose square $'
CRLF DB 0DH,0AH,'$'
WINS DW 1,2,3,4,5,6,7,8,9 ;any row
      DW 1,4,7,2,5,8,3,6,9 ;any column
      DW 1,5,9,3,5,7 ;either diagonal
XWIN DB 'X wins the game!',0DH,0AH,'$'
OWIN DB 'O wins the game!',0DH,0AH,'$'
MTIE DB 'The game is a tie.',0DH,0AH,'$'

.CODE
.STARTUP
LEA DX,GMSG ;set up pointer to greeting
MOV AH,9 ;display string function
INT 21H
CALL SHOWBRD ;display board
NEXT: CALL PMOVE ;get player move
      CALL SHOWBRD ;display board
      CALL CHECK ;did player win or tie?
      JZ EXIT
      CALL CMOVE ;let computer move
      CALL SHOWBRD ;display board
      CALL CHECK ;did computer win or tie?
      JZ EXIT
      JMP NEXT ;continue with game
EXIT: .EXIT
SHOWBRD PROC NEAR
      MOV CX,9 ;set up loop counter
      SUB SI,SI ;set up index pointer
LBC: MOV AL,BPOS[SI] ;get a board position
      CBW ;convert to word
      MOV DI,AX ;set up pointer to board string
      MOV AL,BOARD[SI] ;get player symbol
      MOV BTXT[DI],AL ;write into board string
      INC SI ;advance index pointer
      LOOP LBC ;repeat for all nine positions
      LEA DX,BTXT ;set up pointer to board string
      MOV AH,9 ;display string function
      INT 21H ;DOS call
      RET
SHOWBRD ENDP

PMOVE PROC NEAR
      LEA DX,PMSG ;set up pointer to player string
      MOV AH,9 ;display string function
      INT 21H ;DOS call
      MOV AH,1 ;read keyboard function

```

```

INT 21H                ;DOS call
CMP AL,'1'             ;ensure user input is a digit
JC  BPM
CMP AL,'9'+1
JNC BPM
SUB AL,31H             ;remove ASCII bias
CBW                    ;convert to word
MOV SI,AX              ;set up index pointer
MOV AL,BOARD[SI]       ;get board symbol
CMP AL,'X'             ;is position occupied?
JZ  PSO
CMP AL,'O'
JZ  PSO
MOV BOARD[SI], 'X'     ;save player move
LEA DX,CRLF            ;set up pointer to newline string
MOV AH,9               ;display string function
INT 21H                ;DOS call
RET

BPM: LEA DX,PIM1        ;set up pointer to illegal string
STP: MOV AH,9           ;display string function
INT 21H                ;DOS call
JMP PMOVE              ;go give user a second chance
PSO: LEA DX,PIM2        ;set up pointer to occupied string
JMP STP                ;go process error message
RET

PMOVE ENDP

CMOVE PROC NEAR
SUB SI,SI              ;clear index pointer
NCM: MOV AL,BOARD[SI]   ;get board symbol
CMP AL,'X'             ;is position occupied?
JZ  STN
CMP AL,'O'
JZ  STN
MOV BOARD[SI], 'O'     ;save computer move (not very tough, is it?)
MOV AX,SI              ;save move value
PUSH AX
LEA DX,MSG             ;set up pointer to choice string
MOV AH,9               ;display string function
INT 21H                ;DOS call
POP DX                 ;get move value back
ADD DL,31H             ;add ASCII bias
MOV AH,2               ;display character function
INT 21H                ;DOS call
LEA DX,CRLF            ;set up pointer to newline string
MOV AH,9               ;display string function
INT 21H                ;DOS call
RET

STN: INC SI            ;advance to next position
JMP NCM                ;go check next position
CMOVE ENDP

CHECK PROC NEAR
SUB SI,SI              ;clear index pointer
MOV CX,8               ;set up loop counter
CAT: MOV DI,WINS[SI]   ;get first board position

```

```

MOV AH,BOARD[DI-1]      ;get board symbol
MOV DI,WINS[SI+2]       ;get second board position
MOV BL,BOARD[DI-1]      ;get board symbol
MOV DI,WINS[SI+4]       ;get third board position
MOV BH,BOARD[DI-1]      ;get board symbol
ADD SI,6                ;advance to next set of positions
CMP AH,BL               ;do all three symbols match?
JNZ NMA
CMP AH,BH
JNZ NMA
CMP AH,'X'              ;does match contain X?
JNZ WIO
LEA DX,XWIN             ;set up pointer to x-wins string
JMP EXC                ;go process string
WIO: LEA DX,OWIN         ;set up pointer to o-wins string
JMP EXC                ;go process string
NMA: LOOP CAT           ;no match, try another group
SUB SI,SI               ;clear index pointer
CFB: MOV AL,BOARD[SI]    ;get board symbol
CMP AL,'X'              ;is symbol X?
JZ IAH
CMP AL,'O'              ;is symbol O?
JZ IAH
RET                    ;no tie yet
IAH: INC SI             ;advance to next position
LOOP CFB               ;go check another board symbol
LEA DX,MTIE            ;set up pointer to tie message
EXC: MOV AH,9           ;display string function
INT 21H                ;DOS call
SUB AL,AL              ;set zero flag
RET
CHECK ENDP
END

```

The tic-tac-toe board is contained within the text string BTXT. The positions in the BTXT array that will contain board symbols are stored in the BPOS array. For example, the position of the first square in the first row is 2. The position of the first square in the second row is 24. The position of the last square in the third row is 54. These position values can be verified by carefully counting character places in BTXT. Together with the symbols stored in the BOARD array, the entire board can be analyzed. A sample tic-tac-toe board (generated during a game) might look like this:

```

X | 0 | 3
---
X | 5 | 6
---
7 | 8 | 9

```

where each unoccupied square contains a digit. The board is displayed by the SHOWBRD procedure.

The user enters a move through the PMOVE procedure. This procedure gets a user move from the keyboard and checks to ensure that the move is valid (a legal digit representing an unoccupied board position). Then the BOARD array is updated. The BOARD array for the sample tic-tac-toe board previously shown is:

XO3X56789

If the computer puts an O into position 3, BOARD will become:

XOOX56789

Once the BOARD array is updated, it is checked for a win. This is done by the CHECK procedure, which tests all rows, columns, and diagonals for three Xs or three Os. It indicates either a win (or a tie) by setting the zero flag upon return. CHECK uses the WINS array to examine eight groups of three board symbols, which represent all three rows and columns, and both diagonals.

---

**Programming Exercise 7.17:** Modify the TICTAC program so that the computer's first move is the center square. If the center square is already occupied, one of the four corner squares should be chosen.

---

**Programming Exercise 7.18:** Modify the TICTAC program so that the user or computer may go first.

---

**Programming Exercise 7.19:** Modify the TICTAC program so that the user may choose X or O.

---

**Programming Exercise 7.20:** Write a BLOCK procedure that examines the tic-tac-toe board and determines if any row, column, or diagonal contains exactly two Xs and a blank. Return the position of the blank (1-9) or 0 if no blocking move is found.

---

## 7.11 PROTECTED-MODE DETECTION

The 80386, 80486, and Pentium microprocessors are capable of operating in two basic modes of operation: *real mode* and *protected mode*. In real mode, they act like really fast 8086s. Registers are 16-bits wide. Memory space is limited to 1MB. No special instructions are allowed.

In protected mode, the full power of the processor is available. Registers are 32-bits wide, and the memory space can be as large as 4 gigabytes (4 billion bytes).

DOS is an example of a program that runs in real mode, because DOS was created for an 8086-based machine. Therefore, DOS has access to only 1MB of memory. An example of a program that runs in protected mode is Windows. As a matter of fact, most Windows application programs also run in protected mode. This can be verified easily. Find an .EXE program in the WINDOWS directory, such as CALC.EXE. If you execute CALC from DOS, you will get the following result:

This program requires Microsoft Windows.

CALC.EXE was able to determine that the computer did not have protected-mode capability. However, when Windows loads and executes, a DOS Protected-Mode Interface (DPMI) is established through Windows code. This DPMI is accessible through the use of *multiplex* interrupt 2FH. The multiplex interrupt is provided by DOS to support background programs (such as PRINT or DOSVER). The multiplex interrupt requires AX to be equal to 1687H to

check DPMI status. If a DPMI program is active, we get the following:

AX:0  
 BX: 32-bit support flag  
 CL: Processor type  
 DX: DPMI version number  
 SI: Private data paragraph count  
 DI: Entry point offset  
 ES: Entry point segment

The DPMISTAT program shown here uses this information to display the DPMI status.

;Program DPMISTAT.ASM: Check for DOS Protected-Mode Interface.

```
;
.MODEL SMALL
.DATA
NODPMI DB 'No DOS Protected-Mode Interface detected.',0DH,0AH,'$'
NOTSUP DB '32-bit programs are NOT supported.',0DH,0AH,'$'
ARESUP DB '32-bit programs ARE supported.',0DH,0AH,'$'
P286MSG DB 'Processor is an 80286.',0DH,0AH,'$'
P386MSG DB 'Processor is an 80386.',0DH,0AH,'$'
P486MSG DB 'Processor is an 80486 or Pentium.',0DH,0AH,'$'
PUNMSG DB 'Unidentified processor.',0DH,0AH,'$'

.CODE
.STARTUP
MOV AX,1687H ;get mode switch entry point
INT 2FH ;MULTIPLEX interrupt
OR AX,AX ;DPMI Present if AX = 0
JZ SHOWB ;go show 32-bit status
LEA DX,NODPMI ;set up pointer to nodpmi message
MOV AH,9 ;display string function
INT 21H ;DOS call
JMP EXIT

SHOWB: OR BX,BX ;are 32-bit programs supported?
JNZ YES32 ;jump if yes
LEA DX,NOTSUP ;set up pointer to not supported message
MOV AH,9 ;display string function
INT 21H ;DOS call
JMP SHOWP ;go show processor type

YES32: LEA DX,ARESUP ;set up pointer to supported message
MOV AH,9 ;display string function
INT 21H ;DOS call

SHOWP: CMP CL,2 ;is processor 80286?
JNZ P2
LEA DX,P286MSG ;set up pointer to 80286 message
JMP SCPU ;go display string

P2: CMP CL,3 ;is processor 80386?
JNZ P3
LEA DX,P386MSG ;set up pointer to 80386 message
JMP SCPU ;go display string

P3: CMP CL,4 ;is processor 80486?
JNZ P4
LEA DX,P486MSG ;set up pointer to 80486 message
JMP SCPU ;go display string

P4: LEA DX,PUNMSG ;set up pointer to unidentified message
```

```

SCPU:  MOV    AH,9           ;display string function
        INT    21H          ;DOS call
EXIT:  .EXIT
        END

```

When the program is executed from the DOS environment, the result is:

No DOS Protected-Mode Interface detected.

But, if we start up Windows (via WIN /3) and then choose MS-DOS Prompt from the Main window, we will get access to DOS from *inside* Windows.

When DPMISTAT is executed now, we get:

```

32-bit programs ARE supported.
Processor is an 80486 or Pentium.

```

So, checking for the presence of a DPMI program is important for programs that desire access to the 32-bit processing power of protected mode.

## 7.12 INTERFACING C WITH ASSEMBLY LANGUAGE

The C programming language has become very popular over the last few years, primarily because it is extremely portable. Programs of a **portable programming language** are easily transferred to other machines. For example, a C program written on a VAX<sup>TM</sup> mainframe will usually compile without errors when ported to an MS-DOS (80x86-based) or Macintosh<sup>®</sup> (680x0-based) machine. The purpose of the C compiler is to convert the statements in a C program into the correct sequence of machine language instructions for the host processor.

In this section, we will examine the machine code generated by a C compiler and the standard way the C compiler uses memory to access information.

One way to interface C with assembly language is to put the instructions directly into the C program. Examine the following C program:

```

main ()
{
    asm      mov     dl,0x30      /* begin counter at '0' */
    asm      mov     ah,2        /* display character function */
next:
    asm      int     0x21        /* DOS call */
    asm      inc     dl          /* increment counter */
    asm      cmp     dl,0x3a     /* is counter > '9' ? */
    asm      jnz     next
}

```

Notice that the program is written almost entirely in assembly language, with the exception of the `main()` keyword, and the braces `{ }` surrounding the statements. The C keyword `asm` is used to perform *in-line assembly* of the assembly language statements in the C program. The six assembly language statements form a loop that outputs the digits 0 through 9 to the display.

Normally, this C program would be compiled to create an executable program. This process, however, will not allow us to see the final assembly language generated by the C compiler. So, a different technique will be used to create an assembly language *source file* from the C program. A program available with many C compiler packages is used to convert the .C program into an .ASM program. This program is called TCC.EXE and is used like so:

```
TCC -S -EFILENAME.ASM FILENAME.C
```

This DOS command tells TCC to create FILENAME.ASM using FILENAME.C. The result of TCC's execution is as follows:

```

__TEXT    segment byte public 'CODE'
;
;    main()
;
;    assume    cs:__TEXT
__main    proc        near
    push        bp
    mov         bp,sp
;
;    {
;        asm     mov     dl,0x30      /* begin counter at '0' */
;
;        mov     dl,030H
;
;        asm     mov     ah,2        /* display character function */
;
;        mov     ah,2
@1@86:
;
;        next:
;        asm     int     0x21        /* DOS call */
;
;        int     021H
;
;        asm     inc     dl          /* increment counter */
;
;        inc     dl
;
;        asm     cmp     dl,0x3a     /* is counter > '9' ? */
;
;        cmp     dl,03aH
;
;        asm     jnz     next
;
;        jne     short@1@86
;
;    }
    pop         bp
    ret
__main    endp
__TEXT    ends
end

```

The assembly language generated by the `asm` keyword is almost identical to that used in the `asm` statement, except when numbers or labels are used in the operand field. For example, the `0x30` operand in the first statement is used to represent the value 30 hexadecimal. This is the standard way a hexadecimal number is represented in C. Recall that the assembly language required by ML requires `30H` to specify 30 hexadecimal. This is illustrated in the instruction `mov dl,30H`.

When a label is used, TCC must generate a unique name for it. The label `next` has the name `@1@86` associated with it. Notice that the `asm jnz next` statement created the instruction `jne short @1@86`. Both are functionally equivalent.



Finally, TCC supplies entry and exit code to allow execution of the program in the DOS environment. The instructions for program entry are:

```
push    bp
mov     bp, sp
```

and for program exit, we have:

```
pop     bp
ret
```

These two groups of instructions indicate that the BP and SP registers play an important role in C's interface with assembly language. This role is well defined and can best be explained through the examination of another C program and its associated assembly language code (generated by TCC).

The following C program contains a function called `addem` that uses two input parameters, `x` and `y`, both of which are defined as integers.

```
#include <stdio.h>

void addem(int x, int y);

main()
{
    int a, b;

    a = 5;
    b = 7;
    addem(a, b);
}

void addem(int x, int y)
{
    int c;

    c = x + y;
    printf("The sum is %3d\n", c);
}
```

This can be seen in the function's definition statement:

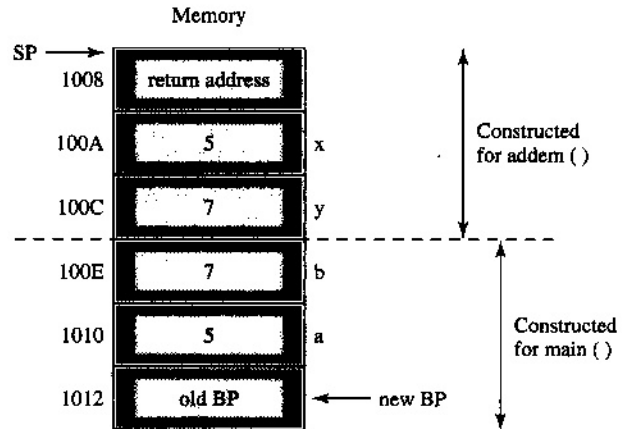
```
void addem(int x, int y);
```

The function definition statement provides important information to the C compiler, which must ensure that the parameter values are accessible when the function is called in the main program. For example, when the C compiler encounters the statement:

```
addem(a, b);
```

it creates a working environment for the function according to a standard known as the **C calling convention**. This calling convention dictates that parameter values used in a function call be pushed onto a run-time stack in right-to-left order. Thus, the value of variable `b` (parameter `y`) is pushed onto the run-time stack, followed by the value of variable `a` (parameter `x`). This is illustrated in Figure 7.4. Notice that the integer values of both variables (7 and 5) each occupy one word of stack space. This is due to the C compiler's use of a word to represent an integer.

After the function parameters are pushed, the program's return address is pushed. This is the address that will get control when the function completes execution. Thus, a portion of the run-time stack has been constructed to allow execution of the `addem( )` function.

**FIGURE 7.4** Run-time stack during a C function call

A second portion, which was constructed for the `main()` routine, has been previously loaded onto the stack. The first value pushed onto the stack in this portion was the original value of the BP register when `main()` got control. The BP register is then updated to point to the bottom of `main()`'s run-time stack. For purposes of this discussion, a set of addresses has been included in Figure 7.4 to help illustrate exactly where information is being saved. Thus, the bottom of the run-time stack is the word located at address 1012.

The new value of the BP register is used by `main()` to access all information on the run-time stack. When the compiler encounters:

```
int a, b;
```

it reserves two words of run-time stack space for the values of variables `a` and `b`. Figure 7.4 shows that these two words immediately follow the original BP register value. The compiler may now refer the variable `a` using the operand `[bp-2]`. Variable `b` is accessed through the operand `[bp-4]`. Note that the order of the variables stored in the run-time stack is identical to the order in which they were declared. This is exactly opposite the order used to push parameter values for a function.

To better understand the run-time stack usage, let us examine the assembly language file created by TCC for our current C program.

```
_TEXT    segment byte public 'CODE'
;
;    main()
;
;    assume    cs:_TEXT
_main    proc      near
;    push      bp
;    mov       bp,sp
;    sub       sp,4
;
;    {
;        int a, b;
;
;        a = 5;
;
;        mov    word ptr [bp-2],5
;
;        b = 7;
;
;        mov    word ptr [bp-4],7
```

```

;
;           addem(a, b);
;
;   push    word ptr [bp-4]
;   push    word ptr [bp-2]
;   call    near ptr _addem
;   pop     cx
;   pop     cx
;
;   )
;
;   mov     sp, bp
;   pop     bp
;   ret
_main      endp
;
;   void addem(int x, int y)
;
;   assume  cs:_TEXT
_addem     proc    near
;   push    bp
;   mov     bp, sp
;   sub     sp, 2
;
;   {
;
;       int c;
;
;       c = x + y;
;
;   mov     ax, word ptr [bp+4]
;   add     ax, word ptr [bp+6]
;   mov     word ptr [bp-2], ax
;
;       printf("the sum is %3d\n", c);
;
;   push    word ptr [bp-2]
;   mov     ax, offset DGROUP:s@
;   push    ax
;   call    near ptr _printf
;   pop     cx
;   pop     cx
;
;   )
;
;   mov     sp, bp
;   pop     bp
;   ret
_addem     endp
;
;   _TEXT
;   ends
;   _DATA segment word public 'DATA'
;   s@ label byte
;       db 'The sum is %3d'
;       db 10
;       db 0
;   _DATA ends
;   end

```

The first group of instructions in `_main` are:

```
push    bp
mov     bp, sp
sub     sp, 4
```

These instructions save the original value of the BP register, adjust the BP register so that it points to the bottom of the run-time stack, and adjust the SP register so that space is reserved for the integer variables `a` and `b` (as shown in Figure 7.4). The statement;

```
a = 5;
```

is coded in assembly language as:

```
mov     word ptr [bp-2], 5
```

This translates to the address 1010 in the run-time stack of Figure 7.4. A similar method is used to initialize the value of variable `b`, which is stored at address 100E in the run-time stack.

When the `addem(a, b);` statement is encountered, the compiler generates instructions to place `addem`'s parameter values onto the run-time stack. These instructions are as follows:

```
push    word ptr [bp-4]
push    word ptr [bp-2]
```

Remember that the `[bp-4]` value is the value of variable `b` and that `[bp-2]` refers to the value of variable `a`. Thus, the parameter values for the `addem()` function have been pushed onto the run-time stack in right-to-left order (`y` then `x`). The return address is automatically pushed by the:

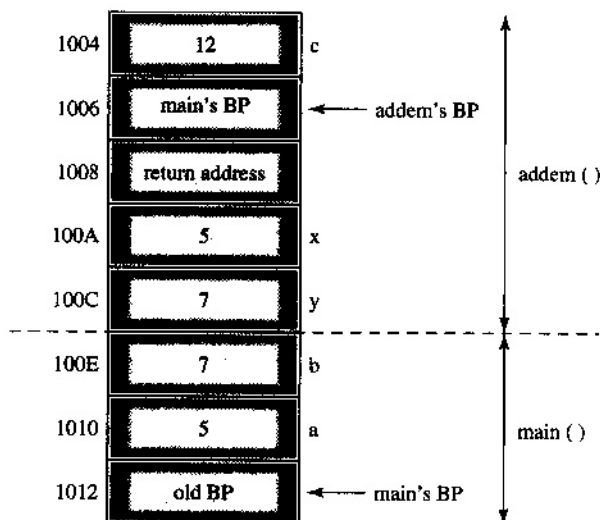
```
call    near ptr_addem
```

instruction.

When the `_addem` procedure gets control, it performs the same initial run-time stack allocation that `main()` did. The original value of the BP register is saved and then reloaded, and the SP register is updated to reserve room for the variable `c` used by `addem()`. Figure 7.5 shows the new run-time stack.

Now that the `_addem` procedure is executing, how does it access its `x` and `y` parameter values? By the nature of the C calling convention, the parameter's values are accessed by use of the BP register. However, because a return address was pushed onto the run-time

**FIGURE 7.5** Run-time stack while executing the `addem()` function



stack *after* the two parameter values, the offsets used with the BP register must be changed. For example, in the statement:

$$c = x + y$$

the value of variable *x* is accessed through the operand [bp+4] and variable *y* through [bp+6]. This is necessary to skip over the return address in the run-time stack. Use the address values in Figure 7.5 to prove the correctness of these operand addresses.

You may agree that the run-time stack gets very cluttered with information during a program's execution. Having to skip over return addresses and reserve room for variable storage makes it hard to keep track of where the actual return address for any given procedure is. The C calling convention offers an easy solution to this problem. Because the SP and BP registers are updated upon entry to any routine, upon exit we need only:

```
mov    sp, bp
pop    bp
ret
```

to always return correctly.

#### ■ EXAMPLE 7.11 The following C program calls an external integer function `xplusy()`:

```
#include <stdio.h>
extern int xplusy(int x, int y);
main()
{
    int a,b;
    a = 5;
    b = 7;
    printf("The sum is %d\n", xplusy(a,b));
}
```

The `xplusy()` function is an assembly language routine. A number of assembler directives may be used when writing `xplusy()` in order to implement the C calling convention and correctly manipulate the run-time stack. Here is what `xplusy()` looks like:

```
.MODEL SMALL,C
.CODE
PUBLIC xplusy
xplusy PROC    NEAR C, x:WORD, y:WORD
LOCAL    z:WORD
        MOV     AX,x        ;load value of x into AX
        ADD     AX,y        ;add value of y to AX
        MOV     z,AX        ;store AX in z
        RET
xplusy ENDP
END
```

The `PUBLIC` keyword must be used because `xplusy()` is an external reference to the `ADDER.C` program. The `PROC NEAR C` directive (along with `.MODEL SMALL,C`) informs the assembler that procedures must have the standard entry and exit code required by the run-time stack. This code is automatically generated when the assembler encounters

**FIGURE 7.6** xplusy() machine code

|          |  |      |             |                                                           |
|----------|--|------|-------------|-----------------------------------------------------------|
| 55       |  | PUSH | BP          | } Generated by PROC NEAR C                                |
| 8B EC    |  | MOV  | BP, SP      |                                                           |
| 83 C4 FE |  | ADD  | SP, -02     | Generated by LOCAL z:WORD                                 |
| 8B 46 04 |  | MOV  | AX, [BP+04] | x:WORD                                                    |
| 03 46 06 |  | ADD  | AX, [BP+06] | y:WORD                                                    |
| 89 46 FE |  | MOV  | [BP+02], AX | z:WORD                                                    |
| 8B E5    |  | MOV  | SP, BP      | } Automatically generated when RET is seen in source file |
| 5D       |  | POP  | BP          |                                                           |
| C3       |  | RET  |             |                                                           |

the first instruction of the procedure (MOV AX,x) and the last (RET). The PROC directive also allows us to specify the formal parameters used by the procedure (x:WORD and y:WORD) and their data type. A WORD corresponds to an integer in C.

The LOCAL directive is used to reserve room on the run-time stack for variables that are local to the procedure (z:WORD). All variables are referenced by the standard [BP +/- offset] notation.

Because xplusy() is an integer function, it must return an integer value. This value is returned in AX. Simple data types, such as chars and ints, are returned in the accumulator.

Figure 7.6 details the resulting machine code of the xplusy() function. Note the offsets associated with x [BP+4], y [BP+6], and z [BP-2]. These are the expected locations within the run-time stack for these variables.

When the ADDER program is compiled and linked with xplusy(), the resulting execution is as follows:

The sum is 12

This simple example should help you begin writing your own assembly language functions for C. ■

---

Understanding how the C calling convention operates allows you to write your own interface code for existing C routines. Through correct use of the run-time stack (via the BP register), you should be able to access and use compiler-generated C code for your own purposes.

---

**Programming Exercise 7.21:** Write an in-line assembly language C program that displays your initials on the screen.

---

**Programming Exercise 7.22:** Change the variable declarations for a, b, and c from int to float in the second C program. Make any other necessary changes to ensure correct compilation and then create an assembly language file using TCC. What is pushed onto the run-time stack now?

---

**Programming Exercise 7.23:** Explain what is done to the run-time stack in the second C program to support the printf() function.

---

**Programming Exercise 7.24:** Write an assembly language C function called max() that compares two integers x and y and returns the value of the larger integer.

## 7.13 ASSEMBLY LANGUAGE IN THE WINDOWS ENVIRONMENT

Why would a programmer choose to write code in assembly language, rather than a high-level language like C or Java? Does it provide easier control over hardware? Is it a requirement of the manufacturer? Is it easier to optimize handwritten assembly language than allowing a compiler to do it? Does the programmer simply enjoy using assembly language?

Windows programmers use assembly language to write many different things:

- Drivers for specific hardware.
- Dynamic Link Libraries (DLLs), libraries of ready-to-run code modules.
- WIN32 32-bit applications, including networking utilities and games.
- Computer worms and viruses.

All of these uses for assembly language require the skills of an experienced assembly language programmer, not someone who has just learned how to write an 80x86 assembly language program in the past month or two. Experienced programmers have written lots of their own assembly language programs (possibly even in other processor languages) and have spent a good deal of time examining other people's assembly language programs to learn how they work.

The Internet is full of assembly language tutorials for the 80x86, for both real-mode and protected-mode applications. There are also many free assemblers and linkers available for the 80x86 architecture. One popular package is called MASM32 (<http://www.masm32.com>), a complete package of tools used to create 32-bit code via assembly language. Those that own their own assemblers (MASM, TASM) have the ability to upgrade the software to assemble instructions for the most current processor.

Let us examine the structure of a simple assembly language program for Windows, and how to make it executable, using the MASM32 environment. This program, HELLO.ASM, is designed as a **Console application**. This means you must run the program from the command prompt in Windows. Console applications are text based, there are no graphic windows, or buttons to click.

```
;HELLO.ASM Simple Console application
.586
.model flat, stdcall
option casemap :none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\masm32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\masm32.lib

.data
    GreetMsg db "Greetings!",0

.code
start:
    invoke StdOut, addr GreetMsg
    invoke ExitProcess, 0
end start
```

The .586 directive could easily be .386, .486, or whatever your target architecture is. The flat model is the standard addressing scheme used in Windows applications. A flat

addressing space uses 32-bit addressing everywhere; there are no near and far types of addresses (you could say everything is far if you want). The `stdcall` parameter indicates how subroutine parameters are pushed onto the stack (right to left in this case, which is the standard convention). The `invoke` macro is used to process subroutine calls.

From the MASM32 editor, select **Console Assemble & Link** from the **Project** menu to create **HELLO.EXE**. When executed, **HELLO.EXE** simply outputs the string "Greetings!" to the command window.

To examine the actual code produced from the **HELLO.ASM** source file, it is necessary to use a disassembler or debugger. One such tool is **IDA**, the Interactive Disassembler, available from <http://www.datarescue.com>. Using **IDA**, the **HELLO.EXE** program can be loaded and disassembled. Here is a portion of the results:

```

        public start
start    proc near
        push    Offset aGreetings ; "Greetings!"
        call    sub_401018
        push    0                  ; uExitCode
        call    ExitProcess
        int     3                  ; Trap to Debugger
start    endp

```

These statements were generated from the following **HELLO.ASM** source statements:

```

start:
        invoke StdOut, addr GreetMsg
        invoke ExitProcess, 0
end start

```

But the `push` and `call` pairs of instructions are not all that is done by the `invoke` macro. The `sub_401018` subroutine, which actually performs the `StdOut` processing, is automatically generated, and its code looks like this:

```

sub_401018 proc near
nNumberOfBytesToWrite    = dword ptr -0Ch
NumberOfBytesWritten      = dword ptr -8
hFile                    = dword ptr -4
lpBuffer                 = dword ptr 8

        push    ebp
        mov     ebp, esp
        add     esp, 0FFFFFFF4h
        push    0FFFFFFF5h          ; nStdHandle
        call    GetStdHandle
        mov     [ebp+hFile], eax
        push    [ebp+lpBuffer]
        call    sub_401050
        mov     [ebp+nNumberOfBytesToWrite], eax
        push    0                  ; lpOverlapped
        lea     eax, [ebp+NumberOfBytesWritten]
        push    eax                ; lpNumberOfBytesWritten
        push    [ebp+nNumberOfBytesToWrite] ; nNumberOfBytesToWrite
        push    [ebp+lpBuffer]        ; lpBuffer
        push    [ebp+hFile]           ; hFile
        call    WriteFile
        mov     eax, [ebp+NumberOfBytesWritten]
        leave
        retn    4
sub_401018 endp

```



Phew! That is a lot of instructions and a lot to remember if a macro like `invoke` is not available in your chosen assembler.

Instead of the text-only interface available with Console applications, one might prefer the windowed approach to program development. MASM32 is able to create code for **windowed applications** as well. With only a few changes to `HELLO.ASM`, we can add a simple message box window to our application. Examine `WINHELLO.ASM` to see how this is accomplished:

```
.386
.model flat, stdcall
option casemap :none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

.data
    TitleMsg db "MASM32 Example #1",0
    GreetMsg db "Greetings!",0

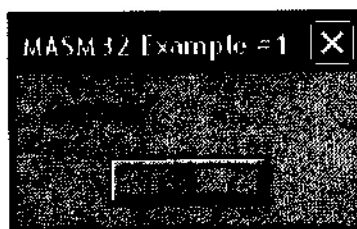
.code
start:
    invoke MessageBox, NULL, addr GreetMsg, addr TitleMsg, MB_OK
    invoke ExitProcess,0
end start
```

To create `WINHELLO.EXE` using MASM32, select **Assemble ASM File** from the **Project** menu. When `WINHELLO.EXE` is executed, the message box window pops up, as indicated in Figure 7.7. Clicking **OK** ends the program.

Table 7.3 shows some characteristics of the `HELLO.EXE` and `WINHELLO.EXE` programs. What do you imagine accounts for the additional object file size in `WINHELLO.EXE`?

If you choose to develop programs for Windows in assembly language, and you have the necessary skills, it does not take long to turn an idea into a working application.

**FIGURE 7.7** Window created by `WINHELLO.EXE`



**TABLE 7.3** Comparing Console and Windowed applications

| Program      | Type     | Object File Size | EXE File Size |
|--------------|----------|------------------|---------------|
| HELLO.EXE    | Console  | 565 bytes        | 2,560 bytes   |
| WINHELLO.EXE | Windowed | 654 bytes        | 2,560 bytes   |

---

**Programming Exercise 7.25:** What happens if you leave the comma and the 0 off the greeting message, as in

```
GreetMsg db "Greetings!"
```

Try this with the HELLO.ASM program. What are the results?

---

**Programming Exercise 7.26:** Repeat Programming Exercise 7.25 for the windowed application WINHELLO.ASM.

---

## 7.14 ASSEMBLY LANGUAGE IN THE LINUX ENVIRONMENT

The reasons users may have for developing Windows applications using assembly language may apply to the Linux environment as well. In this section, we will examine the structure of an assembly language program for Linux, and how to make it executable. But first, let us get a better picture of the execution environment in Linux.

Examine the following C program (adder.c) that uses a function to return the sum of two integers:

```
#include <stdio.h>

int addem(int x, int y);

int main()
{
    int a, b, c;

    a = 5;
    b = 7;
    c = addem(a,b);
    printf("The sum is %d\n",c);
}

int addem(int x, int y)
{
    return(x + y);
}
```

The values for main's local variables a and b are passed to the addem() function. Addem() returns the sum of the values, which is stored in main's local c variable before being printed out by the printf() statement.

In Linux, the command

```
$ gcc -S adder.c
```

will use the gcc compiler to create the assembly language source for adder.c, called adder.s, which looks like this:

```
.file "adder.c"
.section .rodata
.LC0:
.string "The sum is %d\n"
.text
```

```

.globl main
.type    main, @function
main:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $24, %esp
    andl     $-16, %esp
    movl     $0, %eax
    subl     %eax, %esp
    movl     $5, -4(%ebp)
    movl     $7, -8(%ebp)
    subl     $8, %esp
    pushl    -8(%ebp)
    pushl    -4(%ebp)
    call     addem
    addl     $16, %esp
    movl     %eax, -12(%ebp)
    subl     $8, %esp
    pushl    -12(%ebp)
    pushl    $.LC0
    call     printf
    addl     $16, %esp
    leave
    ret
.size     main, .-main
.globl addem
.type     addem, @function
addem:
    pushl    %ebp
    movl     %esp, %ebp
    movl     12(%ebp), %eax
    addl     8(%ebp), %eax
    leave
    ret
.size     addem, .-addem
.section   .note.GNU-stack,"", @progbits
.ident    "GCC: (GNU) 3.3.2 20031022 (Red Hat Linux 3.3.2-1)"

```

Note the differences with MASMs (and Intel's) operand format, such as the use of `%eax` instead of just `eax` in the operand fields. This format is accepted AT&T assembly language syntax. You will also notice that the order of the operands has changed, being `<source>`, `<destination>`, and operand-size suffixes are attached to some instructions. So, the Intel instruction

```
MOV EAX, 0
```

becomes

```
movl 0, %eax
```

in the AT&T-syntax format required for Linux assembly language programs. Not all Linux assemblers work this way, so it is best to review the assembler documentation before experimenting.

Linux makes use of interrupts to handle **syscalls**, calls made to the operating system. Table 7.4 lists some of the more common syscalls. There are several hundred syscalls defined.

**TABLE 7.4** The first 16 syscalls available in Linux

| <i>Number</i> | <i>Syscall</i> |
|---------------|----------------|
| 1             | sys_exit       |
| 2             | sys_fork       |
| 3             | sys_read       |
| 4             | sys_write      |
| 5             | sys_open       |
| 6             | sys_close      |
| 7             | sys_waitpid    |
| 8             | sys_creat      |
| 9             | sys_link       |
| 10            | sys_unlink     |
| 11            | sys_execve     |
| 12            | sys_chdir      |
| 13            | sys_time       |
| 14            | sys_mknod      |
| 15            | sys_chmod      |
| 16            | sys_lchown     |

Syscalls operate in a manner similar to the DOS INT 21H function call. The syscall number is placed into EAX, and other registers are initialized (if any) prior to executing INT 80H (0x80 in C notation). Up to six registers may be used to pass parameters to a syscall, and they are always used in the following order: EBX, ECX, EDX, ESI, EDI, and EBP.

Here is a short assembly language source file called `syshello.s` that is used to display a text string on the standard output:

```
.data
hello:
    .string "Greetings\n"
.text

.globl _start
_start:

    # setup parameters for SYS_WRITE
    movl $4,%eax
    movl $1,%ebx
    movl $hello,%ecx
    movl $10,%edx
    int $0x80

    # setup parameters for SYS_EXIT
    movl $1,%eax
    movl $0,%ebx
    int $0x80
```

The source file is assembled using the following command:

```
$ as -a -o syshello.o syshello.s
```

The “as” utility is part of the gcc compiler package. The command line instructs “as” to create an object file called syshello.o. The “as” utility outputs the following statements:

```
GAS LISTING syshello.s                                page 1
 1          .data
 2          hello:
 3 0000 47726565          .string "Greetings\n"
 3          74696E67
 3          730A00
 4          .text
 5
 6          .globl _start
 7          _start:
 8
 9          # setup parameters for SYS_WRITE
10 0000 B8040000          movl    $4,%eax
10          00
11 0005 BB010000          movl    $1,%ebx
11          00
12 000a B9000000          movl    $hello,%ecx
12          00
13 000f BA0A0000          movl    $10,%edx
13          00
14 0014 CD80          int     $0x80
15
16          # setup parameters for SYS_EXIT
17 0016 B8010000          movl    $1,%eax
17          00
18 001b BB000000          movl    $0,%ebx
18          00
19 0020 CD80          int     $0x80

GAS LISTING syshello.s                                page 2
DEFINED SYMBOLS
          syshello.s:2          .data:00000000 hello
          syshello.s:7          .text:00000000 _start

NO UNDEFINED SYMBOLS
```

The object file is not executable until it is linked. This is accomplished with this command line:

```
$ ld -s -o syshello syshello.o
```

Here the executable file syshello is created from the syshello.o object file. It is executed with this command:

```
$ ./syshello
```

Table 7.5 compares the object code and executable code size of syshello with the two Windows applications from the previous section.

**TABLE 7.5** Comparing Linux and Windows applications

| Program      | Type             | Object File Size | Executable File Size |
|--------------|------------------|------------------|----------------------|
| syshello     | Linux Console    | 588 bytes        | 392 bytes            |
| HELLO.EXE    | Windows Console  | 565 bytes        | 2,560 bytes          |
| WINHELLO.EXE | Windows Windowed | 654 bytes        | 2,560 bytes          |

The Linux Console application surely has a more efficient way of displaying strings than the Windows Console application, even though their object files are roughly equivalent in size.

As with Windows, there are plenty of assembly language examples for Linux available on the Internet. Spend some time examining the examples before attempting your own project; it will be a good investment.

---

**Programming Exercise 7.27:** Draw a figure showing the run-time stack associated with the `syshello` application.

---

## 7.15 TROUBLESHOOTING TECHNIQUES

Many different types of programming applications were covered in this chapter. Let us take a look at some of the techniques presented, as they may be useful when beginning a new application.

- Create source files that contain code for all of the things you normally do in a program, such as display strings, input and output numbers, or scan the command tail. Assemble the source files and combine them into a library of code modules that you can link to. This will allow you to reuse your code when writing new applications.
- The execution time of a block of code depends on many factors, such as clock speed, processor type (8086 vs. Pentium), and the number of branch instructions. The calculated execution time should be thought of as an estimate and used as a guide to control further code development.
- Use memory-resident programs with caution. It is easy to crash the system with an improperly written TSR. If you intercept an interrupt to gain access to the TSR program, remember to restore the original interrupt vector before exiting.
- Add memory management and mouse support to your list of PC software/hardware that you can control. You may wish to create a subdirectory that contains working examples of all the different control applications. The example programs can serve as templates for further development.
- Do not attempt any protected-mode programming without further research into protected-mode operation. Chapter 14 provides a detailed look at protected mode.
- Review the way the stack is used when interfacing a C program to an assembly language routine. Stack-based parameter passing is very useful and should be well understood.

Your own special programming techniques will develop over time. You may see an example program in a magazine that does something neat and adopt it (start using it in your code). Programs written by others are a good source of information for you. Search the Web for assembly language examples; you will find plenty of archives. Go to a local computer show and buy a used assembly language book. It pays to have lots of references.

---

## SUMMARY

In this chapter we examined many advanced topics that have direct application to real-world needs. Instruction execution time, interrupts, multitasking, and memory management are all important areas for the serious programmer. Interfacing with the mouse and

working with memory-resident code add up to more powerful programming applications. Specific programming techniques, such as assembling and linking separate source modules, and macro usage were also covered. Some technical details of protected mode and C assembly language structure were also explained. Assembly language in the Windows and Linux environments was also explored.

---

## STUDY QUESTIONS

1. Explain the meaning and use of the PUBLIC and EXTRN assembler directives.
2. Show the linker command needed to create the executable XY.EXE from the object files XY.OBJ and FUNCT.OBJ.
3. Explain what is meant by a nested macro.
4. Explain why a label must be declared LOCAL if a macro is used more than once.
5. Compute the execution time of this section of code; assume a 100-MHz clock frequency.

```
        MOV     CX,1000H
NEXT:   ADD     AL,2
        MOV     [SI],AL
        LOOP   NEXT
```

6. Repeat Question 5, assuming that all memory references take an additional two cycles.
7. What are the addresses in the interrupt vector table for INT 21H? How can DEBUG be used to display the interrupt vector?
8. How can DOS INT 21H, Function 35H be used to get the interrupt vector for INT 10H?
9. Use DOS INT 21H, Function 25H to store the interrupt vector 0AE3:09BF.
10. Why is it necessary for the SWITCHER program to restore the original INT 1CH interrupt vector before exiting?
11. Explain how memory is managed by DOS.
12. What instructions are needed to allocate 8,192 bytes of memory?
13. How is mouse data accessed? When is mouse information available?
14. How does MOUSETEST check for the presence of a mouse?
15. What must be done to make a program memory resident?
16. Why do you have to save the original interrupt vector when hooking an interrupt?
17. Explain how the tic-tac-toe board is managed in TICTAC.
18. What are the differences between real mode and protected mode?
19. How is the stack used in a C program?
20. What is *in-line* code in a C program?

---

## ADDITIONAL PROGRAMMING EXERCISES

1. Write a series of string procedures, such as STRLEN (string length), STRCAT (string catenation), STRCPY (string copy), and STRCHR (string has character). Create an object code library for the string procedures.
2. Write a macro that will set or clear a specific bit in register AX. For example, BITOP S,4 will set bit 4 of AX. BITOP C,11 will clear bit 11.

3. Write a macro called MAX that returns the maximum value contained in parameters A and B. For example, MAX 50,36 returns the value 50. MAX DX,2000 returns either the value of DX or 2000, whichever is higher. The maximum value must always be returned in register AX.
4. Write an interrupt service routine that handles these four functions:

```

AH = 05H: NOT  AL
AH = 10H: AND  AL,BL
AH = 20H: OR   AL,BL
AH = 80H: XOR  AL,BL

```

Install the routine using DOS INT 21H, Functions 35H and 25H.

5. Write a program that will hook INT 16H and keep track of the number of keys pressed on the keyboard. Save the key count as a word value and show how DEBUG can display the count.
6. Modify the MEMMAN program to reserve memory blocks that double in size with each request. The first block size should be 64 bytes. How many blocks are assigned? What are their segment addresses?
7. Write a program that will request three 1KB blocks of memory. Add the bytes in the first block to the bytes in the second block, and store the sum in the third block. Use DEBUG to demonstrate that the third block contains the correct results.
8. Write a mouse application that places a red \* on the screen at the current mouse coordinates whenever the left button is pushed.
9. Write a memory-resident program that beeps the speaker three times at the beginning of each hour, and once on the half-hour.
10. Write a blackjack program that allows the user to play one hand of blackjack against the computer. The computer should deal two random cards to the user and itself, evaluate the points in both hands, and ask the user for hits, if necessary.
11. Generate assembly language for the following C program:

```

#include <stdio.h>
int adder (int x, int y);
main()
{
    int a = 10, b = 20;
    printf("The sum is %d\n", adder (a,b));
}
int adder (int x, int y)
{
    return (x + y);
}

```

12. Write a program that disassembles the machine code beginning at a specific address, similar to the unassemble command in DEBUG.
13. Download MASM32 and install it. Then locate an example program on the Internet and get it to work.
14. Locate a Linux assembly language program on the Internet and get it to work.

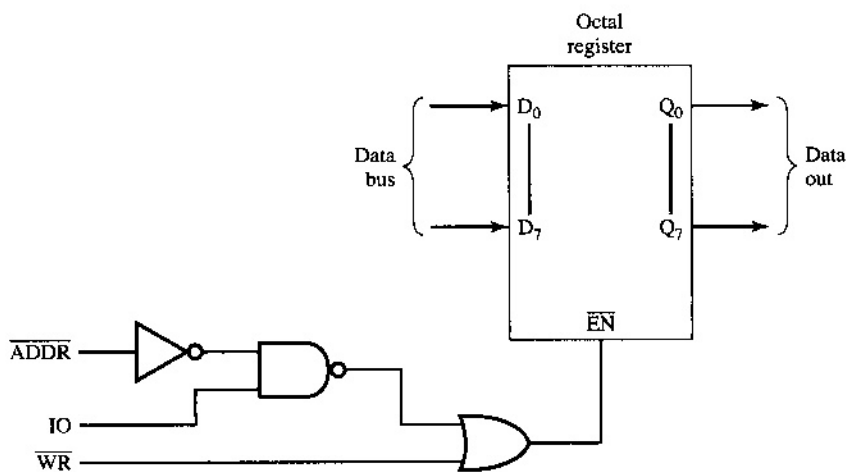


# PART 4

## Hardware Architecture

---

- 8 Hardware Details of the 8088
- 9 Memory System Design
- 10 I/O System Design
- 11 Interfacing with the 80x86
- 12 Building a Working 8088 System



Basic logic gates play an important role in hardware interfacing

---

## CHAPTER 8

---

# Hardware Details of the 8088

---

### OBJECTIVES

In this chapter, you will learn about:

- The general specifications of the 8088 microprocessor
- The processor's control signal names and functions
- General signal relationships and timing
- Methods by which the 8088 can interface with external devices
- The external interrupt signals and their operations
- The 8088 bus controller
- The method used to access an 8085 peripheral

### KEY TERMS

|                           |                       |                           |
|---------------------------|-----------------------|---------------------------|
| Interrupt enable flag     | Maxmode               | Single-stepping           |
| Interrupt service routine | Megabyte              | Suspended execution state |
| Interrupt type            | Minmode               | T state                   |
| LOCK prefix               | Nonmaskable interrupt | Transceiver               |
| Masking                   | Power-on reset        | Wait state                |

---

## 8.1 INTRODUCTION

Before using any microprocessor, it is necessary to understand both its hardware requirements and its software functions. In this chapter we will examine all 40 pins of the 8088's package and see what their uses are in a larger system employing the 8088 as its CPU. We will not concentrate on interfacing, because this important topic is covered in Chapters 9, 10, and 11. Upon completion of this chapter, we should, however, know about the various signals of the processor to begin interfacing it with support circuitry, which includes memories, I/O devices, and coprocessors. The hardware architecture of the Pentium, which is covered in Chapter 13, is radically different from that of the 8088. Even so, the experience gained working with the 8088's hardware is applicable to current microprocessor

technology, especially because the motherboards of today's personal computers still support the original bus architecture and signals. The information presented in this chapter, and in Chapters 9 through 11, will prepare you for the design of a working 8088 system in Chapter 12.

Section 8.2 gives a quick overview of the capabilities of the 8088, its memory addressing capabilities, available clock speeds, and various other functions. Section 8.3 covers all 40 pins of the 8088 in detail. The pins are separated into eight functional groups, such as interrupt control, system control, and processor status. Block diagrams and timing waveforms are given where applicable, except where they might apply to interfacing. Section 8.4 describes the operation of the 8284 clock generator, an essential integrated circuit used in the 8088-based systems. A second essential component is the 8288 bus controller, which is covered in Section 8.5. Timing diagrams for certain processor operations are examined in Section 8.6, and 8.7 covers the various types of bus connectors found on the motherboard in many personal computers. Hardware troubleshooting techniques are presented in Section 8.8.

## 8.2 CPU SPECIFICATIONS

Although we covered some of the 8088's specifications in Chapter 2, it will be useful to cover them again, this time adding more detail.

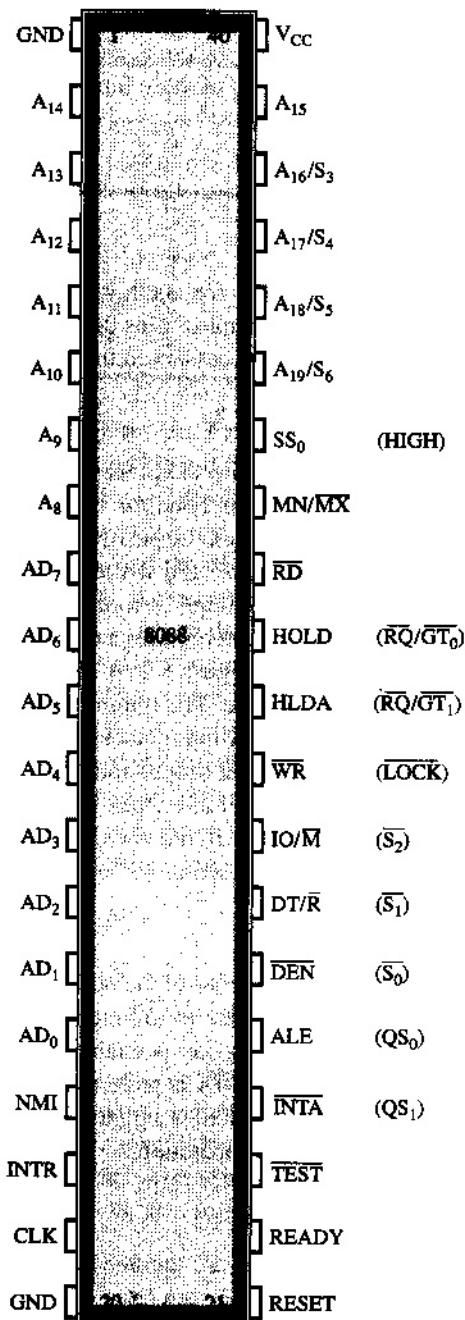
The 8088 is a 16-bit microprocessor that communicates with the outside world via an 8-bit bidirectional data bus. This requires the 8088 to perform *two* read cycles to capture 16-bit chunks of data. This has the effect of doubling memory access time and increasing program execution time. The 8086, having a 16-bit external data bus, needs to perform only *one* read cycle to fetch the same data and thus executes programs faster than the 8088.

The 8088's 20-bit address bus can access over *1 million* bytes of memory ( $2^{20} = 1,048,576$  bytes, to be exact). We commonly refer to this number as 1 **megabyte** of memory. Control signals are provided that enable external circuitry to take over the 8088's buses (a must for DMA [direct memory access] operations), and two interrupt lines are included to provide maskable and nonmaskable interrupt capability. A number of status outputs are available, which may be used to decode any of eight internal CPU states, and other control signals are provided to allow interfacing with 8085 and 8088 peripherals.

The 8088 comes with maximum clock speeds of 5 or 8 MHz, as of this writing, and has been on the market long enough to be purchased at a reasonable cost. But the power of the 8088 can be tapped only if we know how to use it. So let us begin examining the functional operation of the processor.

### Minmode Operation

The 8088 has two functional modes of operation: minimum mode (**minmode**) and maximum mode (**maxmode**). Certain pins on the processor have been designed for dual purposes, one for minmode and the other for maxmode. Figure 8.1 shows a pinout diagram of the 8088. Notice how pins 24 through 31 and pin 34 have two sets of signal names. In minmode, these nine signals are  $\overline{INTA}$ , ALE,  $\overline{DEN}$ ,  $\overline{DT/R}$ ,  $\overline{IO/\overline{M}}$ ,  $\overline{WR}$ , HLDA, HOLD, and  $\overline{SS_0}$ . They correspond to control signals needed to operate memory and I/O devices connected to the 8088 and are compatible with signals used in older 8085-based systems. Because the 8088 generates these signals in minmode, fewer chips are needed in the overall system; however, some functions are unavailable when the 8088 operates in minmode, including bus request/grant operations and coprocessor capability.

**FIGURE 8.1** 8088 pin assignments

Note: ( ) denotes a maxmode signal

### Maxmode Operation

When the 8088 operates in maxmode, the nine signals we just examined change their functions. The new signals become QS<sub>1</sub>, QS<sub>0</sub>,  $\overline{S_0}$ ,  $\overline{S_1}$ ,  $\overline{S_2}$ ,  $\overline{LOCK}$ ,  $\overline{RQ}/\overline{GT_1}$ ,  $\overline{RQ}/\overline{GT_0}$ , and HIGH. The lack of control signals (which exist only in minmode operation) now requires the use of the 8288 bus controller to generate memory and I/O read/write signals. Because an external

chip is now generating these control signals, the processor is free to expand its functional capability. This allows the use of an 8087 coprocessor and provides bus request/grant operation and queue status. All of these functions will be explained in detail in the next section.

### 8.3 CPU PIN DESCRIPTIONS

Refer to Figure 8.1 for another look at the 40 pins of the 8088's Dual In-line Package. There are eight groups of pins that we will examine in this section. Each group performs a specific function, necessary to the proper operation of the 8088.

#### $V_{CC}$ , GND, and CLK

This group deals with the processor power and clock inputs. Note that there are two pins for ground (GND) and one for  $V_{CC}$ . Both grounds must be used for proper operation. The 8088 operates on a single, positive supply of  $5\text{ V} \pm 10$  percent (with some versions having a 5 percent tolerance) and will dissipate 2.5 watts of power at this voltage. The specified supply current is 340 mA at room temperature.

The CLK input requires a digital waveform with a 33 percent duty cycle. This waveform is shown in Figure 8.2. A 33 percent duty cycle means that the digital level is high one-third of the time. The minimum clock period is 200 ns, corresponding to a frequency of 5 MHz. The maximum clock period is 500 ns, resulting in a minimum clock speed of 2 MHz. Rise and fall times should be kept under 10 ns. The TTL-compatible clock signal is generated by the 8284 clock generator covered in Section 8.4. Even though the clock input is internally buffered, the clock signal should be kept at a *constant* frequency (via the 8284 and a crystal oscillator) for best operation.

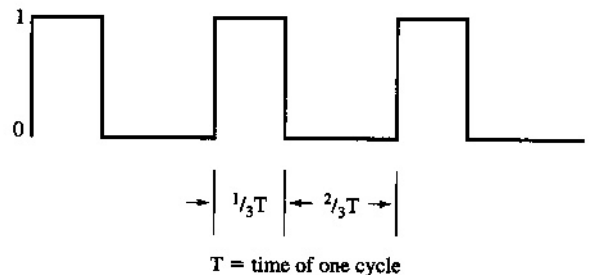
#### $\overline{MN}/\overline{MX}$

This pin is used to control the 8088's mode of operation. Remember from Section 8.2 that the 8088 may operate in minmode or maxmode. The specific mode is selected with the  $\overline{MN}/\overline{MX}$  pin. Maxmode is enabled when  $\overline{MN}/\overline{MX}$  is connected to ground. Minmode is enabled when  $\overline{MN}/\overline{MX}$  is pulled high (through an appropriate pullup resistor). As we saw before, nine of the 8088's signals have two functions, with each function depending on the mode of processor operation.

#### $\overline{S}_0$ , $\overline{S}_1$ , and $\overline{S}_2$

These three signals are the 8088's status outputs and are active only when the processor is in maxmode. They are used to indicate internal processor operations. The 8288 bus

**FIGURE 8.2** Three cycles of the CLK waveform



**TABLE 8.1** 8088 status signals

| $\overline{S}_2$ | $\overline{S}_1$ | $\overline{S}_0$ | Indicated Operation   |
|------------------|------------------|------------------|-----------------------|
| 0                | 0                | 0                | Interrupt acknowledge |
| 0                | 0                | 1                | I/O read              |
| 0                | 1                | 0                | I/O write             |
| 0                | 1                | 1                | Halt                  |
| 1                | 0                | 0                | Code access           |
| 1                | 0                | 1                | Memory read           |
| 1                | 1                | 0                | Memory write          |
| 1                | 1                | 1                | Passive               |

controller (covered in Section 8.5) uses the status outputs to generate memory and I/O read/write signals. Table 8.1 shows the eight different conditions that can be indicated by the status outputs. The *interrupt acknowledge* status can be used by external circuitry to manipulate the processor's interrupt mechanism. The *code access* status indicates when the processor is fetching instructions. When the 8088 has completed a bus cycle, the status outputs will indicate the passive state. An easy way to decode all eight processor states is to use a three- to eight-line decoder, such as the 74LS138; this is left for you to do as a homework problem. Normally in a small system we have no use for most of the decoded cycle states. In fact, an interrupt acknowledge signal ( $\overline{INTA}$ ) is already provided for us when operating in minimum mode.

The status outputs are capable of tri-stating when the 8088 enters into hold acknowledge (see the description of HOLD and HLDA in the next section).

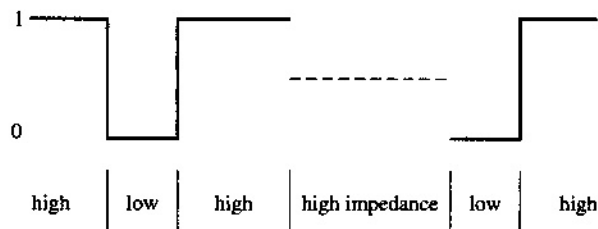
A tri-state signal may be in one of three states at any time: low (0) state, high (1) state, and high-impedance (open) state, as indicated in Figure 8.3.

## RESET, READY, HOLD, and HLDA

This group of signals is used for system control. RESET and READY operate the same way in both minmode and maxmode. HOLD and HLDA (*hold acknowledge*) work only when the processor is in minmode.

RESET is the signal that really gets the processor running after a power-up. It is very important to RESET the 8088 when power is first applied to guarantee that the 8088 begins doing intelligent things. A high logic level is needed to activate the RESET input, which must remain high for at least four clock cycles to ensure proper operation. RESET can be applied during program execution as well, as a sort of panic button used to start the program over from the beginning.

Asserting a RESET causes the 8088 to fetch the first instruction from memory, beginning at address FFFF0H. This requires some type of ROM data at that address to guarantee that the 8088 begins running correctly.

**FIGURE 8.3** Tri-state signal levels

READY is an input that informs the processor that the selected memory or I/O device is ready to complete a data transfer. READY is often used to synchronize the fast processor with a slow memory or I/O device that may need an extended bus cycle to perform a read or write operation. A high logic level on READY indicates that the 8088 may go ahead and complete the bus cycle. A low logic level causes the processor to extend the bus cycle with all signals frozen at their current logic levels. This means that the processor keeps the address lines, data bus, and control signals in their current states so that the external device may use them for a longer period of time.

HOLD is a minmode input that is used to place the processor into a **suspended execution state**. While the 8088 is in a hold state, it does not continue program execution. In fact, many of the processor's outputs are automatically tri-stated to prevent conflict on the system bus. The most common use of HOLD is in a computer system having two or more processors that need to share a bus. If the processors share memory (such as EPROM or RAM space), only one processor may access memory at a time. When a processor wishes to take control of the buses and access memory, it must first issue a HOLD request to the other processors in the system, which will suspend their processing and release the system bus. A high logic level is needed to activate HOLD. Furthermore, the 8088 will remain held only as long as the HOLD input remains high. The processor will resume program execution where it left off as soon as HOLD goes low.

HLDA (hold acknowledge) is a minmode output used to indicate to external devices that the 8088 has suspended execution (via HOLD). HLDA will go to a high logic level to show that the processor has actually stopped execution. Technically, another device should take over the system bus only *after* HLDA goes high. When execution resumes (i.e., HOLD has been taken low), HLDA will go low.

## NMI, INTR, and $\overline{\text{INTA}}$

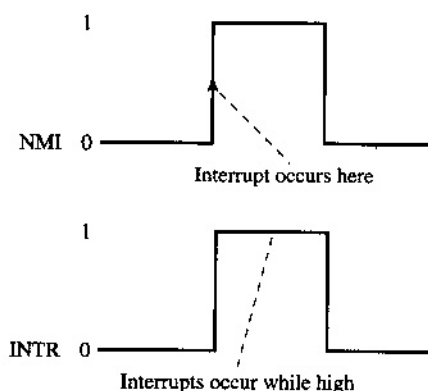
These three signals control the activity of external hardware interrupts. NMI and INTR are inputs and function identically in either processor mode.  $\overline{\text{INTA}}$ , an output, is available only in minmode.

External hardware interrupts are used to suspend current program execution and vector the processor to a special set of instructions called an **interrupt service routine (ISR)**. Interrupts are used to perform high-priority tasks without affecting the main processor program (except for a loss of execution time). A very useful application of hardware interrupts is keeping track of time. It is not difficult to convert the 60-cycle powerline frequency into a digital signal (with the use of a Schmitt trigger). The resulting 60-Hz digital signal is then connected to NMI or INTR, generating 60 interrupts every second. The corresponding ISR that services the interrupt decrements a counter each time it runs. If the counter is initially set to 60, it is easy for the computer to know when 1 second has passed.

**NMI (nonmaskable interrupt)** requires a *rising edge* to be recognized, as indicated in Figure 8.4. It cannot be internally disabled (masked) by software, hence its name. NMI generates a type-2 interrupt. We have seen (in Chapter 5) that there are 255 different types of interrupts. NMI is recognized by the 8088 at the end of the currently executing instruction. The address of the type-2 ISR is then read from a table containing all ISR addresses. This table is stored in memory and is called the interrupt vector table. The 8088 then jumps to the ISR address to process the interrupt code.

**INTR (interrupt request)** requires a *high* logic level to be recognized. Leaving INTR in a high state could cause repeated interrupts, so it is necessary to design circuitry to prevent this from happening. A bit in the processor's status register that is used to enable/

FIGURE 8.4 NMI and INTR signals



disable INTR is called the **interrupt enable flag (IF)**. Using the IF to disable INTR can be done easily with software, and is referred to as **masking**. INTR operates in much the same way as NMI except for the way the interrupt type is generated. NMI automatically causes a type-2 interrupt. The interrupt type for INTR is actually read from the processor's data bus during an interrupt acknowledged cycle. The 8-bit interrupt vector read from the data bus is internally converted into the proper address for the interrupt vector table.

Each interrupt has its own priority, with NMI having the higher priority of the two. The priority is used to determine which interrupt is recognized first if both occur at the same time. The higher priority of NMI guarantees that it is recognized before INTR. A comparison of NMI and INTR is shown in Table 8.2.

$\overline{INTA}$  (interrupt acknowledge) is an active low output that operates in minmode and is used to indicate that the 8088 has received an INTR and is beginning interrupt processing. The external circuitry connected to INTR should use  $\overline{INTA}$  to control when the 8-bit interrupt vector is placed onto the data bus. The exact timing of  $\overline{INTA}$  will be covered in Section 8.6.

### $\overline{RQ}/\overline{GT}_0$ , $\overline{RQ}/\overline{GT}_1$ , and $\overline{LOCK}$

These three maximum mode signals are used to interface the 8088 with other devices capable of taking over the system bus. In the description of the HOLD input, we saw one way two 8088s could share a common system bus and memory. The three signals presented here offer a second technique.  $\overline{RQ}/\overline{GT}_0$  and  $\overline{RQ}/\overline{GT}_1$  are request/grant signals used by other devices called bus masters to take over the 8088's system bus. An example is the 8087 coprocessor, which periodically takes over the system bus to read data or write results into memory. Both  $\overline{RQ}/\overline{GT}$  signals are bidirectional, meaning that they act as inputs and outputs.  $\overline{RQ}/\overline{GT}_0$  has priority over  $\overline{RQ}/\overline{GT}_1$ , and both have internal pullup resistors. The following discussion will concern  $\overline{RQ}/\overline{GT}_0$  only, but applies to  $\overline{RQ}/\overline{GT}_1$  as well.

When another bus master decides to take over the system bus, it will pull  $\overline{RQ}/\overline{GT}_0$  low for one clock period. When the 8088 is ready to release the system bus, it will use

TABLE 8.2 Comparison of NMI and INTR

| Interrupt | Logic Level Needed to Trigger | Disabled via Software | Priority |
|-----------|-------------------------------|-----------------------|----------|
| NMI       | Rising edge                   | No                    | High     |
| INTR      | High                          | Yes                   | Low      |



$\overline{RQ}/\overline{GT}_0$  as an output to inform the new bus master. A second low-level pulse of one clock period does this. The 8088 then enters a hold acknowledge state until the new bus master is ready to give the system bus back. This is done by a third low-level, one-clock-period pulse on  $\overline{RQ}/\overline{GT}_0$  (from the new bus master back to the 8088). So, the normally high  $\overline{RQ}/\overline{GT}_0$  signal went from being an input to an output and back to an input again.

$\overline{LOCK}$  is an active low output used to inform other possible bus masters that the 8088's system bus is not available for takeover. A special instruction called a **LOCK prefix** activates the  $\overline{LOCK}$  signal, which will go back to its inactive state after execution of the instruction following the LOCK prefix instruction. If an  $\overline{RQ}/\overline{GT}$  sequence is requested while the bus is  $\overline{LOCK}$ ed, it will not be acted on until the bus becomes un $\overline{LOCK}$ ed.

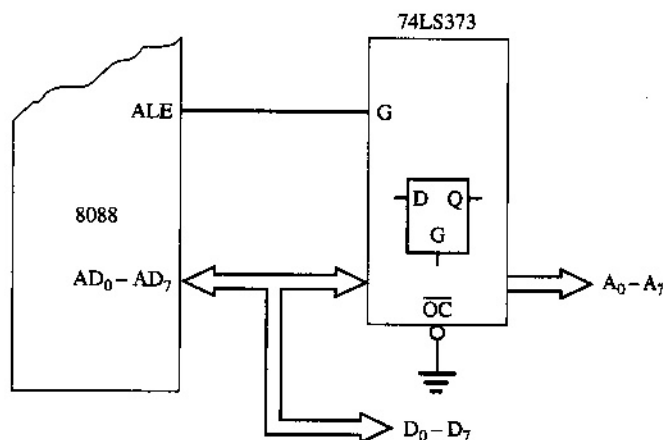
### ALE, $\overline{DEN}$ , $\overline{DT}/\overline{R}$ , $\overline{WR}$ , $\overline{RD}$ , and $\overline{IO}/\overline{M}$

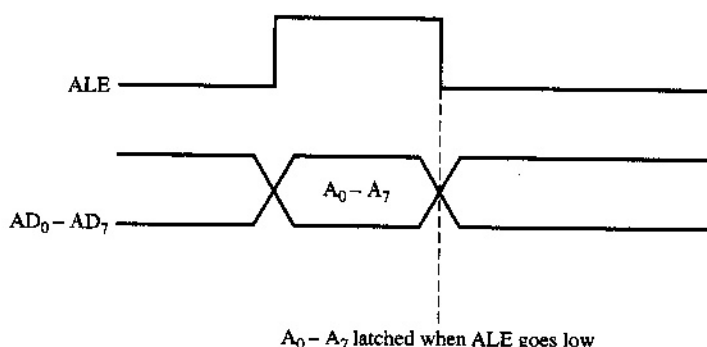
Five of these signals operate in minmode. Only  $\overline{RD}$  operates in both minmode and maxmode. Some of these signals are used to interface the 8088 with 8085 peripherals. All six signals are outputs, and all but ALE will be tri-stated during hold acknowledge.

ALE (*address latch enable*) is an output signal used to demultiplex the 8088's address/data bus. ALE is usually used to control an external latch capable of storing the lower 8 bits of the processor's address bus. ALE goes high to indicate that the 8088 is outputting address information. Take another look at Figure 8.1 and note the signals  $AD_0$  through  $AD_7$ . These eight signals perform two functions. They operate as  $A_0$  through  $A_7$  during the beginning of a bus cycle, and as  $D_0$  through  $D_7$  for the rest of the bus cycle. So,  $AD_0$  through  $AD_7$  are constantly switching back and forth between address bus mode and data bus mode. The processor uses ALE to indicate when  $AD_0$  through  $AD_7$  contain address information. Figure 8.5 shows one way the address/data lines may be demultiplexed. A 74LS373 octal D latch is used to capture  $A_0$  through  $A_7$  when enabled by ALE. This capture is illustrated by Figure 8.6. We will see another example of how the address/data lines are demultiplexed when we cover the 8288 bus controller in Section 8.5.

$\overline{DEN}$  (*data enable*) is an active low output used in a minmode system to control a bidirectional buffer (also called a **transceiver**) connected to the processor's data bus. The bidirectional buffer is used to buffer data going both ways on the data bus (into the 8088 and out of the 8088). Because there may be times when we want to disconnect the data bus from the processor (e.g., when we are sharing memory with another processor),  $\overline{DEN}$  gives us a way to turn the transceiver off. In Section 8.5, we will see how  $\overline{DEN}$  is generated in a maxmode system.

**FIGURE 8.5** Demultiplexing the 8088's address/data bus



**FIGURE 8.6** Latching the lower address byte

$\overline{DT/\overline{R}}$  (*data transmit/receive*) is an output used in a minmode system to control the direction of data flow in the bidirectional buffer used on the data bus. When  $\overline{DT/\overline{R}}$  is low, data should flow into the 8088. When  $\overline{DT/\overline{R}}$  is high, the 8088 is outputting data. We will see an example of how  $\overline{DT/\overline{R}}$  is used in Section 8.5.

$\overline{WR}$  (write) is an active low output used to indicate when the processor is writing to a memory or I/O location.

$\overline{RD}$  (read) is an active low output used to indicate when the processor is reading a memory or I/O location.

$\overline{IO/\overline{M}}$  is an output that indicates whether the current bus cycle is a memory access or an I/O access. A memory access is indicated by a logic zero on  $\overline{IO/\overline{M}}$ , and I/O access by a logic one.  $\overline{IO/\overline{M}}$  is used with  $\overline{RD}$  and  $\overline{WR}$  to generate separate read and write signals for memory and I/O devices. Figure 8.7 shows how OR gates can be used to decode different read/write operations. In Section 8.5 we will see that some of the signals generated by the 8288 bus controller are identical in operation to those generated in Figure 8.7.

### A<sub>8</sub> Through A<sub>19</sub>, AD<sub>0</sub> Through AD<sub>7</sub>

These signals constitute the 8088's 20-bit address bus and 8-bit data bus. AD<sub>0</sub> through AD<sub>7</sub> are the processor's multiplexed address/data bus. These signals are multiplexed because there are only 40 pins in the 8088's package, but more than 40 signals. These signals can behave like A<sub>0</sub> through A<sub>7</sub> or D<sub>0</sub> through D<sub>7</sub> (depending on the state of ALE). Usually an external latch is used to store address information when it is present. The other address lines, A<sub>8</sub> through A<sub>19</sub>, make up the rest of the 8088's address bus. These address lines do

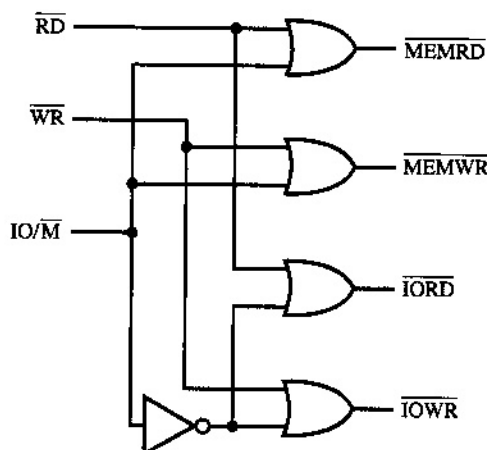
**FIGURE 8.7** Decoding 8088 memory and I/O read/write signals

TABLE 8.3 8088 signal summary

| Signal                                                                | Input | Output | Tri-state | Minmode | Maxmode |
|-----------------------------------------------------------------------|-------|--------|-----------|---------|---------|
| CLK                                                                   | ✓     |        |           | ✓       | ✓       |
| MN/ $\overline{\text{MX}}$                                            | ✓     |        |           | ✓       | ✓       |
| $\overline{\text{S}}_0, \overline{\text{S}}_1, \overline{\text{S}}_2$ |       | ✓      | ✓         |         | ✓       |
| RESET                                                                 | ✓     |        |           | ✓       | ✓       |
| READY                                                                 | ✓     |        |           | ✓       | ✓       |
| HOLD                                                                  | ✓     |        |           | ✓       |         |
| HLDA                                                                  |       | ✓      |           | ✓       |         |
| NMI                                                                   | ✓     |        |           | ✓       | ✓       |
| INTR                                                                  | ✓     |        |           | ✓       | ✓       |
| $\overline{\text{INTA}}$                                              |       | ✓      |           | ✓       |         |
| $\text{RQ/GT}_0$                                                      | ✓     | ✓      |           |         | ✓       |
| $\text{RQ/GT}_1$                                                      | ✓     | ✓      |           |         | ✓       |
| $\overline{\text{LOCK}}$                                              |       | ✓      | ✓         |         | ✓       |
| ALE                                                                   |       | ✓      |           | ✓       |         |
| $\overline{\text{DEN}}$                                               |       | ✓      | ✓         | ✓       |         |
| $\text{DT}/\overline{\text{R}}$                                       |       | ✓      | ✓         | ✓       |         |
| $\overline{\text{WR}}$                                                |       | ✓      | ✓         | ✓       |         |
| $\overline{\text{RD}}$                                                |       | ✓      | ✓         | ✓       | ✓       |
| $\text{IO}/\overline{\text{M}}$                                       |       | ✓      | ✓         | ✓       |         |
| $\text{AD}_0\text{--AD}_7$                                            | ✓     | ✓      | ✓         | ✓       | ✓       |
| $\text{A}_8\text{--A}_{19}$                                           |       | ✓      | ✓         | ✓       | ✓       |

not have to be latched because their outputs are always valid. Together, the twenty address lines can access 1MB of memory. The ability to directly address this many address locations provides the programmer with a very flexible programming environment. Older systems required that special software be used to manage memory in "pages" that were some portion of the system's main address space. This software is not even needed now in some applications because of the great increase in address lines and, hence, memory locations. The 8088 has special segment registers that manage 64KB blocks of memory starting on any 16-byte boundary, so it is relatively easy to manage the system's addressing space.

$\text{A}_0$  through  $\text{A}_{15}$  have a dual role: they are also used to access I/O devices (depending on the state of  $\text{IO}/\overline{\text{M}}$ ). Sixteen address lines provide for up to 65,536 I/O locations. We will see more about memory and I/O devices in Chapters 9 and 10.

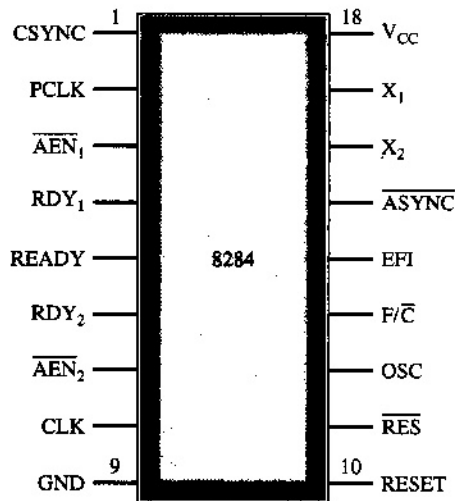
### Signal Summary

Table 8.3 summarizes all of the signals we have just covered. It shows whether a signal is an input or an output (or both), if it has tri-state capability, and in which mode (min or max) it is active. In the following sections, we will see how many of these signals are used in an actual system.

## 8.4 THE 8284 CLOCK GENERATOR

As we saw in the previous section, CLK, RESET, and READY are three of the most important signals in the overall operation of the processor. A small number of logic gates would be needed to implement the correct signals on CLK and RESET. Intel provides most

FIGURE 8.8 8284 clock generator



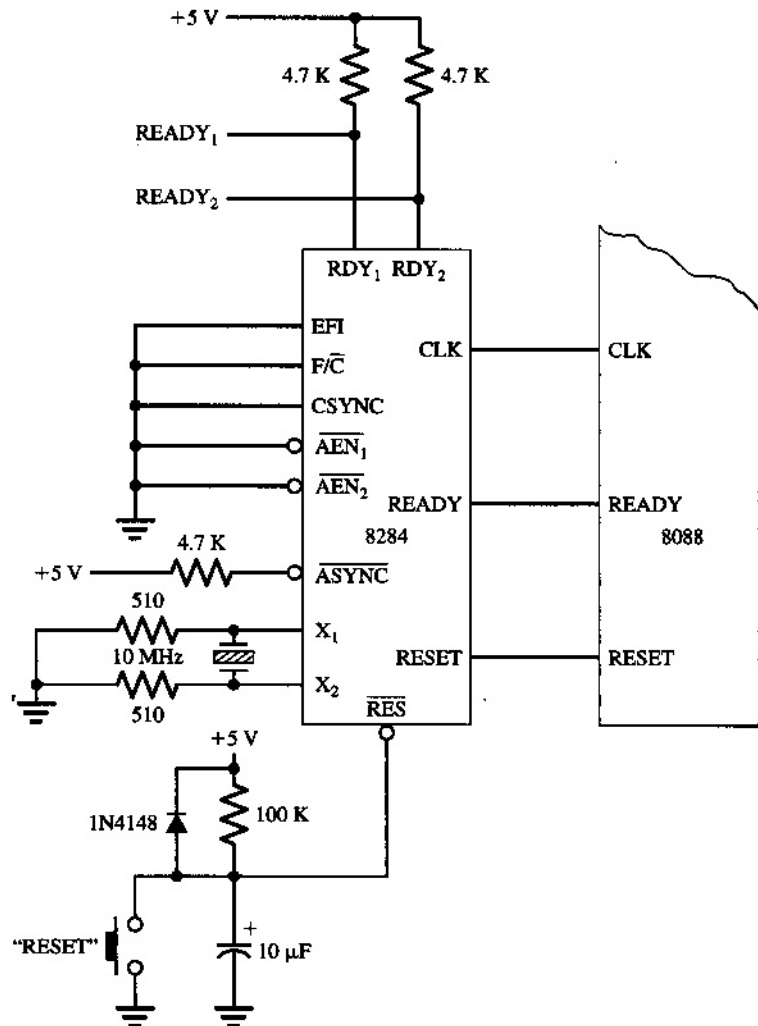
of this circuitry in an 18-pin DIP called the 8284 clock generator. Figure 8.8 shows a pin diagram of the 8284. Two of the pins, X<sub>1</sub> and X<sub>2</sub>, are meant to be directly connected to a crystal. The internal clock circuitry of the 8284 then generates the proper CLK signal for the 8088. Because the frequency of CLK will be one-third the crystal frequency (due to an internal frequency divider), the 8284 provides the OSC output, whose frequency is the same as the crystal's. When using a crystal, the 8284's  $\text{F}/\overline{\text{C}}$  input must be grounded. When  $\text{F}/\overline{\text{C}}$  is high, the 8284's EFI pin must be connected to an external oscillator, or some type of timing circuit that generates a TTL signal at the proper frequency. In this case, the frequency of OSC will match the frequency of the EFI input. In addition to  $\text{F}/\overline{\text{C}}$ , a second signal, CSYNC, is used to provide clock synchronization when EFI is used as the frequency source. When a crystal is used, CSYNC must be taken low.

Another signal,  $\overline{\text{RES}}$ , is the 8284's reset input. This input is normally connected to a simple resistor-capacitor network. When power is first applied, the RC network allows a logic zero to remain on the  $\overline{\text{RES}}$  input for a short period of time. The internal circuitry of the 8284 uses  $\overline{\text{RES}}$  to generate the processor's RESET signal. Together, the RC network and the 8284 provide a **power-on reset** signal to the 8088.

Two ready inputs are provided on the 8284, RDY<sub>1</sub> and RDY<sub>2</sub>. Together with  $\overline{\text{AEN}}_1$  and  $\overline{\text{AEN}}_2$ , the 8284 generates the processor's READY signal. In a system where memory and I/O devices are used, one RDY signal can be used with the memory circuitry and the other for the I/O circuitry. The  $\overline{\text{AEN}}$  signals are used as *qualifiers* for the RDY inputs. For example, to use RDY<sub>1</sub>,  $\overline{\text{AEN}}_1$  must be low. It may be necessary to generate a **wait state** in the system, due to the use of slow memory or I/O devices. The RDY inputs are designed for this purpose. A fifth signal,  $\overline{\text{ASYNC}}$ , is used to select the number of stages of synchronization on READY. Only one stage is used when  $\overline{\text{ASYNC}}$  is high.

Many of the devices in an 8088 system require a clock that is slower than the processor's (the UART is one example). The PCLK output of the 8284 has a frequency that is one-sixth that of the crystal (or EFI) frequency. Together with EFI,  $\text{F}/\overline{\text{C}}$ , and X<sub>1</sub> and X<sub>2</sub>, we have a number of ways of controlling the timing of our system with the 8284. Figure 8.9 shows one way the 8284 can be connected to the 8088. Notice that a 10-MHz crystal is connected to X<sub>1</sub> and X<sub>2</sub>, and that  $\text{F}/\overline{\text{C}}$  is grounded. The 8284 will thus generate a CLK frequency just over 3.3 MHz! How fast would the processor run if a 12-MHz crystal were used?

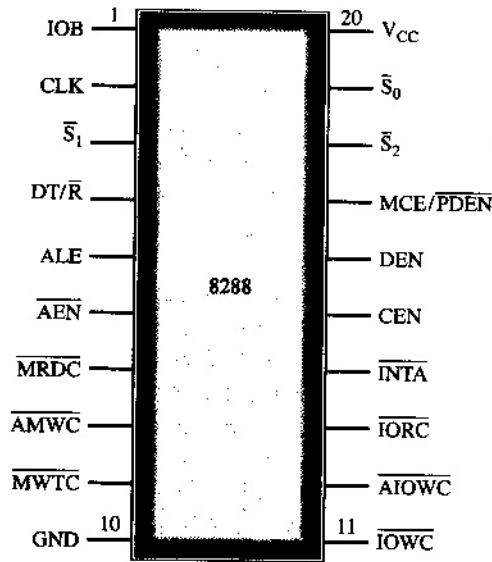
**FIGURE 8.9** An actual 8284 timing circuit



The RC network on  $\overline{\text{RES}}$  is used to provide the power-on reset signal. A push button is connected across the capacitor to allow manual resets with power still applied. Figure 8.9 represents the timing circuitry used in the single board computer of Chapter 12.

## 8.5 THE 8288 BUS CONTROLLER

We have already seen a number of differences brought about by the 8088's minmode and maxmode operation. To get the maxmode signals, we sacrifice other signals that have important uses. For example, in maxmode we get  $\overline{\text{RQ}}/\overline{\text{GT}}$  signals but have to give up  $\text{HOLD}$  and  $\text{HLDA}$ . Other signals are replaced as well, leaving us with no ability to decode read/write accesses in maxmode *unless we use the status outputs*. Because  $\overline{\text{S}}_0$ ,  $\overline{\text{S}}_1$ , and  $\overline{\text{S}}_2$  are available in maxmode, we use them as inputs to a special 8288 bus controller, which in turn decodes the missing signals. Figure 8.10 shows a pin diagram of the 8288. Notice the three status inputs and also the three *outputs*  $\text{ALE}$ ,  $\text{DEN}$ , and  $\text{DT}/\overline{\text{R}}$ . Three of the signals that were

**FIGURE 8.10** 8288 bus controller

lost when we switched to maxmode are now being generated by the 8288. Other signals that we require are  $\overline{\text{MRDC}}$  (*memory-read command*),  $\overline{\text{MWTC}}$  (*memory-write command*),  $\overline{\text{IORC}}$  (*I/O-read command*), and  $\overline{\text{IOWC}}$  (*I/O-write command*). These signals control the memory and I/O devices. Two additional signals,  $\overline{\text{AMWC}}$  (*advanced memory-write command*) and  $\overline{\text{AIOWC}}$  (*advanced I/O-write command*), are provided to give memory and I/O circuitry advance warning that they will be accessed (so that they can get a jump on the decoding process).

$\overline{\text{INTA}}$  is decoded and also available as an output. Remember that this signal indicates that the processor received an interrupt request on  $\text{INTR}$ .

The 8288 can be used to control the operation of the buffers and bidirectional latches used on the address and data buses. The ALE, DEN, and  $\text{DT}/\overline{\text{R}}$  outputs of the 8288 operate like the actual minmode signals, with the exception of DEN being inverted. Figure 8.11 shows how the 8288 connects to the 8088 and its bus circuitry in a typical application. The inverter on the DEN output is used to enable the 8286 data bus transceiver.  $\text{DT}/\overline{\text{R}}$  controls the direction of data in the 8286. The latching of the lower eight address lines is accomplished by ALE and an 8282 octal latch. Address lines  $\text{A}_8$  through  $\text{A}_{15}$  are buffered with a 74LS244 octal buffer so they have the ability to drive the address lines of the memory and I/O devices. Similar buffering is also needed on address lines  $\text{A}_{16}$  through  $\text{A}_{19}$ , which are not shown because they are unused in the application (not unusual, because many minimal systems do not require more than 64KB of memory).

The 8288 is capable of additional operations, one being I/O-bus mode. Here the IOB (*I/O bus mode*) input must be high, CEN (*command enable*) must be high, and  $\overline{\text{AEN}}$  (*address enable*) should be low to enable the buses. These signals will also make  $\text{MCE}/\overline{\text{PDEN}}$  operate like DEN, except only during I/O operations. The 8288 can thus be used to control a special I/O bus, if necessary. IOB, CEN, and  $\overline{\text{AEN}}$  can also configure  $\text{MCE}/\overline{\text{PDEN}}$  to operate as MCE (*master cascade enable*), a signal used during interrupt cycles. Other configurations allow multiple 8288s (with their own 8088s) to share a common system bus. In Figure 8.11, IOB and  $\overline{\text{AEN}}$  are grounded, and CEN is pulled high. This selects system-bus mode and makes  $\text{MCE}/\overline{\text{PDEN}}$  operate as MCE. The system bus cannot be disabled in this configuration.

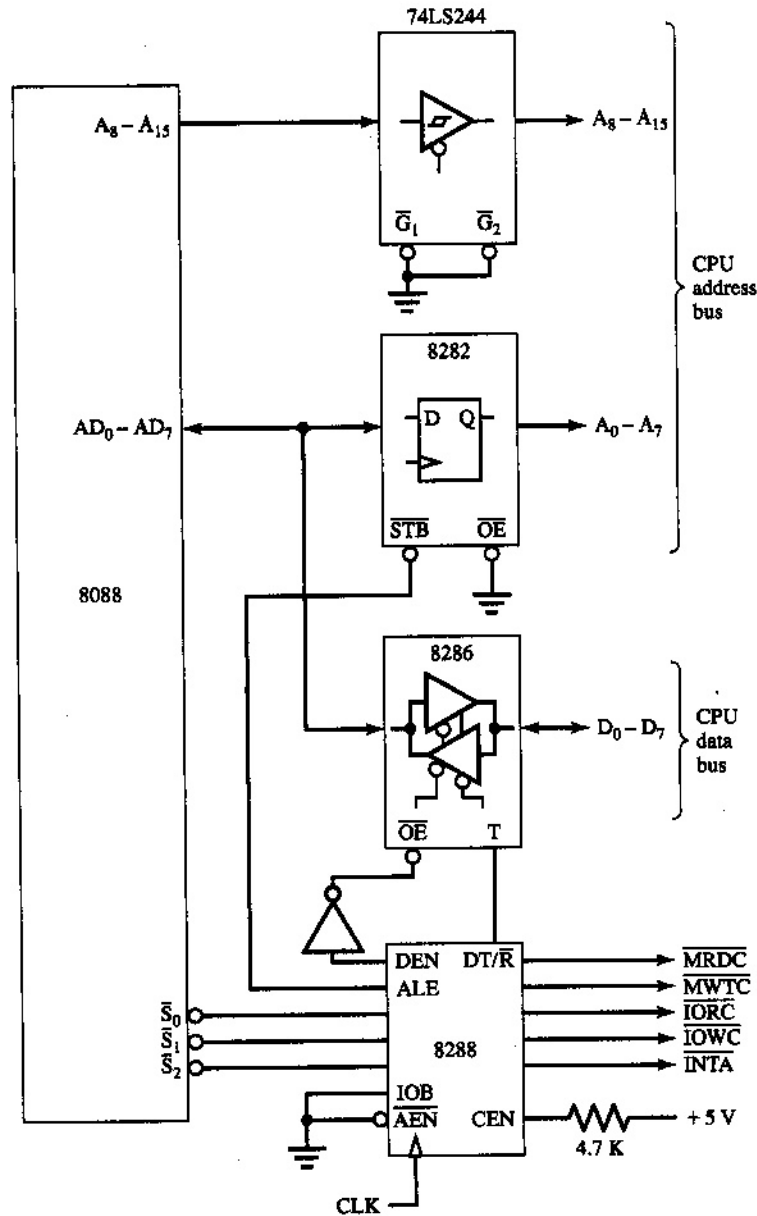


FIGURE 8.11 A sample maxmode 8088 system's bus logic

## 8.6 SYSTEM TIMING DIAGRAMS

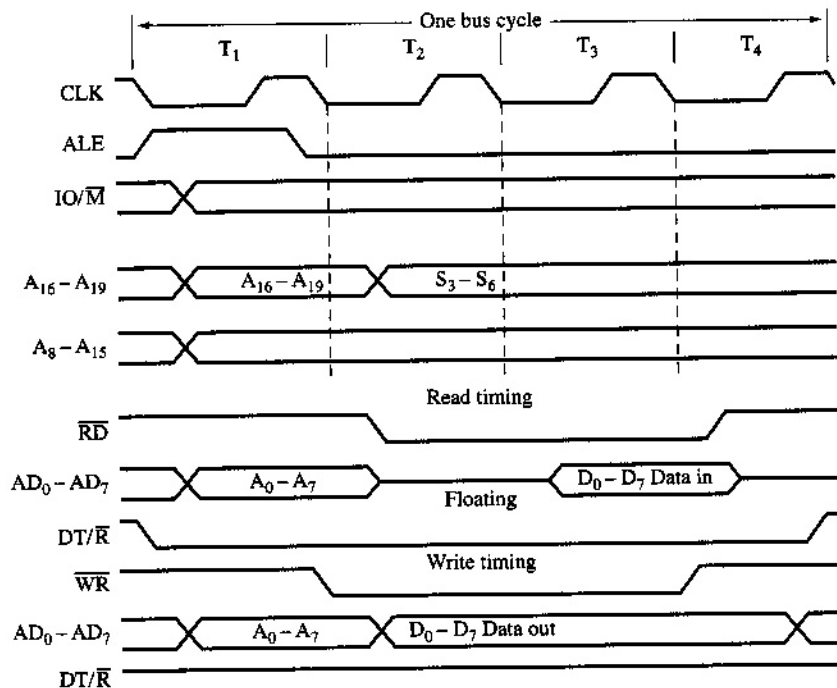
When timing is a critical issue in the design of a new 8088-based system, it pays to know how to interpret the CPU timing diagrams supplied by the manufacturer. This section will provide more details on the processor's control and timing signals by analyzing a number of timing scenarios.

We will start with the timing of a typical bus cycle.

## CPU Bus Cycle

The 8088 routinely performs a number of different types of bus cycles. Memory read, memory write, I/O read, I/O write, and interrupt acknowledge are most of the important ones. A bus cycle is composed of four or more **T states**, numbered  $T_1$ ,  $T_2$ ,  $T_3$ , and  $T_4$ , where each T state represents the time of one CLK period. For example, if an 8088 has a CLK frequency of 4 MHz, the time of one T state is 250 ns, and a four-state bus cycle takes 1  $\mu$ s. A T state is the smallest amount of time in which the processor can perform *any* function. Because all instructions are composed of a number of T states, instruction execution time depends on the CLK frequency. Figure 8.12 shows the simplified timing for a CPU bus cycle in a minmode system. Each new CLK pulse defines a new T state. Because the CLK must have a 33 percent duty cycle, you will notice that the CLK signal is low for two-thirds of its period and high for the last third. Note that ALE pulses high during the first T state. This means that the multiplexed address/data lines  $AD_0$  through  $AD_7$  are in address mode and should be latched by external circuitry. During  $T_1$  we also see address lines  $A_8$  through  $A_{19}$  changing to their next state.  $IO/\overline{M}$  and  $DT/\overline{R}$  may also need to change state during  $T_1$ , depending on the type of access (memory or I/O) and the direction of data on the bus (read or write). During  $T_2$ , the processor's  $\overline{RD}$  and  $\overline{WR}$  lines change state, in preparation for the data read or write that will occur in state  $T_3$ . What else happens in  $T_2$ ? The upper four address lines change state to become four additional status outputs. Generally, these status bits indicate which type of segment register is being used during the bus cycle.  $T_3$  is the state in which the selected memory or I/O device should complete the data transfer. If the accessed memory or I/O logic cannot complete its job during  $T_3$ , it can extend the bus cycle via the processor's READY logic. The 8088 will insert wait states between  $T_3$  and  $T_4$  until READY indicates that the cycle may continue normally. Wait states have a duration of one CLK period also, so the time of any bus cycle will always be a multiple of CLK

**FIGURE 8.12** Simplified bus cycle timing in a minmode system





periods. During the end of  $T_4$ , all control signals and buses switch back to their inactive levels in preparation for the next cycle.

The processor may not automatically begin a new bus cycle after completing the current one. Suppose that the most recent bus cycle was the final fetch cycle for a multiply instruction. During the many CLK cycles it will take to internally perform the binary multiplication, the processor has no use for the bus, which may *idle* until completion of the multiply instruction. During this idle time, other devices are free to use the processor's bus, as long as they are done with it by the time the processor resumes its next bus cycle.

## Interrupt Acknowledge Cycle

A minmode system uses one bus cycle to perform an interrupt acknowledge. Figure 8.13(a) shows the events that occur in this cycle. During  $T_1$  the processor tri-states the address bus. During  $T_2$  the  $\overline{INTA}$  output is asserted, remaining low until it becomes inactive in state  $T_4$ . The interrupting device should respond to activity on  $\overline{INTA}$  by placing an 8-bit **interrupt type** onto the data bus, which will be captured by the processor before it deactivates  $\overline{INTA}$ .

In a maxmode system, two bus cycles are used for interrupt acknowledge. Figure 8.13(b) shows the events that occur in each cycle. In the first cycle the address bus is again placed into a high impedance state. In addition,  $\overline{LOCK}$  is asserted to inform other devices on the system bus that they cannot take over the bus until completion of the second cycle.  $\overline{INTA}$  is asserted twice, once during each cycle. Special interrupt peripherals exist that have been designed to respond to the interrupt acknowledge cycles. One example is the 8259 programmable interrupt controller, which we will examine in Chapter 11.

## HOLD/HLDA Timing

From Section 8.3, we know that it is possible to place the 8088 into a HOLD state (the processor idles) by asserting HOLD. The processor will respond by placing its buses in

**FIGURE 8.13** Interrupt acknowledge timing

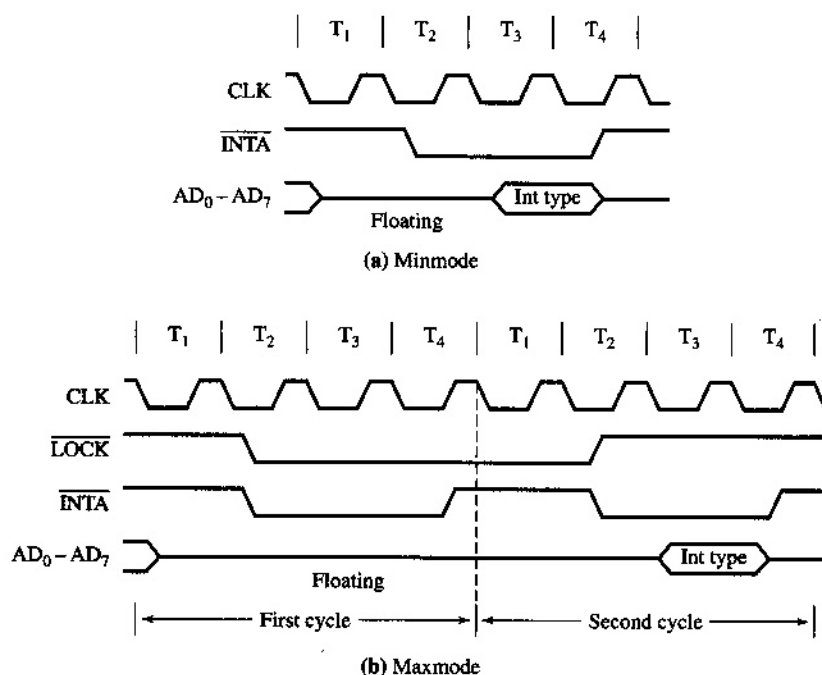
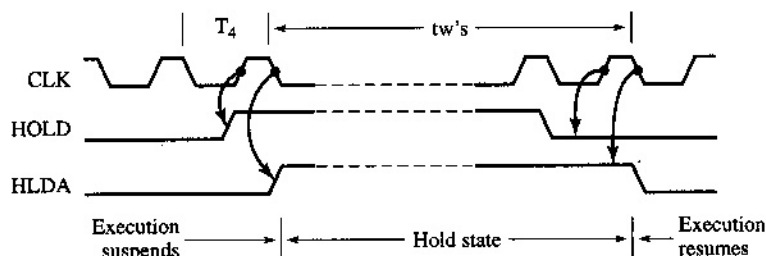


FIGURE 8.14 HOLD timing



the high impedance state and will acknowledge that it is in a HOLD state with HLDA. Figure 8.14 shows the corresponding timing relationship between HOLD and HLDA. The state of the HOLD input is sampled on every rising edge of CLK. If it is high, the processor will activate HLDA at the end of  $T_4$  (or at the end of the current idle state). The processor may need a number of internal states to complete what it was doing (e.g., finishing a multiply instruction), but once HLDA is asserted it will no longer use its buses until it sees HOLD go low. While the 8088 is in the hold state, it will continue to sample HOLD on each rising edge of CLK. When HOLD does go low, the processor will reset HLDA at the beginning of the next T state and resume program execution. A technique that activates HOLD at the end of every instruction, so that the processor's buses can be examined, is called **single-stepping**. So, a simple circuit could be used that would allow only one instruction to be executed by the 8088 each time a button was pressed. Think of how you might design such a single-step circuit (starting with a flip-flop might help).

## 8.7 EARLY PERSONAL COMPUTER BUS STANDARDS

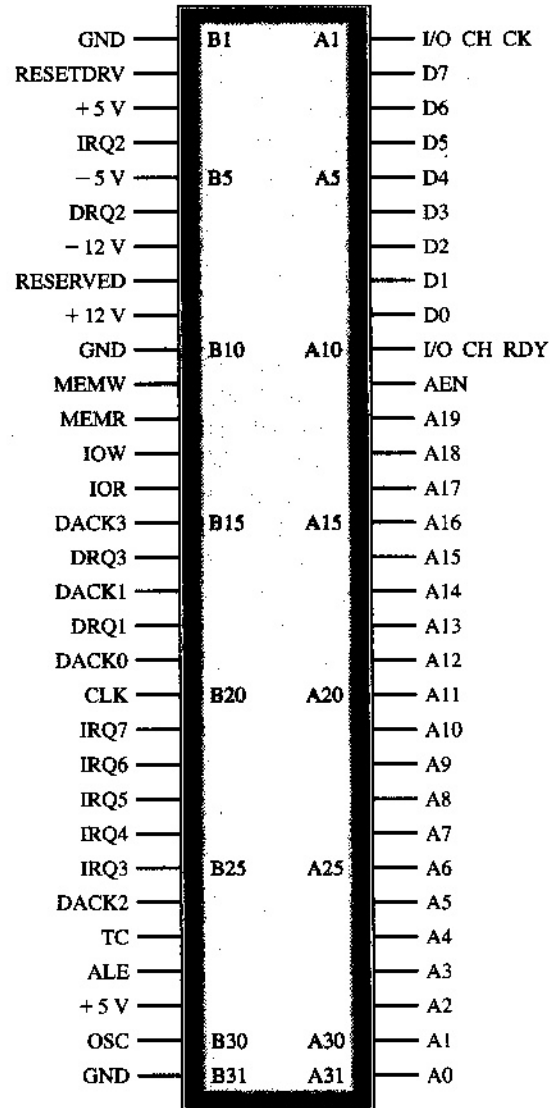
Hardware interfacing is accomplished by the use of a standard connection bus on the motherboard of the machine. If you were to make a list of which processor signals you might need to connect your computer to a disk drive, or a serial data terminal, or a video card, what signals would you choose? Certainly we would need data lines for the exchange of information, and address lines to select the device we wish to communicate with. Then, we would also need all the necessary control signals, such as memory and I/O read/write, and ALE.

The list we have been considering was made up many years ago and agreed upon by all people involved in the personal computer business. Figure 8.15 shows the pin assignments for a standard 62-pin connector found on early PC motherboards that allow expansion with the 8088 microprocessor. In addition to the signals already discussed, provisions for power and many levels of interrupts are also included. Do not confuse address lines  $A_0$  through  $A_{19}$  with their respective connector pin names.

This connector is referred to as an ISA (Industry Standard Architecture) connector. It is also known as the PC-XT connector. Because it was designed for use with the original 8088 microprocessor, it contains only an 8-bit data path ( $D_0$  through  $D_7$ ).

When the 8086 and 80286 microprocessors found their way onto motherboards, it was necessary to provide a 16-bit data path to the connectors. A 36-pin extension connector was added to the original 62-pin connector, containing eight new data lines ( $D_8$  through  $D_{15}$ ) and additional interrupt and DMA signals. This 16-bit ISA connector is commonly called the PC-AT connector (the computer it was first used in). Figure 8.16 shows the structure of the PC-AT connector pair. Plug-in cards that require the full 16-bit data bus have two edges with connector traces. Older 8-bit cards have only one edge of connector traces, and do not use the additional 36-pin socket.

**FIGURE 8.15** Pin assignments for the ISA connector, the original PC expansion bus

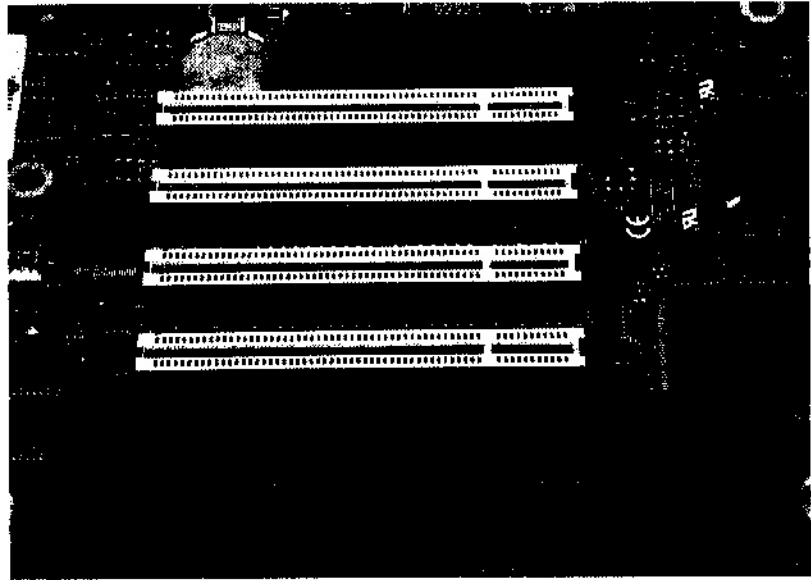


An improvement to the PC-AT connector is the EISA (Extended ISA) standard. This connector supports 80386, 80486, and Pentium microprocessors by providing a full 32-bit data bus. Additional address lines are provided to allow memory expansion as well. The EISA connector is very similar to the PC-AT connector, with the original 62-pin and 38-pin definitions unchanged, and new connector pins added *between* the old connector pins, and at a lower depth! This prevents an old ISA board from making contact with the newer EISA signals. Three special bus-controlling chips are used to manage data transfers through the EISA connectors.

Modern expansion buses, such as PCI and USB, will be explored in Chapter 11.

Overall, we can see that connector technology has advanced to keep up with the improvements in microprocessors.

**FIGURE 8.16** PC Expansion connectors (one AGP, four PCI, and two PC-AT connectors). AGP (Accelerated Graphics Port) is a dedicated video adapter connector. PCI (Peripheral Component Interconnect) is an advanced, high-speed 32-bit bus.



## 8.8 TROUBLESHOOTING TECHNIQUES

The hardware operation of the 8088 is complex and requires great patience during design or troubleshooting sessions. Your basic knowledge of 8088 hardware architecture should include operation of all its signals, and their functional groupings:

- System signals CLK, RESET, READY, HOLD, HLDA, and  $\overline{MN}/\overline{MX}$ .
- The signals involved with memory and I/O accesses. These are data/address lines  $AD_0$  through  $AD_7$ , address lines  $A_8$  through  $A_{19}$ , ALE,  $\overline{RD}$ ,  $\overline{WR}$ , DEN,  $DT/\overline{R}$ , and  $IO/\overline{M}$ .
- The signals involved with interrupts. These are NMI, INTR, and  $\overline{INTA}$ .
- The processor status signals  $\overline{S}_0$  through  $\overline{S}_2$ .

The groupings are important because they point the way at the beginning of a design or while troubleshooting. Knowing the relationships between signals allows you to make informed decisions about what to do next, such as how to design a memory address decoder or determine why the READY input stays low in a faulty system.

## SUMMARY

In this chapter, we examined the operation of the 8088's hardware signals. We saw that there are actually two sets of processor signals, one for minmode operation and the other for maxmode operation. Minmode signals can be directly decoded by memory and I/O circuits, resulting in a system with minimal hardware requirements. Maxmode systems are generally more complicated, resulting from the use of the 8288 bus controller and the new maxmode signals that allow for bus grants.

We saw that the 8088 can access 1MB of memory, and that it contains two hardware interrupt mechanisms and uses a multiplexed address/data bus. Hardware examples showing how the bus is demultiplexed, how memory and I/O control signals are generated, and

how the 8284 clock generator and 8288 bus controller are used were also given. The chapter finished with a short look at CPU timing diagrams and the interface connectors for PCs. In Chapters 9 through 12, we will draw on the information presented in this chapter, so use it as a handy reference.

## STUDY QUESTIONS

1. What are some of the differences between minmode and maxmode operation?
2. How is the 8088 put into minmode operation?
3. Sketch four cycles of the 8088's CLK signal if its frequency is 3.125 MHz. Compute the time, in nanoseconds, of the low-portion and high-portion of each cycle.
4. If an instruction requires 20 states to complete, what is the instruction execution time if the CLK period is 250 ns?
5. What is the processor's CLK frequency if a 10-MHz crystal is used with the 8284?
6. Design a circuit that will turn an LED on when the status outputs  $\bar{S}_0$  through  $\bar{S}_2$  indicate the HALT state.
7. Show how a 3- to 8-line decoder (74LS138) can be used to decode the status assignments found in Table 8.1. The 74LS138 is shown in Figure 8.17.

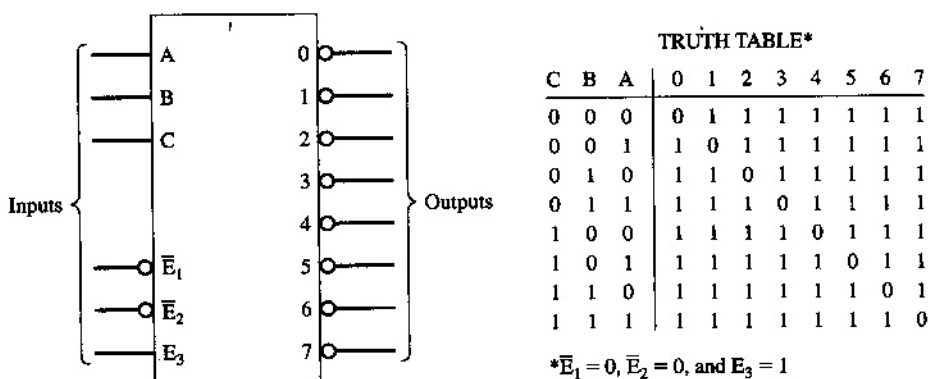


FIGURE 8.17 For Question 7

8. What address is used first after a RESET?
9. Why do 8088-based systems need EPROM at the high end of memory?
10. How is the 8088 slowed down enough to communicate with a slow memory device?
11. What signal(s) might a second 8088 use to take over the buses of another 8088?
12. What are the differences between NMI and INTR?
13. What would happen if INTR were stuck high?
14. What is so special about the operation of ALE?
15. What other types of data latches could be used in Figure 8.5 in place of the 74LS373?
16. If a minimal 8088 system uses only address lines  $A_0$  through  $A_{17}$ , how many memory locations is it capable of accessing?
17. Show how the address 3A8C4 would be represented on the 20-bit address bus.
18. What is one advantage of a multiplexed address/data bus?
19. What is one disadvantage of a multiplexed address/data bus?
20. Show how NAND gates could be used to decode the signals in Figure 8.7.

21. Show the connections needed to feed a 12-MHz clock signal into the 8284 (note the lack of external crystal now).
22. In Figure 8.11, why does the data bus require a bidirectional driver?
23. During which state is ALE active?
24. Design a single-step circuit that allows only one instruction to execute each time a push button is pressed.
25. When two or more 8088s share a common memory system, what do you expect happens to the overall bus activity of the system? What about the bus activity of each processor?
26. What are the states of  $\overline{IO/\overline{M}}$ ,  $\overline{RD}$ , and  $\overline{WR}$  when the 8088 is:
  - (a) writing to memory
  - (b) reading from an I/O device
27. What is placed on  $D_0$  through  $D_7$  during an interrupt acknowledge cycle?
28. Explain the differences between ISA and EISA connectors.
29. How can two 8088 CPUs be connected so that they *share* a 64KB RAM and have separate 16KB EPROMs?
30. Design a circuit that will reset the processor if an NMI occurs while the 8088 is being held (via HOLD).

---

## CHAPTER 9

---

# Memory System Design

---

### OBJECTIVES

In this chapter you will learn about:

- The importance of bus buffering
- How the 8088 addresses (accesses) memory
- The design of custom memory address decoders
- The difference between full- and partial-address decoding
- How wait states may be inserted into memory-read/write cycles
- The differences between static and dynamic RAMs
- How a dynamic RAM is addressed and what purpose refresh cycles serve
- DMA (direct memory access)

### KEY TERMS

|                  |                        |                         |
|------------------|------------------------|-------------------------|
| Address bus      | Data bus               | Memory-mapped I/O       |
| Base address     | Direct memory access   | Partial-address decoder |
| Buffer           | Dynamic RAM            | Refresh                 |
| Control bus      | Master device          | Slave device            |
| Damping resistor | Memory address decoder |                         |

---

## 9.1 INTRODUCTION

The internal data storage capacity of any microprocessor, with the exception of single-chip microprocessors, is severely limited. The 8088 itself has only a handful of 16-bit locations in which it can store numbers, and these locations are the actual data registers available to the programmer. The need for larger, external memories quickly becomes apparent, especially if an application involves number crunching or word processing. The purpose of this chapter is to explore ways of adding external memory to 8088-based systems. We will examine how the 8088's various control signals (ALE, DT/R, WR, RD, and IO/M) are used to supply memory-read and -write signals to read-only memories and both static and dynamic random access memories.

In addition, we will see how an external device called a bus master takes over control of the 8088's memory system during a process called direct memory access (DMA) that bypasses CPU control for faster transfers.

The information provided in this chapter should enable you to design future memory systems from scratch. Also, the concepts presented in this chapter, such as designing an address decoder, are easily extended to more advanced architectures, such as 32- and 64-bit-wide memory systems.

Section 9.2 explains the 8088's address and data buses. The importance of bus buffering is discussed in Section 9.3. Section 9.4 shows how the 8088 accesses memory, Section 9.5 covers the design of a memory address decoder, and Section 9.6 introduces the partial-address decoder. Section 9.7 explores the use of a shift register to generate wait states. Section 9.8 contains a complete 8KB RAM/EPROM memory. In Section 9.9, we show how dynamic RAM can be used with the 8088. Section 9.10 explains how DMA works. Memory-mapped I/O is covered in Section 9.11. Some hardware troubleshooting techniques are given in Section 9.12.

---

## 9.2 THE 8088 ADDRESS AND DATA BUSES

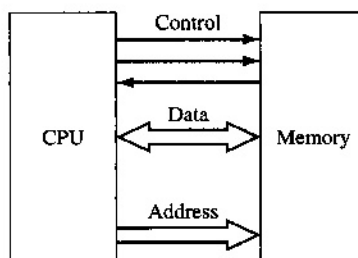
As previously discussed, the 8088 microprocessor has an 8-bit data bus, and a 20-bit address bus that can access 1MB of external memory. The lower eight address lines are multiplexed together with the eight data lines, resulting in signals  $AD_0$  through  $AD_7$ . In maximum mode, this multiplexed bus is decoded by external hardware, specifically by the 8288 bus controller covered in Chapter 8, with the aid of the processor's status outputs. The 8288 takes care of latching the lower eight address lines and controlling the direction of a bidirectional buffer on the data bus. In minimum mode, the processor outputs the necessary control signals directly (via ALE,  $IO/\overline{M}$ ,  $\overline{RD}$ ,  $\overline{WR}$ , and others). Thus, we end up with data lines  $D_0$  through  $D_7$  and address lines  $A_0$  through  $A_{19}$ . We will see that all of these signals are needed to communicate with the RAM and EPROM devices contained in the memory system.

---

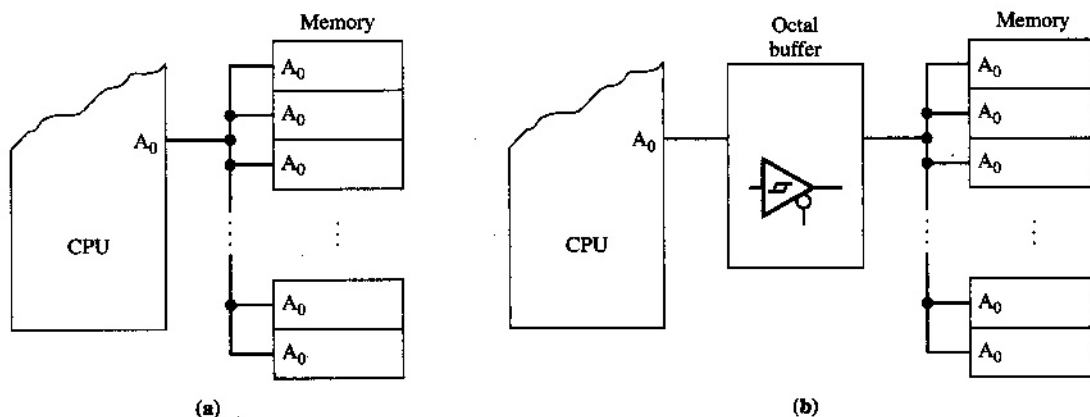
## 9.3 BUS BUFFERING

Every microprocessor-based memory system, whether EPROM or RAM, will have standard buses connecting it to the microprocessor whose functions are to direct the flow of information to and from the memory system. These buses are generally called the **control bus**, the **data bus**, and the **address bus**. Figure 9.1 shows the relationship between the

**FIGURE 9.1** Memory bus structure







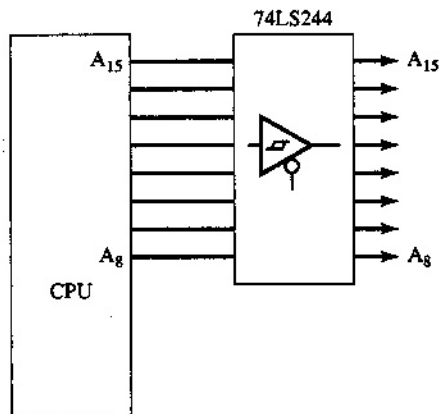
**FIGURE 9.2** Microprocessor address line: (a) cannot drive required number of memory devices; (b) drives all memory devices via octal buffer

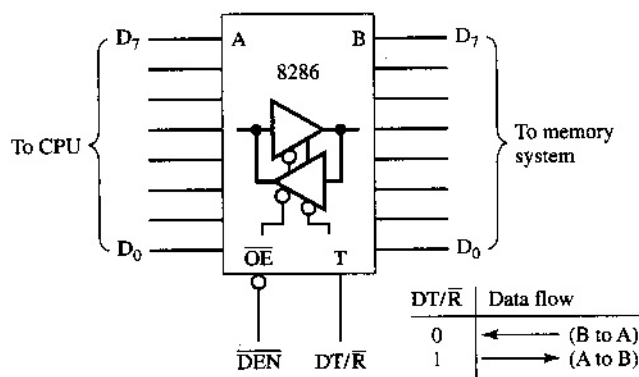
CPU, the buses, and the memory system. Note that the address bus is unidirectional (output only in this case), which means that data on the address bus goes one way, from the CPU to the memory system. The data and control buses, on the other hand, are bidirectional. Data may be written to or read from memory, hence the need for a bidirectional data bus. We will soon see why the control bus is also bidirectional.

Whether they are bidirectional or not, some care must be taken when the buses are connected to the memory section. It is possible to overload an address or data line by forcing it to drive too many loads. As always, it is important *not* to exceed the fanout of a digital output. If, for example, a certain TTL (transistor transistor logic) output is capable of sinking 2 mA, how many 0.4 mA inputs can it drive? The answer is five, which we get by dividing the output sink current by the required input current. If more than five inputs are connected, the output is overloaded and its ability to function properly is diminished. Clearly, the possibility of overloading the 8088's address or data buses exists when they are connected to external memory. For this reason, we will **buffer** the address and data buses. This concept is illustrated in Figure 9.2.

Figure 9.3 shows how address lines  $A_8$  through  $A_{15}$  are buffered by connecting them to a standard high-current buffer, the 74LS244 octal line driver/receiver. An address line on the 8088 is capable of sinking 2 mA all by itself. When the output of the 74LS244 is used

**FIGURE 9.3** Address bus buffering



**FIGURE 9.4** Data bus buffering

instead, the address line has an effective sink current of 24 mA. This means that twelve times as many gates can be driven. Buffering the address lines allows the CPU to drive all the devices in our memory system, without the added worry of overloading the address line.

Buffering the data bus is a little trickier because the data bus is bidirectional. Data must now be buffered in both directions. Figure 9.4 shows how this bidirectional buffering is accomplished. The 8286 is an octal bus transceiver. Data flow through this device is controlled by the T (*transmit*) input, which tells the buffer to pass data from left to right, or from right to left. Left-to-right data is CPU output data. Right-to-left data is considered CPU input data. The natural choice for controlling the direction of the 8286 is the 8088's DT/ $\overline{R}$  line, which always indicates the direction of data on the 8088 data bus. DT/ $\overline{R}$  is directly generated by the 8088 in minimum mode, and by the 8288 in maximum mode.  $\overline{DEN}$  is used to disable the 8286 when the bus is idle or contains address information.

In conclusion, then, remember that address and data buses should be buffered so that many gates can be connected to them instead of the few that can be directly driven by the unbuffered address or data line. All designs presented in this chapter will assume that the buses are already well buffered.

## 9.4 ACCESSING MEMORY

In addition to well-buffered address and data buses, a control bus must also be used to control the operation of the memory circuitry. The three operations we have to consider are the following:

1. Read data from memory
2. Write data to memory
3. Do not access memory

The first two cases represent data that gets transferred between the 8088 and memory. The third case occurs when the 8088 is performing some other duty (internal instruction execution, perhaps) and has no need for the memory system. Thus, it appears that the 8088 either accesses memory or does not access it. Does a processor signal (or group of signals) exist that tells external circuitry that the 8088 needs to use its memory? Yes, a number of signals indicate this need. In minimum mode, the processor will output a 0 on IO/ $\overline{M}$  to indicate that a memory reference is beginning. This signal is combined with  $\overline{RD}$  and  $\overline{WR}$  to form memory-read and memory-write signals for the memory system (as you can see in

**FIGURE 9.5** Decoding memory-read and -write signals in minimum mode

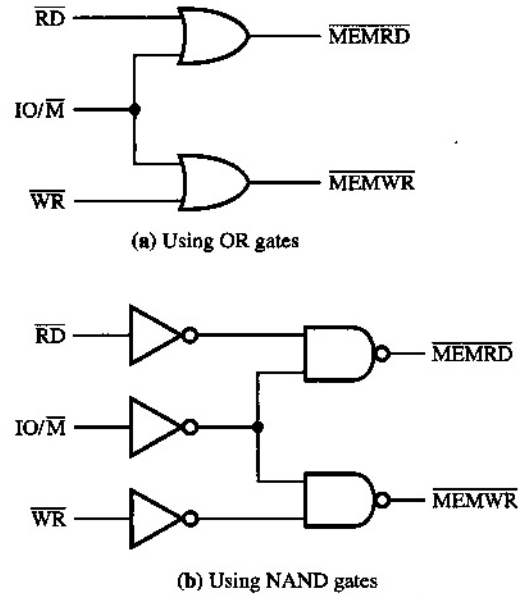


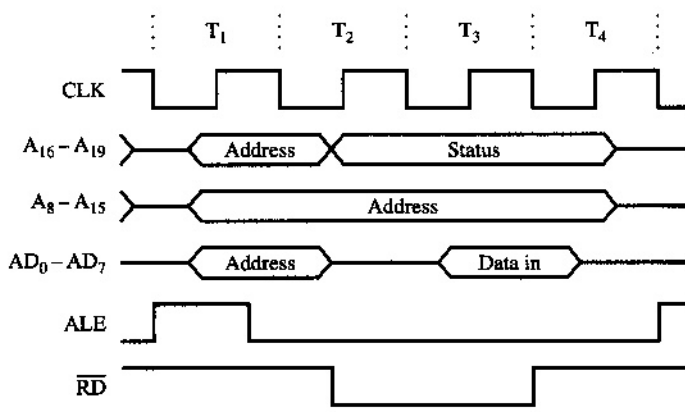
Figure 9.5). We use the active-low  $\overline{\text{MEMRD}}$  and  $\overline{\text{MEMWR}}$  signals to select and enable devices in the memory system.

In maximum mode, the 8288 bus controller decodes the processor status outputs and generates active-low  $\overline{\text{MRDC}}$  (memory-read command) and  $\overline{\text{MWTC}}$  (memory-write command) signals. The presence of a 0 on either signal indicates a memory access.

Figure 9.6 shows a simplified timing diagram for a memory-read cycle. The cycle is composed of four T states, with each T state equivalent to one clock cycle. In  $T_1$  the processor outputs a full 20-bit address on address lines  $A_8$  through  $A_{19}$  and  $AD_0$  through  $AD_7$ . ALE has also gone high, indicating that the multiplexed address/data bus contains address information. Because this is a memory access, the processor also has output a 0 on  $\text{IO}/\overline{\text{M}}$ , which will remain low for the duration of the bus cycle.

In state  $T_2$  the processor tri-states the multiplexed address/data bus in preparation for the data read which will take place in  $T_3$ . Address lines  $A_{16}$  through  $A_{19}$  switch over to status outputs  $S_3$  through  $S_6$ , and a zero is output on  $\overline{\text{RD}}$  to specify a memory-read cycle to external hardware. It is the responsibility of the memory circuitry to use  $\text{IO}/\overline{\text{M}}$ ,  $\overline{\text{RD}}$ , ALE,

**FIGURE 9.6** Memory-read cycle timing



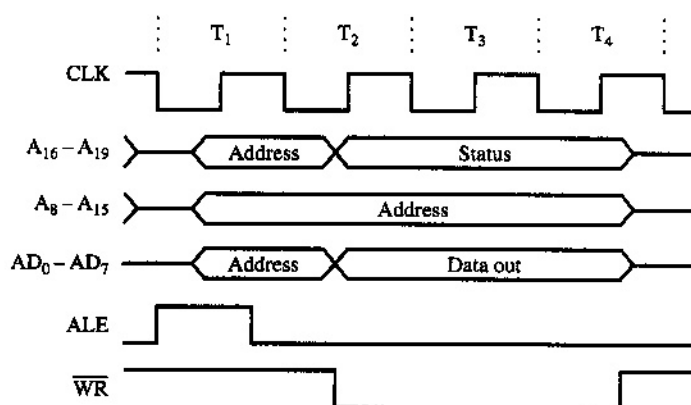


FIGURE 9.7 Memory-write cycle timing

and the address lines in such a way that the data is placed onto the data bus only when  $\text{IO}/\overline{\text{M}}$  and  $\overline{\text{RD}}$  are both low.

Figure 9.7 shows the same basic timing for a memory-write cycle. The most noticeable difference (aside from the use of  $\overline{\text{WR}}$  instead of  $\overline{\text{RD}}$ ) is the activity on the data bus. Unlike the read cycle, the data bus switches from address-out information to data-out information and keeps a valid copy of the output data on the bus for the remainder of the cycle. This should eliminate any setup times required by the memory chips.

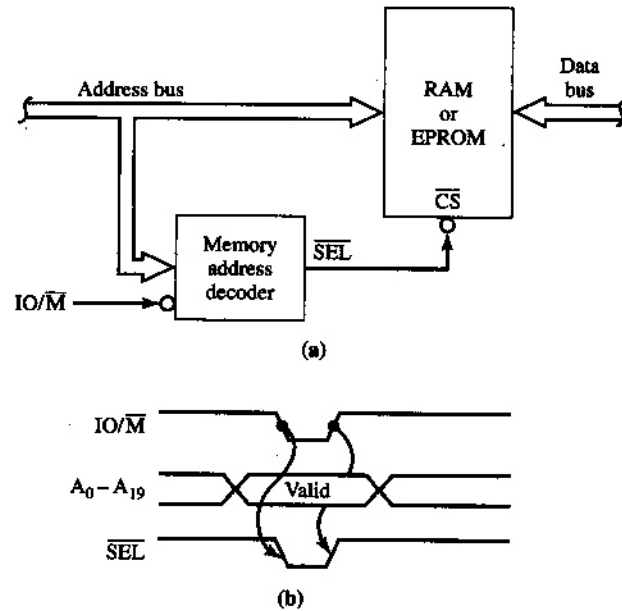
In the next section we will see how a memory address decoder uses the address bus and the read/write signals to enable RAM and EPROM memories.

## 9.5 DESIGNING A MEMORY ADDRESS DECODER

The sole function of a **memory address decoder** is to monitor the state of the address bus and determine when the memory chips should be enabled. But what is meant by *memory chips*? These are the actual RAMs or ROMs the designer wants to use in the computer. So, before the design begins, it must be decided how much memory is needed. If 8KB of EPROM is enough, then the designer knows that thirteen address lines are needed to address a specific location inside the EPROM (because 2 raised to the 13th power is 8,192). How many address lines are needed to select a specific location in a 32K memory? The answer is fifteen, because 2 to the 15th is 32,768! The first step in designing a memory address decoder is determining how many address lines are needed just for the memory device itself. Any address lines remaining are used in the address decoder.

Figure 9.8(a) shows a block diagram of a memory address decoder connected to a memory chip. Figure 9.8(b) shows a simplified timing diagram representing the activity on the address bus and the  $\text{IO}/\overline{\text{M}}$  output. The memory address decoder waits for a particular pattern on the address lines and a low on  $\text{IO}/\overline{\text{M}}$  before making  $\overline{\text{SEL}}$  low. When these conditions are satisfied, the low on  $\overline{\text{SEL}}$  causes the  $\overline{\text{CS}}$  (chip select) input on the memory chip to go low, which enables its internal circuitry, thus connecting the RAM or EPROM to the processor's data bus. When the address bus contains an address different from the one the address decoder expects to see, or if  $\text{IO}/\overline{\text{M}}$  is high, the output of the decoder will remain high, disabling the memory chip and causing its internal buffers to tri-state themselves. Thus, the RAM or EPROM is effectively disconnected from the data bus.

**FIGURE 9.8** Simple memory address decoder: (a) block diagram; (b) timing



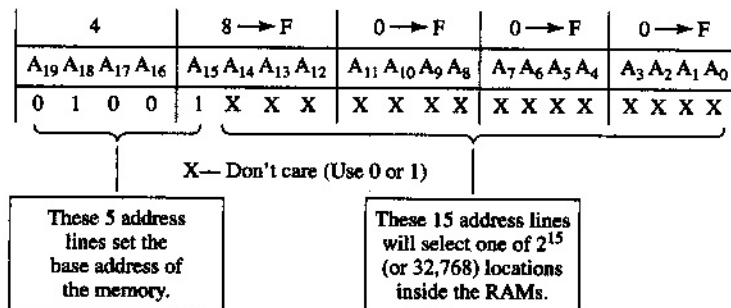
The challenge presented to us, the designers of the memory address decoder, is to chip-enable the memory device at the correct time. The following example illustrates the steps involved in the design of a memory address decoder.

### ■ EXAMPLE 9.1

A circuit containing 32KB of RAM is to be interfaced to an 8088-based system, so that the first address of the RAM (also called the **base address**) is at 48000H. What is the entire range of RAM addresses? How is the address bus used to enable the RAMs? What address lines should be used?

**Solution:** Figure 9.9 shows how the memory lines are assigned.

Because we are using a 32KB device, we need fifteen address lines to select one of 32K possible addresses. We always use the lowest numbered address lines first (the least significant ones). We start with  $A_0$  and use the next fourteen just for the RAM. This means that  $A_0$  through  $A_{14}$  go directly to the RAM circuitry, where they will be used to select a location inside the RAM. The remaining five address lines,  $A_{15}$  through  $A_{19}$ , are used to select the specific 32KB bank located at address 48000.

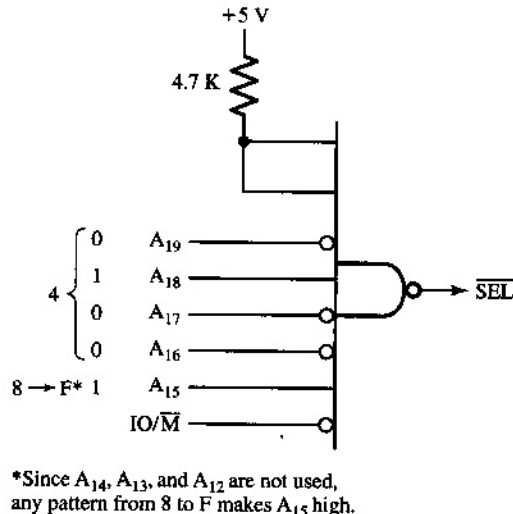


**FIGURE 9.9** Memory address range decoding

To determine the entire range of addresses, first make all the don't cares (the Xs in Figure 9.9) zeros. That gives address 48000, the first address in the range of addresses. Next, make all don't cares high to generate the last address, which becomes 4FFFF.

Note that  $A_{18}$  and  $A_{15}$  are high when the 32KB RAM bank is being accessed, while the other three upper address bits are low. This particular pattern of 1s and 0s is one of 32 possible binary combinations that may occur on the upper five address bits.

**FIGURE 9.10** Memory address decoder for 48000 to 4FFFF range



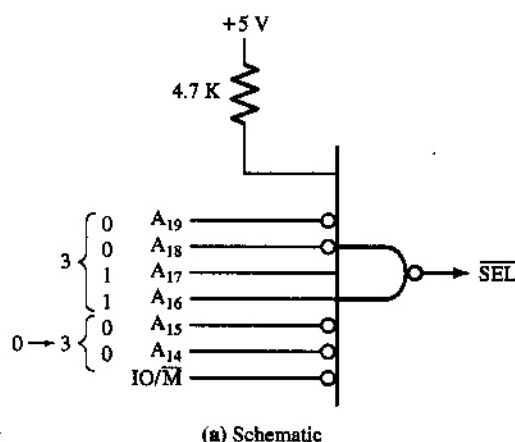
We need to detect a *single* pattern, so that the RAM circuit responds only to the address range 48000H to 4FFFFH. The circuit of Figure 9.10 is one way to do this. The output of the 8-input NAND gate is the output of the memory address decoder, which in turn gets connected to the chip-enable inputs of the 32KB RAM bank. The only time the output of the NAND gate will go low is when *all* of its inputs are high. Because some of the upper address bits are low when the desired memory range is present on the address bus, they must be inverted before they reach the NAND gate. Even though  $A_{19}$ ,  $A_{17}$ , and  $A_{16}$  are low, the NAND gate receives three 1s from them, via the inverters. Because  $A_{18}$  and  $A_{15}$  are already high, there are now five 1s present on the input of the NAND gate. When  $IO/\overline{M}$  goes low, indicating a valid memory address, the last required logic 1 is presented to the NAND gate (via another inverter), and its output goes low, enabling the 32KB RAM bank. ■

In general, a memory address decoder is used to reduce many inputs to a single output. The inputs are address lines and control signals. The single output is usually an enable signal sent to the memory section. Various TTL gates are used depending on the addressing requirements. The following examples present only a few of the hundreds of ways we can design memory address decoders to suit our needs.

### ■ EXAMPLE 9.2

A 16KB EPROM section, with a starting address of 30000, is to be added to an existing memory system. The following circuitry will properly decode the entire address range, 30000 to 33FFF. The 8-input NAND gate in Figure 9.11 is used to detect the 3 pattern on the upper four address bits, and the 0s on  $A_{15}$  and  $A_{14}$ . Six address lines are used in this

**FIGURE 9.11** Memory address decoder for 30000 to 33FFF range



| 3                                                               | 0 → 3                                                           | 0 → F                                                         | 0 → F                                                       | 0 → F                                                       |
|-----------------------------------------------------------------|-----------------------------------------------------------------|---------------------------------------------------------------|-------------------------------------------------------------|-------------------------------------------------------------|
| A <sub>19</sub> A <sub>18</sub> A <sub>17</sub> A <sub>16</sub> | A <sub>15</sub> A <sub>14</sub> A <sub>13</sub> A <sub>12</sub> | A <sub>11</sub> A <sub>10</sub> A <sub>9</sub> A <sub>8</sub> | A <sub>7</sub> A <sub>6</sub> A <sub>5</sub> A <sub>4</sub> | A <sub>3</sub> A <sub>2</sub> A <sub>1</sub> A <sub>0</sub> |
| 0 0 1 1                                                         | 0 0 X X                                                         | X X X X                                                       | X X X X                                                     | X X X X                                                     |

(b) Memory address range decoding

decoder, because the other fourteen are needed to address one of 16K possible byte locations in the EPROM. The EPROM is directly addressed by A<sub>0</sub> through A<sub>13</sub>. ■

Let us pause for a moment to consider a few points. In Figure 9.5, we saw how  $\overline{IO/\overline{M}}$  was combined with  $\overline{RD}$  and  $\overline{WR}$  to create the  $\overline{MEMRD}$  and  $\overline{MEMWR}$  signals used in a minmode system. In this case, we do not have to use  $\overline{IO/\overline{M}}$  again in the address decoder, because this would lead to redundancy in our design. We can, however, eliminate the use of the OR gates to decode the  $\overline{MEMRD}$  and  $\overline{MEMWR}$  signals and use  $\overline{RD}$  and  $\overline{WR}$  directly, but in this case we *must* use  $\overline{IO/\overline{M}}$  in the address decoder. The point is this: in a minmode system,  $\overline{IO/\overline{M}}$  must be used somewhere in the memory system. Without it, the memory cannot differentiate between memory addresses and I/O-port addresses.

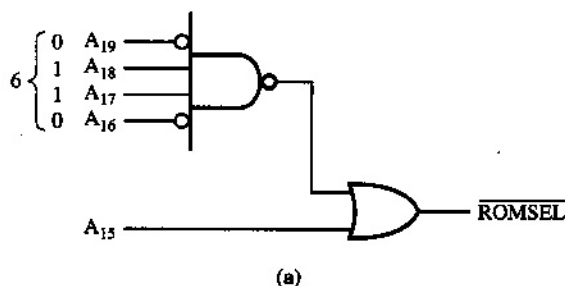
In a maxmode system, the  $\overline{IO/\overline{M}}$  signal is not even generated, so it is not possible to include it in our designs. Instead, we use the 8-input NAND gate (or some other combination of gates) to decode only the address we wish to recognize. The  $\overline{MRDC}$  and  $\overline{MWTC}$  signals generated by the 8288 bus controller will be connected directly to the memory device being accessed.

### ■ EXAMPLE 9.3

Two 32KB memories, an EPROM with a starting address of 60000 and a RAM with a starting address of 70000, are needed for a new maxmode memory system. Figure 9.12(a) shows how the EPROM is enabled, and Figure 9.12(b) shows how the RAM is enabled. In this design, it is only necessary to use a 4-input NAND gate to do the decoding of the “6” or “7” part of the address range. ■

Experienced digital designers can detect binary patterns, and the reward in finding a pattern is generally a reduction in the digital circuitry needed to implement a desired function. Did you notice that the address ranges for the RAM and EPROM in the previous example are

**FIGURE 9.12** Memory address decoders for two different ranges: (a) EPROM bank at 60000; (b) RAM bank at 70000; (c) Memory address range decoding





### EXAMPLE 9.5

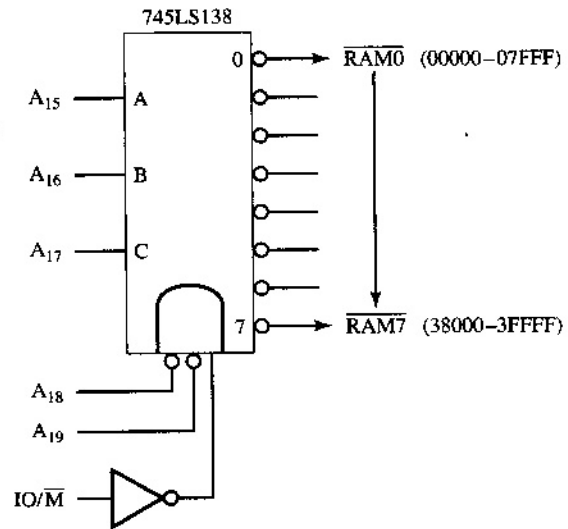
A 256KB RAM memory is composed of eight 32KB RAMs. The address ranges for the RAMs are as follows:

- |                   |                   |
|-------------------|-------------------|
| 1. 00000 to 07FFF | 5. 20000 to 27FFF |
| 2. 08000 to 0FFFF | 6. 28000 to 2FFFF |
| 3. 10000 to 17FFF | 7. 30000 to 37FFF |
| 4. 18000 to 1FFFF | 8. 38000 to 3FFFF |

How might all eight RAMs be selectively enabled by one device?

**Solution:** Our first thought may be to use eight individual memory address decoders, one for each address range and 32KB RAM. But this would be an unnecessary waste of circuitry. If we instead look for a pattern, we see that address lines  $A_{19}$  and  $A_{18}$  are always low in the memory range 00000 to 3FFFF. In addition to this important piece of information, each RAM requires fifteen address lines,  $A_0$  through  $A_{14}$ , to select one of 32KB locations within the RAM. This leaves us with address lines  $A_{15}$ ,  $A_{16}$ , and  $A_{17}$  actually indicating a

**FIGURE 9.14** Multibank address decoder



(a) Schematic

| $A_{19}$ | $A_{18}$ | $A_{17}$ | $A_{16}$ | $A_{15}$ | $A_{14}$ | $A_{13}$ | $A_{12}$ | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | Range          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------------|
| 0        | 0        | 0        | 0        | 0        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 00000 to 07FFF |
| 0        | 0        | 0        | 0        | 1        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 08000 to 0FFFF |
| 0        | 0        | 0        | 1        | 0        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 10000 to 17FFF |
| 0        | 0        | 0        | 1        | 1        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 18000 to 1FFFF |
| 0        | 0        | 1        | 0        | 0        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 20000 to 27FFF |
| 0        | 0        | 1        | 0        | 1        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 28000 to 2FFFF |
| 0        | 0        | 1        | 1        | 0        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 30000 to 37FFF |
| 0        | 0        | 1        | 1        | 1        | X        | X        | X        | X        | X        | X     | X     | X     | X     | X     | X     | X     | X     | X     | X     | 38000 to 3FFFF |

(b) Memory address range decoding

specific 32KB memory range. When these three address lines are all low, address range 00000 to 07FFF is selected. When  $A_{15}$  is high, and  $A_{16}$  and  $A_{17}$  low, address range 08000 to 0FFFF is selected. The last range, 38000 to 3FFFF, is selected when  $A_{15}$ ,  $A_{16}$ , and  $A_{17}$  are all high. What we need then is a circuit that can decode these eight possible conditions by using only the three address lines. Figure 9.14 shows the required circuitry.

In this circuit, a 74LS138 three- to eight-line decoder is used to decode the different memory ranges. The 74LS138 has three select inputs and three control inputs. The select inputs are connected to address lines  $A_{15}$ ,  $A_{16}$ , and  $A_{17}$ . The 3-bit binary number present on the select inputs will pull the selected output of the 74LS138 low (assuming that the 74LS138 is enabled), thus activating a specific RAM bank. To enable the 74LS138, two lows and a high must be placed on its control inputs. The two lows are generated by  $A_{19}$  and  $A_{18}$ . The IO/M signal is inverted to generate the last control input.

By using special integrated circuits like the 74LS138 and a simple pattern recognition technique, we are able to greatly simplify the hardware required to generate all of our memory enable signals. ■

The last four examples have shown how we can decode a specific range of memory addresses using the full address bus of the 8088. In the next section, we will see how to further simplify our decoder, by using a technique called partial-address decoding.

## 9.6 PARTIAL-ADDRESS DECODING

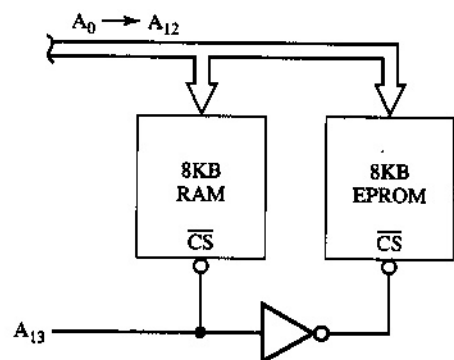
Although the 8088 is capable of addressing over 1 million bytes of memory, some applications may require much smaller memories. A good example might be an educational 8088 single-board computer, much like the one presented in Chapter 12, using only 8K words of EPROM and 8K words of RAM. This type of system needs only fourteen address lines. The first thirteen,  $A_0$  through  $A_{12}$ , go directly to the EPROM and RAM, and the last address line,  $A_{13}$ , is used to select either the EPROM or the RAM. Figure 9.15 details this example system.

In this figure,  $A_{13}$  is connected directly to the  $\overline{CS}$  input of the RAM and is *inverted* before it gets to the  $\overline{CS}$  input of the EPROM. So, whenever  $A_{13}$  is low, the RAM is enabled, and whenever  $A_{13}$  is high, the EPROM is enabled. We use a single inverter to do all the decoding in our memory section! This is what is meant by **partial-address decoding**.

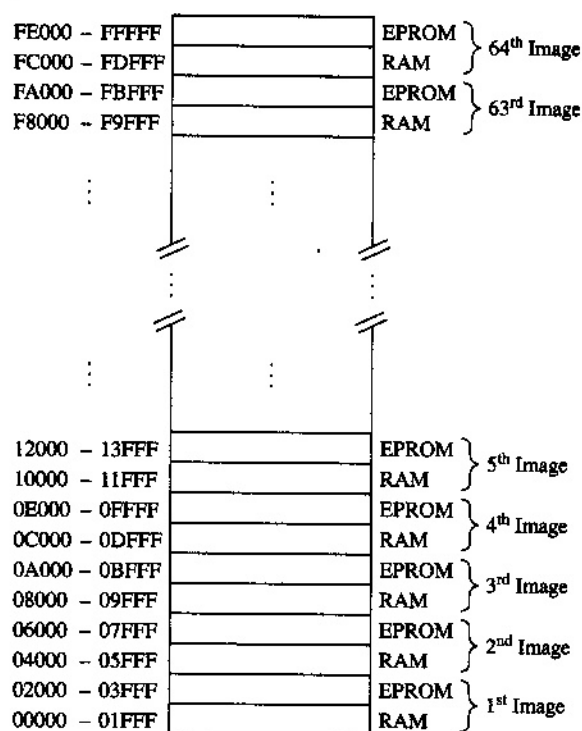
But what about the other address lines,  $A_{14}$  through  $A_{19}$ ? They are ignored, and here is why: when the 8088 is powered-up, a reset causes the processor to look first at memory location FFFF0 to get its initial instruction. We had better make sure good data is in that location at power-up. If we use an 8KB EPROM at FE000, we can be ensured that the correct information will be present.

Going back to Figure 9.15, it is clear that the EPROM will be enabled at power-on, because  $A_{13}$  will be high when the processor tries to access location FFFF0.

But we still do not know why the other six upper address lines,  $A_{14}$  through  $A_{19}$  in this case, can be ignored. The answer lies in Figure 9.15. Do you see any address lines other than  $A_{13}$  being used to enable or disable the EPROM or RAM memories? No! Because we ignore these address lines, it does not matter if they are high or low. We can read from memory locations FFFF0, 3BFF0, 07FF0, or C3FF0 and get the same data each time. The upper address bits have no effect on our memory circuitry, because we are using only the lower fourteen address lines. This leads to multiple copies of the RAM and EPROM data within the processors addressing space. In this case there are sixty-four copies.

**FIGURE 9.15** Partial-address decoding for RAM/EPROM

(a) Schematic



(b) Memory map

Partial-address decoding gives us a way to get the job done with a minimum of hardware. Because fewer address lines have to be decoded, less hardware is needed. This is its greatest advantage. A major disadvantage is that future expansion of memory is difficult and usually requires a redesign of the memory address decoder. This may turn out to be a difficult, or even impossible, job. The difficulty lies in having to add hardware to the system. If a system manufactured by one company has been distributed to a number of users, making changes to all systems becomes a challenge. Furthermore, individuals wanting to make changes themselves may mistakenly place a new memory device into a partially decoded area. This will unfortunately result in two memories being accessed at the same time, probably resulting in invalid data during reads.

As long as these dangers and limitations are understood, partial-address decoding is a suitable compromise and acceptable in small systems.

Two more examples are presented to further show the simplicity of partial-address decoding.

### EXAMPLE 9.6

A 16KB block of memory, composed of two 8KB EPROMs, is to have a starting address of 4000H. What is the address range for each EPROM?

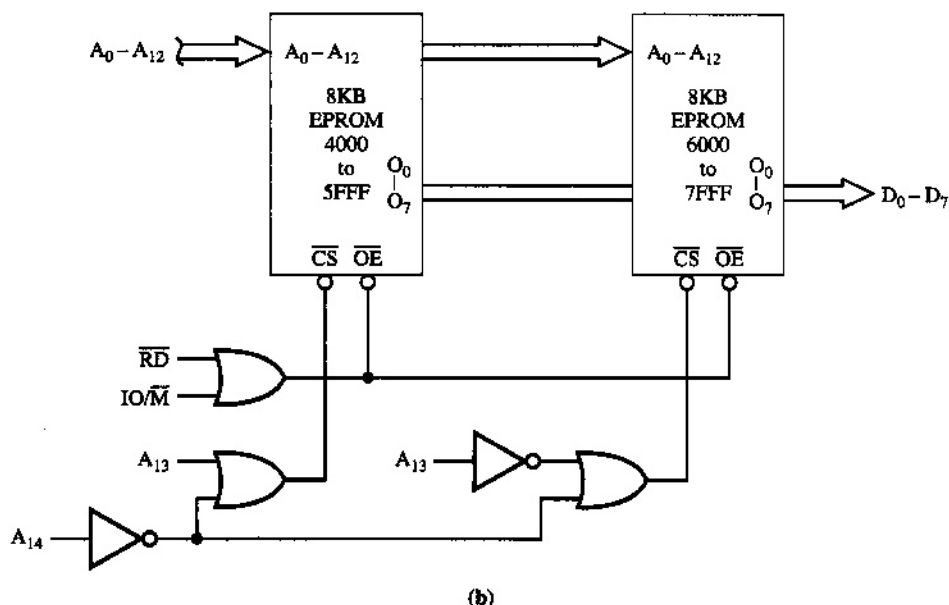
What circuitry is needed to implement a partial-address decoder for a minmode system?

**Solution:** Figure 9.16(a) shows the address decoding table for the 16KB block of storage. The base address of 4000H is written down in binary, with the lower 13 bits associated with  $A_0$  through  $A_{12}$ , the address lines needed to select locations within each 8KB EPROM. Note that  $A_{14}$  is high and  $A_{13}$  is low for all possible binary patterns of 0s and 1s on  $A_0$  through  $A_{12}$ . This sets the address range for the first EPROM, which is 4000 to 5FFF. An

**FIGURE 9.16** 16KB EPROM storage using partial addressing: (a) address decoding table; (b) EPROM circuitry

| $A_{19}$ | $A_{18}$ | $A_{17}$ | $A_{16}$ | $A_{15}$ | $A_{14}$ | $A_{13}$ | $A_{12}$ | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0        | 0        | 0        | 0        | 0        | 1        | 0        | 0        | 0        | 0        | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0        | 0        | 0        | 0        | 0        | 1        | 0        | 1        | 1        | 1        | 1     | 1     | 1     | 1     | 1     | 1     | 1     | 1     | 1     | 1     |
| 0        | 0        | 0        | 0        | 0        | 1        | 1        | 0        | 0        | 0        | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| 0        | 0        | 0        | 0        | 0        | 1        | 1        | 1        | 1        | 1        | 1     | 1     | 1     | 1     | 1     | 1     | 1     | 1     | 1     | 1     |

(a)



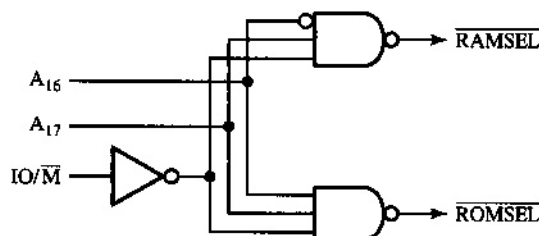
OR gate is used to recognize the 1 0 pattern on  $A_{14}$  and  $A_{13}$ , as shown in Figure 9.16(b). Continuing with this technique, we see that  $A_{14}$  and  $A_{13}$  are both high when the second EPROM is being accessed. This translates into the address range 6000 to 7FFF, or a total address range of 4000 to 7FFF. Another OR gate is used to decode the 1 1 pattern on  $A_{14}$  and  $A_{13}$ . Because this memory is used in a minmode system,  $\overline{IO/\overline{M}}$  and  $\overline{RD}$  are combined with a third OR gate and used to control the output-enable input of both EPROMs. ■

You may want to practice by redesigning this circuit with two-input NAND gates.

### EXAMPLE 9.7

A 32KB EPROM needs a starting address of 30000, and a 32KB RAM needs a starting address of 20000. The circuitry in Figure 9.17 shows how these addresses are partially decoded. In this example, three-input NAND gates are used to do the decoding. All three inputs must be high for the output to go low (and enable the memories).  $\text{IO}/\overline{\text{M}}$  is inverted, so that it presents a 1 when low.  $A_{17}$  is connected directly, because it is high in both the RAM and EPROM address ranges. Only  $A_{16}$  changes. It is low for the RAM range and high for the EPROM range. Address lines  $A_0$  through  $A_{14}$  are used by the memories themselves.

**FIGURE 9.17** Partial-address decoder for 32KB EPROM at 30000, and 32KB RAM at 20000



Can you determine the range of addresses for each memory? ■

## 9.7 GENERATING WAIT STATES

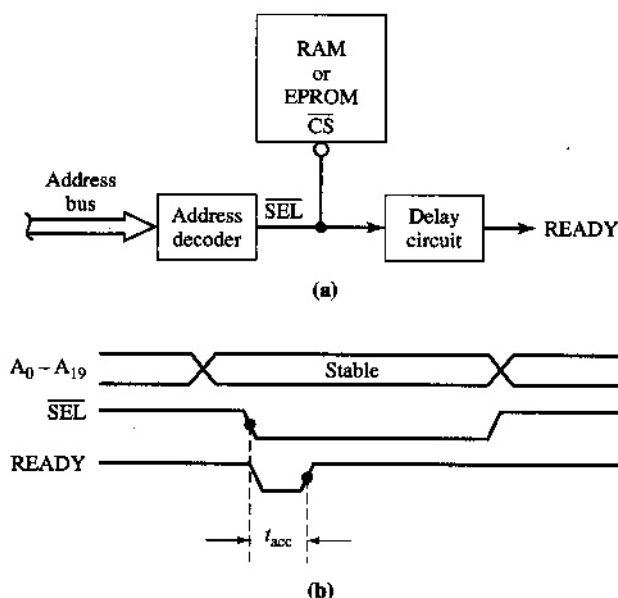
READY is a signal that tells the 8088 CPU that data may be read from or written into memory. We have ignored this signal so far, so that we could develop an understanding of how memory address decoders work. The first function of the memory address decoder is to monitor the address bus and activate the RAMs or EPROMs when a specific address, or range of addresses, is seen. The second function of the decoder is to tell the CPU to *wait* until the memories have been given enough time to become completely active. A typical RAM might require 200 ns to become active after it gets enabled. This is due to the time required by the internal RAM circuitry to correctly decode the supplied address and turn on its internal buffers. If the decoder did not tell the CPU to wait for 200 ns while this was happening, problems such as data loss might arise. The READY signal gives us a way to slow down the 8088 so that it can use slow memories. Typically, the period of the CPU clock is much smaller than the memory access time, requiring multiple clock cycles of waiting time for memory accesses.

In Figure 9.18(a) we see that the output of the address decoder is connected to the memory and to a delay circuit. The delay circuit is used to extend the bus cycle for a time equal to the access time of the memories. The 8088 samples READY during state  $T_2$  of each bus cycle and will not proceed into  $T_3$  until READY is at a high level. Pulling READY low with the delay circuit inserts a wait state into the bus cycle. The wait state time is equal to  $t_{acc}$ , the access time of the memory. This is shown in Figure 9.18(b).

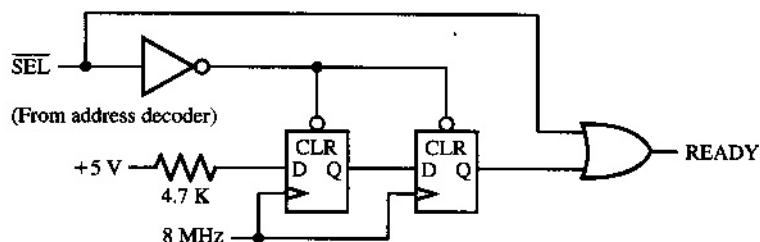
How do we delay the generation of READY? A very simple solution is to use a shift register. Consider the 2-bit shift register of Figure 9.19. It will take two clock pulses for information at the first D input to get to the second Q output. So, if  $\overline{\text{SEL}}$  goes low, we expect READY to go low for two clock pulses. If an 8-MHz clock is used, the flip-flops are clocked every 125 ns, which results in a wait state of 250 ns!

The delay time needed depends on the type of memory being used, the clock frequency, and the size (in stages) of the shift register. A one-shot (mono-stable multivibrator)

**FIGURE 9.18** READY operation during memory access: (a) block diagram of delay circuit; (b) associated timing



**FIGURE 9.19** Using a 2-bit shift register, to generate a wait state



could be used as well but would not be as stable as the digital circuit due to the nature of the resistor/capacitor network needed. We finish this section with an example of a delay circuit. In our next section, we will see a complete schematic of an 8KB RAM/EPROM memory.

### EXAMPLE 9.8

One type of delay circuit is composed of three D-type flip-flops connected as a 3-bit shift register driven by a 4-MHz clock. Compute the length of the delay generated by this circuit.

**Solution:** The length of delay is three times the period of the clock. A 4-MHz clock has a period of 250 ns; therefore, the delay time is 750 ns.

Can the delay time in this circuit be doubled (to 1.5  $\mu$ s) by adding only one more flip-flop? The answer is yes and is left for you to prove as a homework problem. ■

## 9.8 A COMPLETE RAM/EPROM MEMORY

Now that we have covered all the required basics, a complete memory design is presented. Figure 9.20(a) shows the required hardware necessary for 8KB of EPROM (located at base address FE000) and 8KB of RAM (located at 00000).

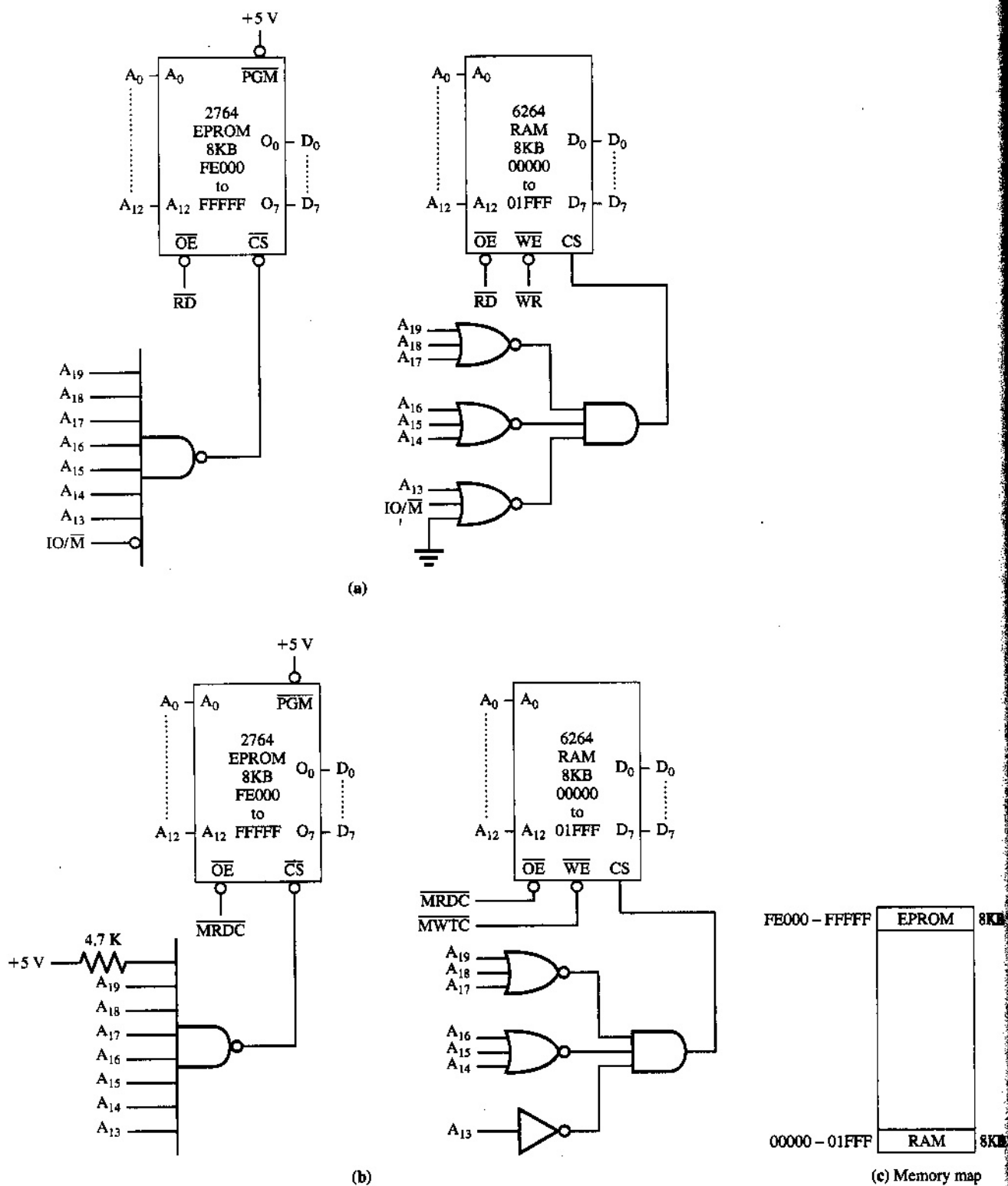


FIGURE 9.20 Complete RAM/EPROM memory: (a) minmode system; (b) maxmode system

The control signals are associated with a minimum mode system. The addresses for each memory device are fully decoded. The 8-input NAND gate is used to enable the EPROM, and three 3-input NOR gates and a 3-input AND gate are used to enable the RAM. Different logic is required in each decoder, because the EPROM address requires recognition of seven 1s and the RAM decoder must recognize seven 0s. The 2764 is an 8KB EPROM, with an internal address range of 0000 to 1FFF, making its system address range FE000 to FFFFF. The 6264 is an 8KB static RAM with a system address range from 00000 to 01FFF. The memories were placed into the 8088's memory space in such a way that the EPROM is enabled upon reset, and the RAM is available for interrupt vector, program, and data storage.

Figure 9.20(b) shows an almost identical memory system, with a few changes made so that it can be placed into a maximum mode system.  $\overline{RD}$  and  $\overline{WR}$  now become  $\overline{MRDC}$  and  $\overline{MWTC}$  (generated by the 8288 bus controller).  $IO/\overline{M}$  disappears, allowing us to eliminate one of the NOR gates in the RAM section and requiring the addition of an inverter (the one that was previously used for  $IO/\overline{M}$  in the EPROM section).

While 8KB of RAM is enough for small educational systems, other systems may require much more memory. In the next section, we will see how dynamic RAM can be interfaced to the 8088.

## 9.9 DYNAMIC RAM INTERFACING

### What Is Dynamic RAM?

**Dynamic RAM** is a special type of RAM memory that is currently the most popular form of memory used in large memory systems for microprocessors. It is important to discuss a few of the specific differences between static RAMs and dynamic RAMs. Static RAMs use digital flip-flops to store the required binary information, whereas dynamic RAMs use MOS capacitors. Because of the capacitive nature of the storage element, dynamic RAMs require less space per chip, per bit, and thus have larger densities.

In addition, static RAMs draw more power per bit. Dynamic RAMs employ MOS capacitors that retain their charges (stored information) for short periods of time, whereas static RAMs must saturate transistors within the flip-flop to retain the stored binary information, and saturated transistors dissipate maximum power.

A disadvantage of the dynamic RAM stems from the usage of the MOS capacitor as the storage element. Left alone, the capacitor will eventually discharge, thus losing the stored binary information. For this reason, the dynamic RAM must be constantly **refreshed** to avoid data loss. During a refresh operation, all of the capacitors within the dynamic RAM (called DRAM from now on) are recharged.

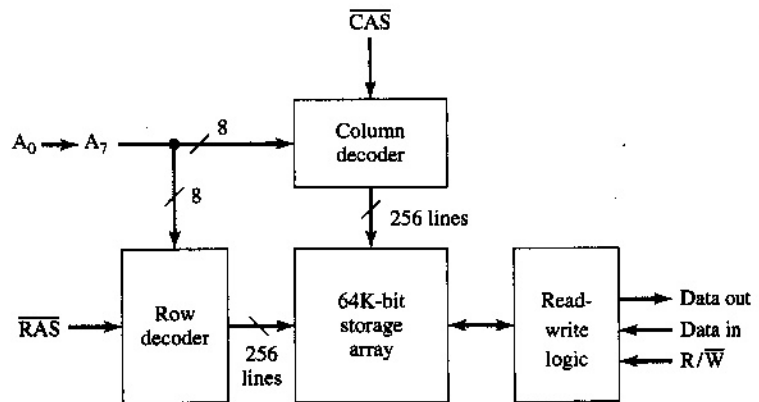
This leads to a second disadvantage. The refresh operation takes time to complete, and the DRAM is unavailable for use by the processor during this time.

Older DRAMs required that all storage elements inside the chip be refreshed every 2 ms. Newer DRAMs have an extended 4-ms refresh time, but the overall refresh operation ties up an average of 3 percent of the total available DRAM time, which implies that the CPU has access to the DRAM only 97 percent of the time. Because static RAMs require no refresh, they are available to the CPU 100 percent of the time, a slight improvement over DRAMs.

In summary, we have static RAMs that are fast, require no refresh, and have low bit densities. DRAMs are slower and require extra logic for refresh and other timing controls but are cheaper, consume less power, and have very large bit densities.



**FIGURE 9.21** Internal block diagram of a 64K-bit dynamic RAM



### Accessing Dynamic RAM

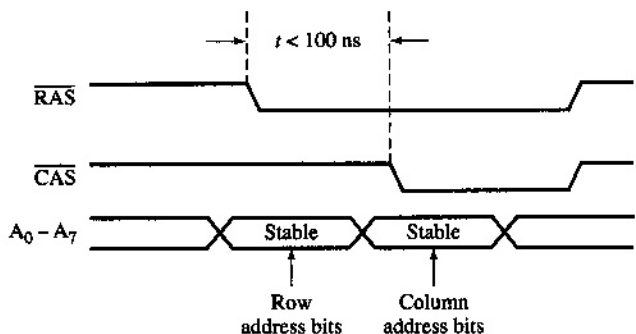
A major difference in the usage of DRAMs lies in the way in which the DRAM is addressed. A 64K-bit DRAM requires 16 address bits to select one of 65,536 possible bit locations, but its circuitry contains only eight address lines. A study of Figure 9.21 will show how these eight address lines are expanded into sixteen address lines with the help of two additional control lines:  $\overline{\text{RAS}}$  and  $\overline{\text{CAS}}$ .

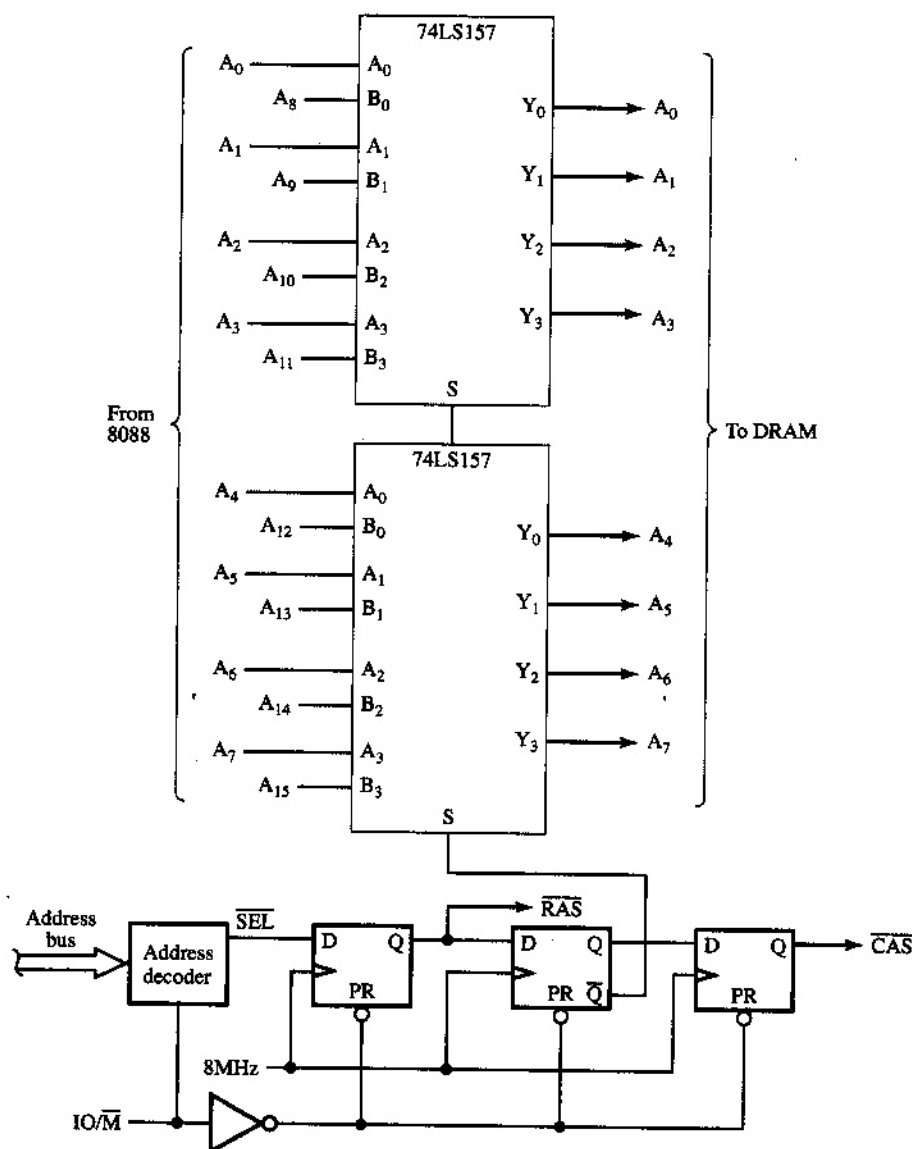
The eight address lines are presented to row and column address buffers and latched accordingly by the application of the  $\overline{\text{RAS}}$  and  $\overline{\text{CAS}}$  signals. To load a 16-bit address into the DRAM, 8 bits of the address are first latched by pulling  $\overline{\text{RAS}}$  low. Then the other 8 address bits are presented to  $A_0$  through  $A_7$ , and  $\overline{\text{CAS}}$  is pulled low. By adding just one more address line to the DRAM, the addressing capability is increased by a factor of 4, because one extra address line signifies an extra row and column address bit. This explains why DRAMs tend to quadruple in size with each new release.

The actual method for addressing the DRAM is presented in Figure 9.22. First, the 8 row address bits are applied to  $A_0$  through  $A_7$ , and  $\overline{\text{RAS}}$  is pulled low. Then  $A_0$  through  $A_7$  receive column address information, and  $\overline{\text{CAS}}$  is pulled low. After a short delay, the circuitry inside the DRAM will have decoded the full 16-bit address, and reading or writing may commence.

The row address strobe and column address strobe signals must be generated within 100 ns of each other to avoid data loss. The specific timing requirements for other DRAM chips depend on the manufacturer.

**FIGURE 9.22** DRAM cycle timing





**FIGURE 9.23** Address bus selector for DRAM

External logic is needed to generate the  $\overline{RAS}$  and  $\overline{CAS}$  signals, and also to take care of presenting the right address bits to the DRAMs. The circuit of Figure 9.23 shows an example of the required logic.

The operation of this circuit is as follows. The address decoder monitors the address bus for an address in the desired DRAM range and outputs a logic 0 when it sees one. Normally the three Q outputs of the shift register are all high. The first clock pulse will shift the logic 0 from the address decoder to the output of the first flip-flop, causing  $\overline{RAS}$  to go low. Because the output of the second flip-flop is still high, the 74LS157s (quad 2-line to 1-line multiplexers) are told to pass processor address lines A<sub>0</sub> through A<sub>7</sub>. This is how we load the ROW address bits into the DRAM.

The second clock pulse will shift the logic 0 to the second Q output (the first is still low also), which causes the 74LS157s to select the processor address lines  $A_8$  through  $A_{15}$ . These address bits are recognized and latched by the DRAM when the third clock pulse occurs, because the logic 0 has been shifted to the third Q output, which causes  $\overline{CAS}$  to go low. The DRAM has been loaded with a full 16-bit address, and reading or writing may commence. At the end of the read or write cycle,  $IO/\overline{M}$  will go high, presetting all three flip-flops via the preset line, and the shift register reverts back to its original state.

This sequence will repeat every time the address decoder detects a valid address.

Figure 9.24 shows a complete DRAM addressing circuit with read-write logic. When the 74LS138 detects a valid memory address, one of its eight outputs will go low,

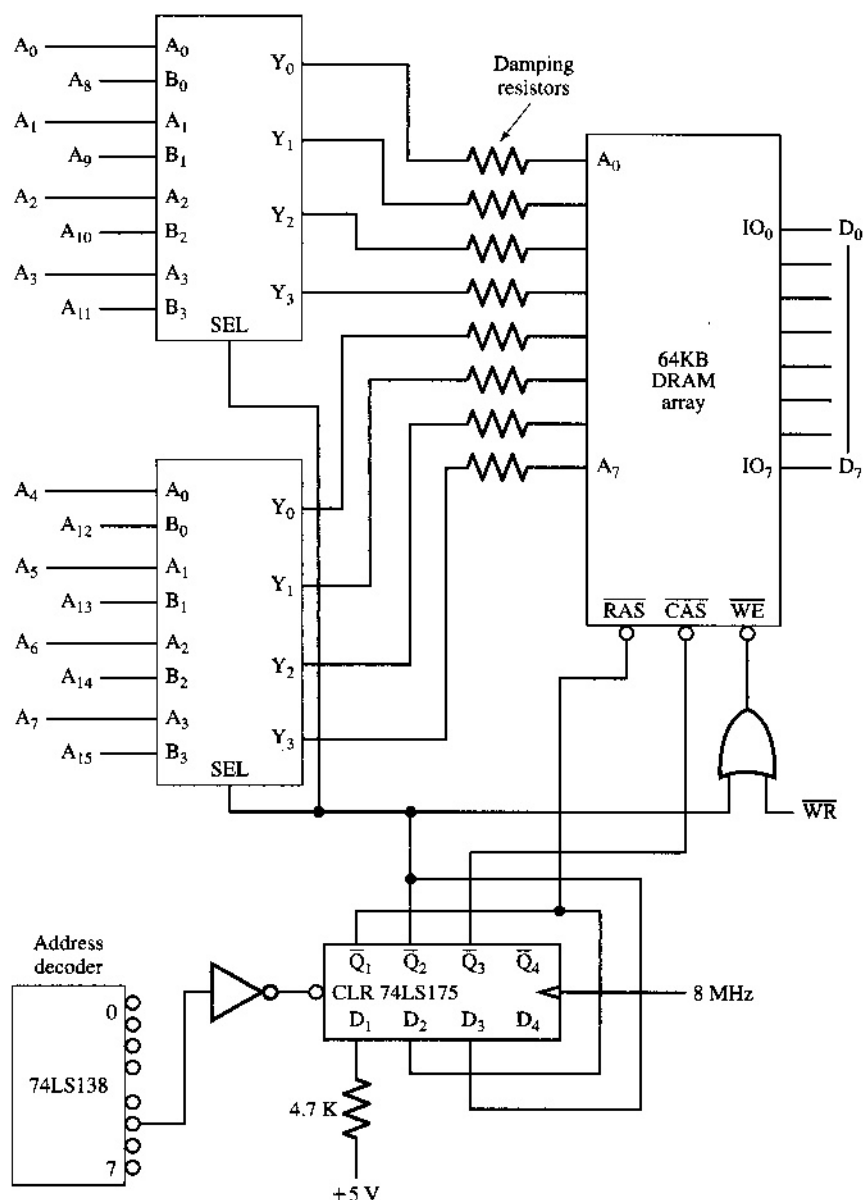


FIGURE 9.24 Complete DRAM addressing circuit

removing the 74LS175 quad D flip-flop from its forced-clear state. All four Q outputs are high at this time. As a logic 1 shifts through the 74LS175 (connected as a 4-bit shift register), the  $\overline{\text{RAS}}$ ,  $\overline{\text{WE}}$ , and  $\overline{\text{CAS}}$  signals will be generated. The resistors in the address and control lines are called **damping resistors** and are used to control the waveshape of the digital signals to the DRAMs. The damping resistors reduce ringing and other noise that would normally occur in a high-speed digital system. The only circuitry missing from Figure 9.24 is the required refresh logic, which we will study in the next section.

## Refreshing Dynamic RAM

Previously we learned that DRAMs need to be refreshed, or the MOS capacitors that retain the binary information will discharge and data will be lost. Older DRAMs required that all cells (storage elements) be refreshed within 2 ms. Although the process of reading or writing a DRAM cell is a form of refresh, it is possible that entire banks of DRAM remain inactive while the CPU addresses other memories or I/O devices, so a safe designer will include a refresh circuit in the new DRAM system.

Newer DRAMs (such as the MCM6664) contain a single control line called  $\overline{\text{REF}}$  that automatically refreshes the DRAM whenever it is pulled low. But how is refreshing accomplished? Let us look at the process that is used to refresh a DRAM and the circuitry needed to control the process.

DRAMs are internally designed as a grid of memory cells arranged as a matrix, with an equal number of rows and columns (hence the  $\overline{\text{RAS}}$  and  $\overline{\text{CAS}}$  control signals). A 4K bit DRAM would need twelve address lines: six for the row decoder and six for the column decoder. Each decoder would pick one row and column out of a possible 64. During a refresh operation, all 64 column-cells would be refreshed by the application of a single  $\overline{\text{RAS}}$  signal. This is called RAS-only refresh. To refresh all 4096 bits, it is necessary only to  $\overline{\text{RAS}}$  select all 64 rows. A larger DRAM, a 64K bit one for example, would require  $\overline{\text{RAS}}$  selecting more rows (256 in this case). The easiest way to ensure that all rows get selected during a refresh operation is to use a binary counter and connect the output of the counter to the DRAM address lines during a refresh. To ensure that the DRAMs get refreshed periodically, a timer is needed to generate a REFRESH signal. The REFRESH signal will suspend processor activity while the DRAM is refreshed. Figure 9.25 shows how a 555 timer can

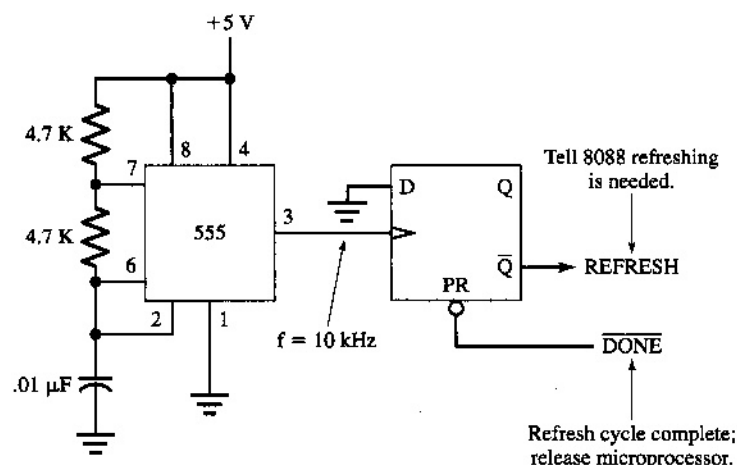


FIGURE 9.25 555 timer generates refresh signals every 100  $\mu\text{s}$

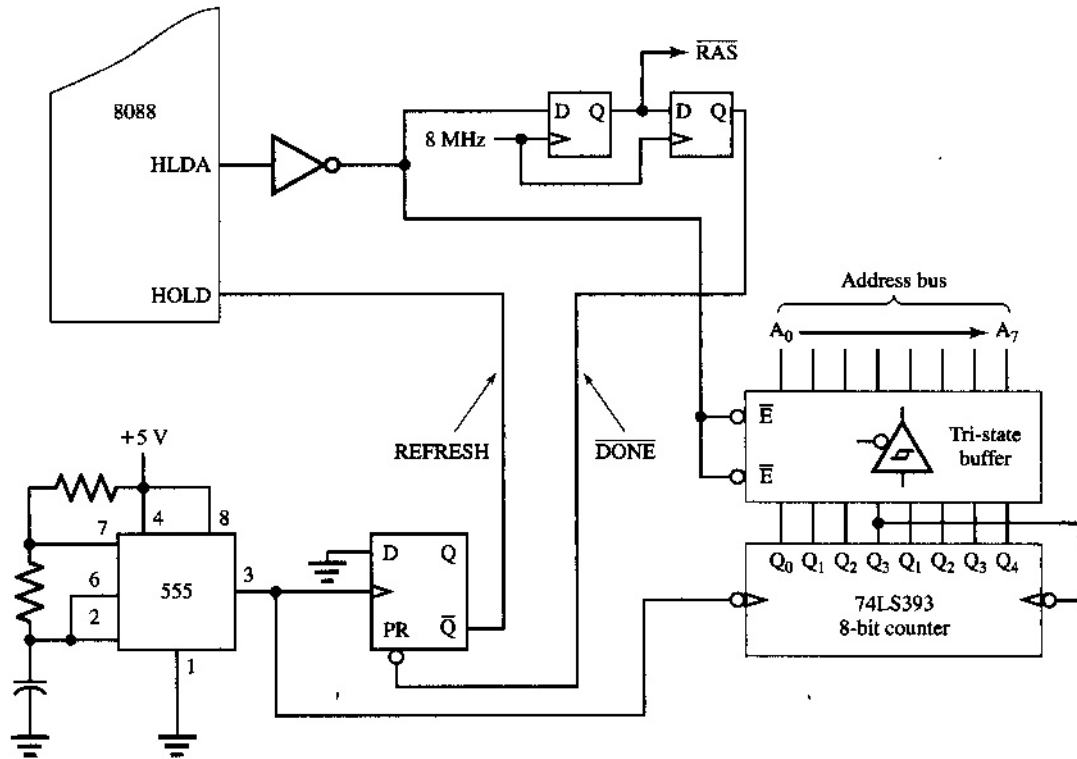


FIGURE 9.26 DRAM refresh generator

be used to generate a REFRESH signal every 100  $\mu$ s. The 555 timer clocks a D-type flip-flop, whose output is REFRESH. When the refresh cycle is completed,  $\overline{\text{DONE}}$  is used to preset the flip-flop and remove the REFRESH request, until the 555 times out again. Figure 9.26 shows how the refresh timer, together with the  $\overline{\text{RAS}}$  refresh circuitry, is used to refresh the DRAMs. When the 555 timer initiates a refresh cycle, REFRESH will go high, issuing a HOLD request to the 8088. The processor will respond by asserting HLDA, which allows a 0 to be clocked into the 2-bit shift register used to control  $\overline{\text{RAS}}$  and  $\overline{\text{DONE}}$ . When  $\overline{\text{RAS}}$  is active, bits  $A_0$  through  $A_7$  of the address bus will contain the 8-bit counter value (the current state of the 74LS393). When  $\overline{\text{DONE}}$  is active, the refresh flip-flop is preset, which removes the HOLD signal. This causes the processor to release HLDA, which in turn causes the 2-bit shift register to be loaded with 1s. At this point, the bus request is over, and the processor resumes execution. Because the 555 timer also clocks the 8-bit counter, a unique row address is generated each refresh cycle.

### A Dynamic RAM Controller

We may conclude that the circuitry required to address, control, and refresh DRAMs is both complicated and extensive (which may translate into *expensive*). There must be a simpler way.

There is!

Various companies make DRAM controller devices that take care of all refreshing and timing requirements needed by the dynamic RAMs. All circuitry is contained in a single

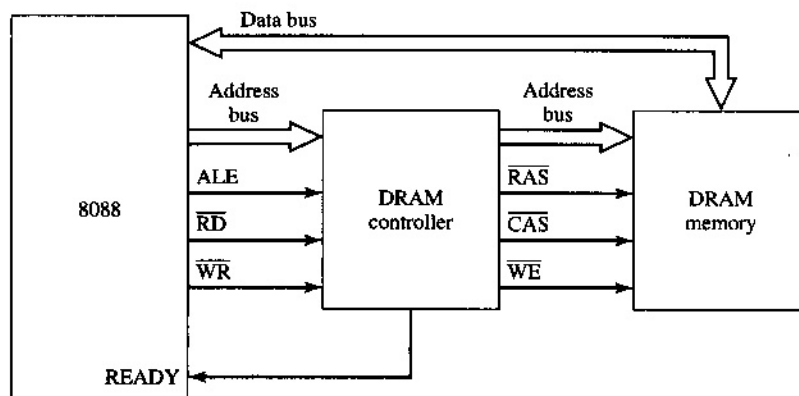


FIGURE 9.27 DRAM controller interfacing

package in most cases. The DRAM controller does its work independently of the processor. This means that the DRAM controller will issue a wait to the processor when the processor tries to access memory during a refresh cycle. Figure 9.27 shows how a DRAM controller is used in an 8088-based system. Using a dedicated DRAM controller minimizes the time required to design, debug, and eventually troubleshoot DRAM memory systems. It may also be more cost effective in the long run.

### Dynamic RAM Summary

Our study of DRAMs has shown that they are slow and require complicated circuitry to get them to work (unless a DRAM controller is used). On the other hand, DRAMs are cheaper, per bit, than static RAM, they consume less power, and have much larger bit densities. With the advance of the microcomputer into the word processing arena, where very large memories are needed to store and manipulate text, dynamic RAM becomes a very economical solution. Image processing, large informational databases, and virtually any large storage system make the use of dynamic RAMs an ideal choice. Furthermore, interfacing dynamic RAMs is made easier with the use of a DRAM controller.

---

## 9.10 DIRECT MEMORY ACCESS

**Direct memory access**, usually called DMA for short, is a process in which a device external to the CPU requests the CPU's buses (address bus, data bus, and control bus) for its own use to speed up memory-to-memory data transfers. Examples of external circuits that might wish to perform DMA are video pattern generators, which share video RAM with the CPU, and high-speed data transfer circuits such as those used in hard disks.

In general, a DMA process consists of a **slave device** requesting the use of the **master device's** buses. In a microprocessor-based system, the master is usually the CPU. Once the slave device has control of the bus, it can read or write to the system memory as necessary. When the slave device is finished, it releases control of the master's buses, and system operation returns to normal.

An example of why DMA is a useful technique can be illustrated in the following way. Suppose you wish to add a hard disk storage unit to your microcomputer. The hard disk boasts a data transfer rate of 5 million bytes per second. This comes to 1 byte transferred every 200 ns! Older microprocessors would be hard put to execute even *one* instruction in 200 ns, much less the multiple number of instructions that would be required to read the byte from the hard disk, place it in memory, increment a memory pointer, and then test for another byte to read. A DMA controller would be very handy in this example. It would merely take over the CPU's buses, write all the bytes into memory very quickly, and then return control to the CPU. Even today, DMA is still a valuable addition to a microprocessor-based system.

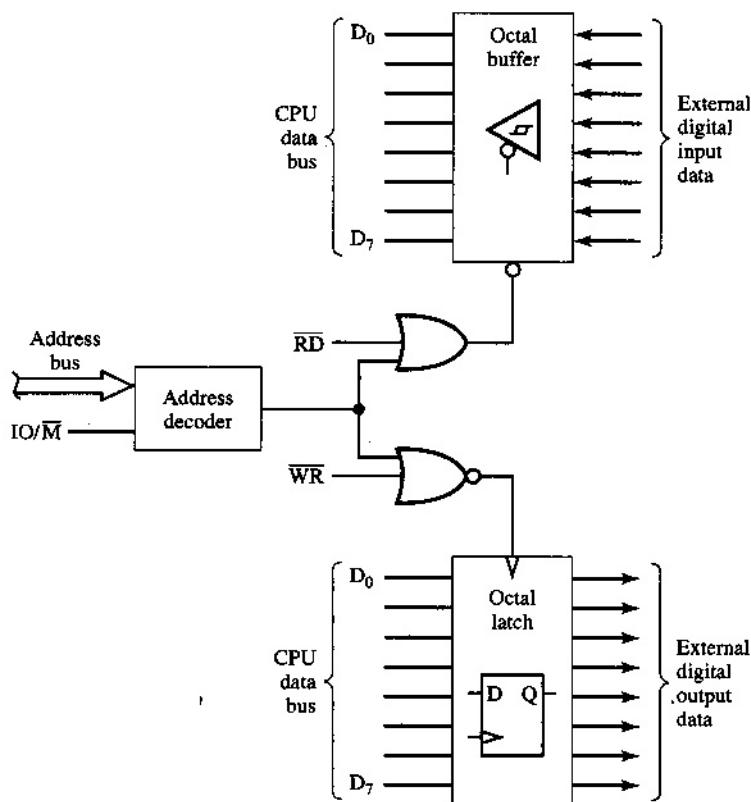
To perform DMA on the 8088, two signals may be used. They are  $\overline{RQ}/\overline{GT}_0$  (*request/grant*) and  $\overline{RQ}/\overline{GT}_1$ . Both of these signals are bidirectional; they are both inputs and outputs. A logic zero must be placed on either signal to request the processor's bus, with  $\overline{RQ}/\overline{GT}_0$  having priority over  $\overline{RQ}/\overline{GT}_1$ . The processor will acknowledge the takeover when the current bus cycle finishes and output a 0 on the appropriate signal line. The low-level request signal must be at least one CLK period long for the 8088 to recognize it. The processor will output a 0 for one CLK period during the next  $T_4$  or  $T_1$  state to acknowledge the takeover. It will then enter a "hold acknowledge" state until the new bus master sends another low-level pulse to  $\overline{RQ}/\overline{GT}$ . All bus takeovers must consist of this three-pulse sequence. It is important to note that the device performing the DMA is responsible for maintaining the DRAM refresh requirements, either by performing them itself, or by allowing them to happen normally with existing circuitry.

Still another way to transfer data to an external device is through a technique called memory-mapped I/O, which is slower than DMA but just as useful. It is even possible for a system to employ both DMA and memory-mapped I/O, using DMA for the high-speed transfers and memory-mapped I/O for the slow ones.

---

## 9.11 MEMORY-MAPPED I/O

Normally, a memory location, or group of locations, is used to store program data and other important information. Data is written into a particular memory location and read later for use. Through a process called **memory-mapped I/O**, we remove the storage capability of the memory location and instead use it to communicate with the outside world. Imagine that you have a keyboard that supplies an 8-bit ASCII code (complete with parity) whenever you press a key. Your job is to somehow get this parallel information into your computer. By using memory-mapped I/O, a memory *location* may be set aside that, when read, will contain the 8-bit code generated by the keyboard. Conversely, data may be sent to the outside world by writing to a memory-mapped output location. The 8088 CPU is capable of performing memory-mapped I/O in either byte or word lengths. All that is required is a memory address decoder, coupled with the appropriate bus circuitry. For a memory-mapped output location, the memory address decoder provides a clock pulse to an octal flip-flop capable of storing the output data. A memory-mapped input location would use the memory address decoder to enable a tri-state octal buffer, placing data from the outside world onto the CPU's data bus when active. Figure 9.28 shows the circuitry for an 8-bit memory-mapped I/O location, sometimes referred to as a memory-mapped I/O port. The memory address decoder may be used for both input and output.



**FIGURE 9.28** Memory-mapped I/O circuitry

## 9.12 TROUBLESHOOTING TECHNIQUES

As we have seen before, a good knowledge of binary numbers is beneficial when working with microprocessors, and necessary to the efficient design of address decoders (and other interfacing circuitry). Once the decoders are designed, however, they must be tested to see whether they perform as required. Testing the operation of a memory address decoder can be accomplished any number of ways. The circuit can be set up on a breadboard, simulated via software, or just plain stared at on paper until it seems correct.

In situations like the troubleshooting phase of the single-board computer project in Chapter 12, testing the memory address decoder (similar to Figure 9.14) is often necessary. In addition to checking the wiring connections visually (or via a continuity tester or DMM), a logic analyzer is connected so that the waveforms can be examined. Even an old eight-channel logic analyzer can be used to diagnose difficult problems in a microprocessor-based system if the clock speed is not too high. Oscilloscopes typically fall short in showing the associated high-speed timing relationships (unless they are storage scopes or the program is executing a loop).

The logic analyzer is connected so that all of the inputs to the address decoder are sampled, and as many of the outputs, as necessary. In addition, the logic analyzer is set up so that it triggers (and begins capturing data) at RESET, and the initial activity of the



processor's address bus can be observed. If the single-board's EPROM is not enabled, the system will not function at all. The logic analyzer will show how the address decoder responds at power on.

The logic analyzer can also be used to examine the data coming out of the EPROM or RAM. Sampling the data and a few of the address lines should be enough to verify whether the data is correct. It is sometimes possible to spot switched data or address lines this way.

These techniques also apply to I/O circuitry, which we examine in the next chapter.

---

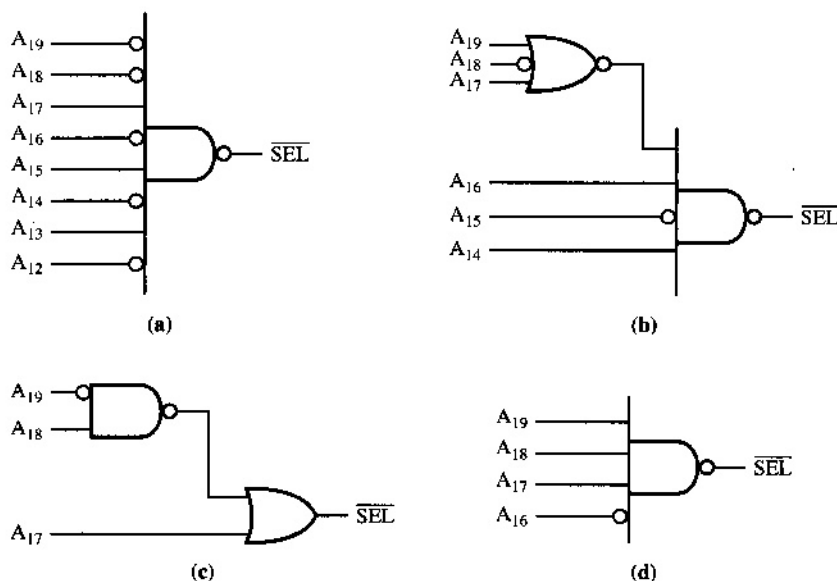
## SUMMARY

In this chapter we studied some of the most common methods used in the design of memory circuitry for microprocessor-based systems. Full- and partial-address decoding, memory-mapped I/O, direct memory access, and the logical requirements for static and dynamic RAMs were all covered. A good designer will employ many of these techniques in an effort to construct a new system that is logically simple and elegant but also functional and easy to troubleshoot. The end-of-chapter questions are designed to further test your knowledge of these topics. You are encouraged to work *all* of them to increase your ability to design memory address decoders, partial-address decoders, and complete memory systems.

---

## STUDY QUESTIONS

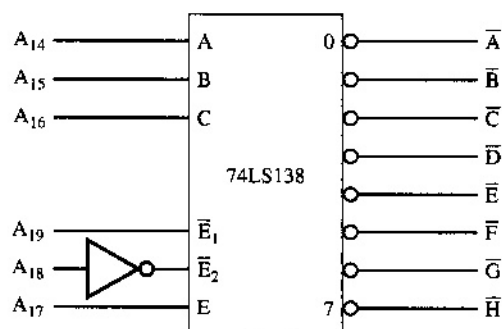
1. Explain the different functions associated with processor signals  $AD_0$  through  $AD_7$ .
2. How does external circuitry know when address information is present on the multiplexed address/data bus?
3. List the different control bus signals used in minimum mode and maximum mode.
4. How can the  $\overline{RD}$ ,  $\overline{WR}$ , and  $IO/\overline{M}$  outputs be used to detect *any* kind of access to memory? Design a circuit that will output a 0 on  $\overline{MEMORY}$  whenever a memory read or write occurs.
5. If a state time on an 8088-based system is 250 ns, what is the minimum time spent doing a memory read?
6. When (and why) are wait states inserted into memory accesses?
7. How many address lines are needed for a 128KB memory? For a 2MB memory?
8. For the state time of Question 5, what is the time spent doing a memory read with two wait states?
9. Two 2KB EPROMs are used to make a 4KB memory. How many address lines are needed for the EPROMs? What upper address lines must be used for the decoder?
10. For the memory of Question 9, what is the address of the last memory location, if the starting address of the EPROM is E4000?
11. Design a memory address decoder for the EPROM memory of Question 10, using a circuit similar to that in Figure 9.10.
12. Repeat Questions 9 through 11 for these memory sizes and starting addresses:
  - (a) 8KB, base address of CC000
  - (b) 32KB, base address of 80000
  - (c) 256KB, base address of 00000
13. What are the decoded address ranges for the circuits in Figure 9.29?



**FIGURE 9.29** For Questions 13 and 14

14. What signal (or signals) is missing from the address decoder in Figure 9.29? Modify the decoders to include the missing signal (or signals).
15. What are the address range groups for the decoder in Figure 9.30?

**FIGURE 9.30** For Questions 15 and 16



16. Use a circuit similar to that of Figure 9.30 to decode these address ranges:
  - 18000 to 187FF
  - 18800 to 18FFF
  - 19000 to 197FF
  - 19800 to 19FFF
  - 1A000 to 1A7FF
  - 1A800 to 1AFFF
  - 1B000 to 1B7FF
  - 1B800 to 1BFFF
17. What are two main advantages gained in using partial-address decoding? Two disadvantages?
18. Give three possible address ranges for each decoder in Figure 9.31. Address lines  $A_0$  through  $A_{13}$  are used by the memories.

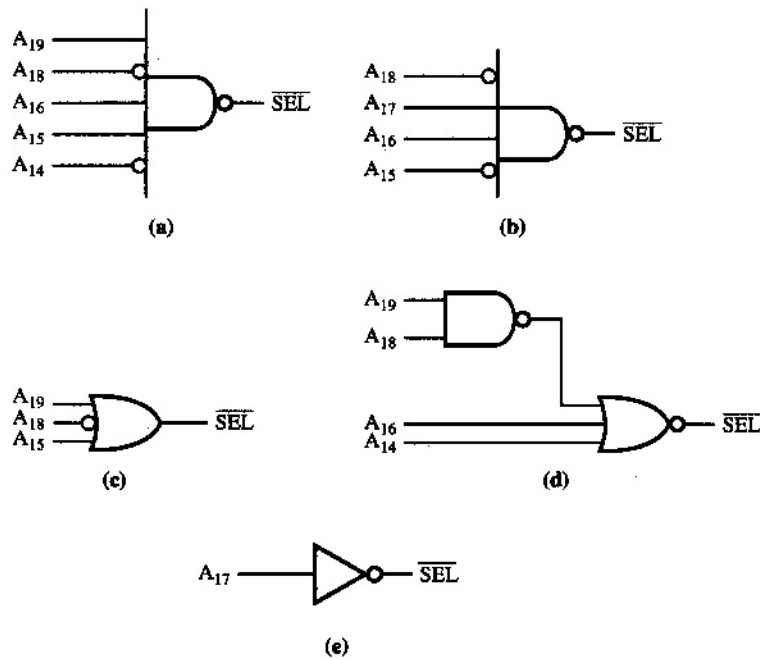


FIGURE 9.31 For Question 18

19. Suppose that three different memory decoders have output signals  $\overline{RAMA}$ ,  $\overline{RAMB}$ , and  $\overline{ROM}$ . Design a circuit to generate a READY delay of 200 ns using a 100-ns-period clock and a circuit similar to that of Figure 9.19. Any of the three signals going low triggers the generator.
20. Use an 8-bit parallel-out shift register to design a variable wait-state circuit. Assume that the shift register is clocked once every 125 ns, and that 0 to 7 125 ns wait states are allowed.
21. Design a 32KB memory using 8KB EPROMs. Show the address and data line connections to all EPROMs and the circuitry needed to switch between the four 8KB sections.
22. How do the  $\overline{RAS}$  and  $\overline{CAS}$  lines on a DRAM eliminate half of the required chip address lines?
23. Why does the size of a DRAM go up by a factor of 4 for each single address line that is added?
24. Why do DRAMs consume less power than static RAMs?
25. Explain how DRAM refreshing could be accomplished using an interrupt service routine.
26. How does program execution change on a system that supports DMA?
27. What is the 8088 doing while its external buses are involved in a DMA transfer?
28. Design a partial-address decoder for a 64KB EPROM with a base address of 40000.
29. Redesign the circuit of Example 9.6 using NAND gates.
30. What are the ranges of addresses for the partial-address decoders of Example 9.7?

---

# CHAPTER 10

---

## I/O System Design

---

### OBJECTIVES

In this chapter you will learn about:

- I/O addressing space
- The design of full and partial I/O address decoders
- The operation of buffered input ports
- The operation of latched output ports
- Parallel I/O with the 8255 PPI
- Serial I/O with the 8251 UART

### KEY TERMS

Base port address

Comparator

Input port

I/O addressing space

Output port

Parallel I/O

Partial port address decoding

Port address decoder

Serial I/O

---

## 10.1 INTRODUCTION

In Chapter 9, we saw how the 8088 is connected to its memory system. Program data and instructions are stored in memory and accessed over a system bus consisting of address, data, and control signals. Many times, a microprocessor is used to control a process that requires an exchange of data between the processor and the hardware used by the process. For example, an 8088 controlling an assembly line may receive input data from switches or photocells that give the position of an assembly making its way down the line. The 8088 will be able to test the state of each sensor by reading its status with an input port. Indicator lights, solenoids, and video display terminals associated with the assembly line may be driven by a few of the 8088's output ports. In this chapter, we will examine the design and operation of input and output ports, and see how they are used to communicate with the outside world.

As usual, our discussion applies to the advanced 80x86 machines as well, because the original bus architecture of the 8088 is still supported.

Section 10.2 discusses the processor's I/O addressing space. Section 10.3 shows how a port address decoder is designed. The operation of input and output ports is covered in Sections 10.4 and 10.5, respectively, followed by binary counter and D/A conversion applications in Section 10.6. The 8255 PPI and 8251 UART are explained in Sections 10.7 and 10.8, respectively, including examples of serial data transmission, A/D conversion, and parallel I/O. Troubleshooting techniques for I/O operations are covered in Section 10.9.

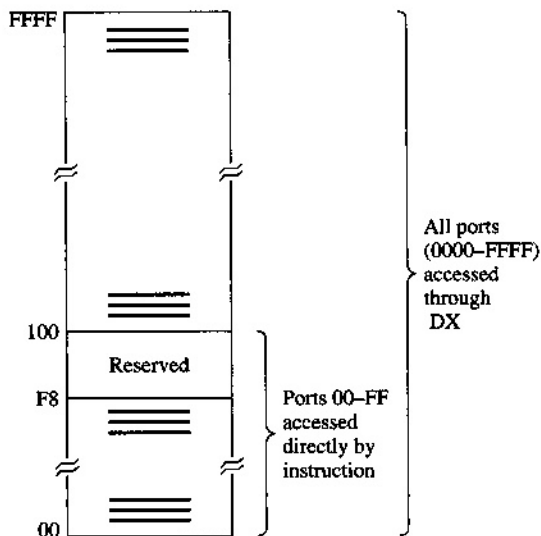
## 10.2 THE 8088 PORT ADDRESSING SPACE

Chapter 9 showed that the 8088 had a memory address space whose 1MB size is determined by the processor's 20-bit address bus. Unique binary patterns on  $A_0$  through  $A_{19}$  select one of the 1,048,576 locations within the processor's memory space. The 8088's **I/O addressing space** is smaller, containing just 65,536 possible input/output ports, accessed by  $A_0$  through  $A_{15}$ . Data transfer between ports and the processor is over the CPU data bus, and may contain 8 or 16 bits of data. The accumulator (AL for 8-bit transfers and AX for 16-bit transfers) is the only processor register involved in an I/O operation, with the single exception of the use of DX as the port address register. Figure 10.1 shows the organization of the 8088's I/O space. Ports 00 through FF may be addressed by one form of the IN and OUT instructions, and the entire I/O space (ports 0000 to FFFF) by another form of IN and OUT that uses register DX. In the first form, the port address may be used directly in the instruction, as in:

```
IN    AL, 80H
IN    AX, 6
OUT   3CH, AL
OUT   0A0H, AX
```

The port address in this form is limited to the range 00 to FF. Intel Corporation reserves the right to use ports F8 through FF for its own needs, and programmers are urged to consider

**FIGURE 10.1** The 8088's I/O addressing space



other addresses in their designs. Port addresses appear on address lines  $A_0$  through  $A_7$  when these instructions are executed.

The second way a port address may be specified is by placing it into register DX and using one of these instructions:

```
IN    AL, DX
IN    AX, DX
OUT   DX, AL
OUT   DX, AX
```

A good practice to follow is to use even port addresses for word transfers. This ensures the fastest 2-byte transfer the processor can perform. Word read or writes to odd I/O addresses require additional clock cycles.

When using DX as the port address register, the I/O addressing space becomes 0000 to FFFF, with the port address showing up on  $A_0$  through  $A_{15}$ . This represents 65,536 8-bit ports (or 32K 16-bit ports). In the next section, we will see how a port address is recognized and decoded.

## 10.3 DESIGNING A PORT ADDRESS DECODER

The **port address decoder** is a circuit designed to recognize the execution of an I/O instruction. In minimum mode, the processor uses the  $\overline{IO/\overline{M}}$  signal to indicate an I/O access by placing a high logic level on it, along with the port address on the address bus. In maximum mode, the 8288 bus controller decodes an I/O access and generates active  $\overline{IORC}$  (I/O read command) and  $\overline{IOWC}$  (I/O write command) signals. These control signals must be incorporated within the design of the port address decoder to distinguish port addresses from memory addresses.

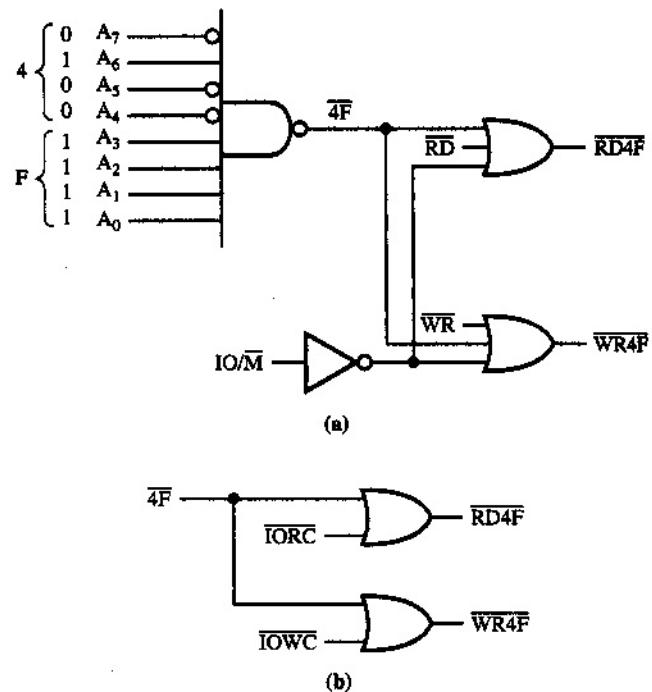
### Full Port Address Decoding

In this type of port address decoder, we include as many address lines as possible in the decoder. For example, Figure 10.2(a) shows the logic needed to do a full decode of port address 4F in a minmode system. Because the port address is between 00 and FF, it can be directly included in the I/O instruction (as in `IN AL, 4FH`), and we need only examine  $A_0$  through  $A_7$  for the required port address. We then combine the valid port address with  $\overline{IO/\overline{M}}$  to generate the  $\overline{RD4F}$  and  $\overline{WR4F}$  signals. These two outputs can also be generated with  $\overline{IORC}$  and  $\overline{IOWC}$ , as shown in Figure 10.2(b).

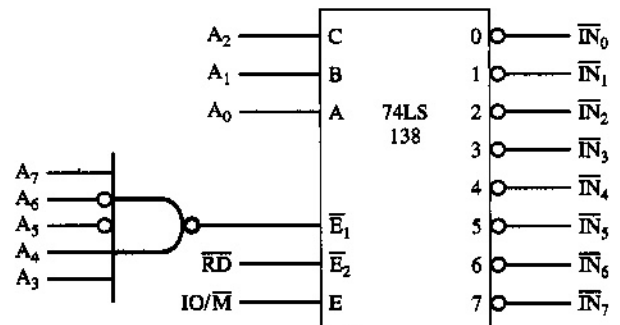
When groups of ports are needed, a technique similar to the one used to map multiple RAMs or EPROMs is used. In Figure 10.3, we see how a 3-line to 8-line decoder is used to decode signals for eight different input ports. Address lines  $A_0$  through  $A_2$  are used by the 74LS138 to select one of its eight outputs (output 0 when all three are low and output 7 when all three are high). Address lines  $A_3$  through  $A_7$  enable the 138 when the correct address is present. The correct port address is any address that matches 10011---. The **base port address**, when  $A_0$  through  $A_2$  are low, is 98H. The last port address is 9FH. So `IN AL, 98H` will cause the  $\overline{IN_0}$  signal to strobe low, `IN AL, 99H` will activate  $\overline{IN_1}$  and so on. What needs to be changed to get this circuit to work for output ports in the same range? What must be done to decode port addresses 80 through 87?

If the port address has been placed in register DX, address lines  $A_0$  through  $A_{15}$  must be used in the decoder. Figure 10.4 shows how port address A4C0 is decoded in a maxmode

**FIGURE 10.2** Port address decoder: (a) minmode system; (b) maxmode system

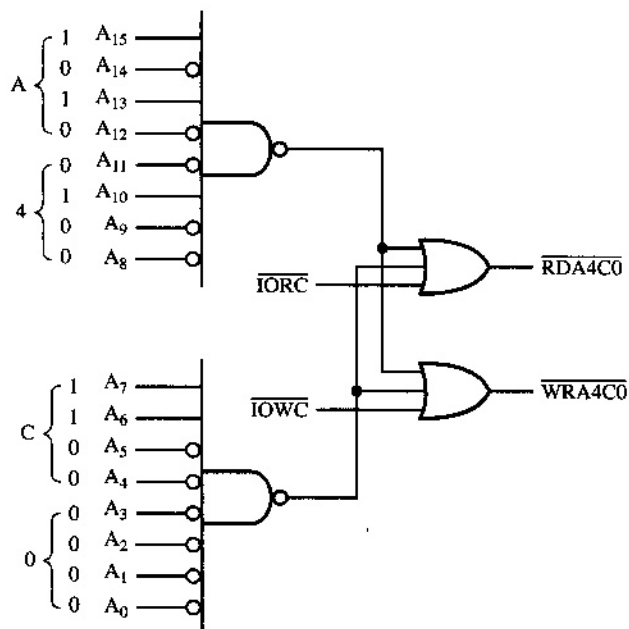
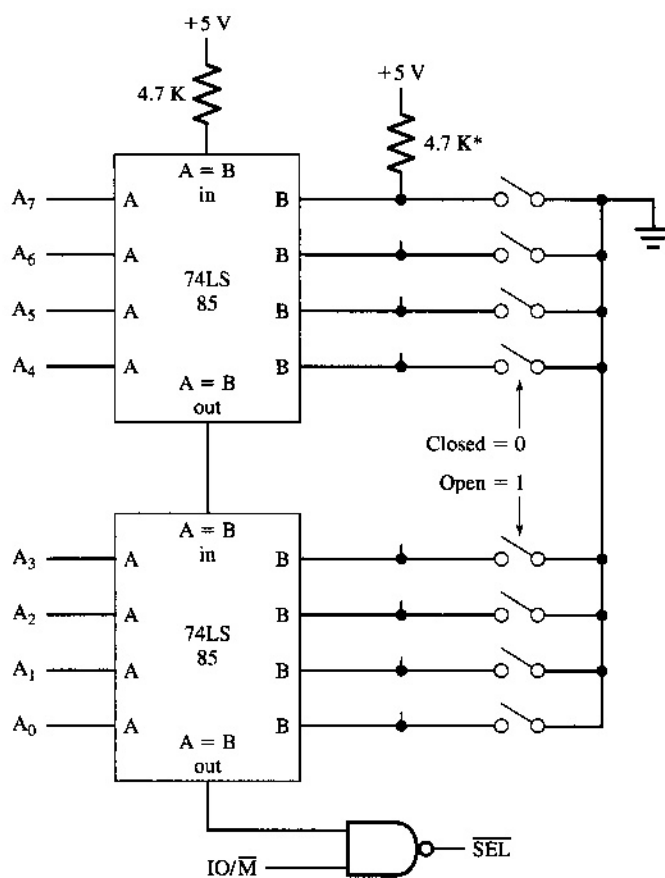


**FIGURE 10.3** Decoding multiple input port addresses 98H through 9FH



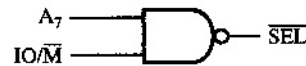
system. One 8-input NAND gate is used for each half of the address bus. One NAND gate recognizes A<sub>4</sub> and the other C<sub>0</sub>. The outputs of each NAND gate are combined with  $\overline{IORC}$  and  $\overline{IOWC}$  to generate the required I/O signals.

An important point to remember when working with these kinds of address decoders is that they decode a *fixed* port address, or range of addresses. Manufacturers who design I/O boards for consumer use (within their PCs) know that each user who buys a board may have a different I/O address in mind, depending on how each system is configured with other hardware items. It would be much more convenient to design a port address decoder that allows selection of a port address through a DIP switch. In this case, some type of binary **comparator** must be used to compare the port address on the address bus with the desired port address represented by a DIP switch. Figure 10.5 shows how two 74LS85 4-bit magnitude comparators can be used to recognize an 8-bit port address. Each magnitude comparator determines if the address signals on the four A inputs are equal to the DIP switch information on the four B inputs. Each comparator has an A = B input and A = B output. To perform an 8-bit comparison, the 85s are cascaded by connecting the first A = B output

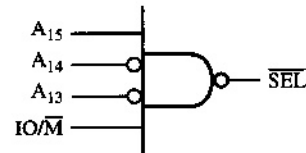
**FIGURE 10.4** Decoding a 16-bit port address**FIGURE 10.5** Using comparators in an address decoder

\*Each switch requires its own 4.7 K pullup resistor.



**FIGURE 10.6** Partial-address decoders

(a) For 8-bit port addresses 80 to FF



(b) For 16-bit port addresses 8000 to 9FFF

to the second  $A = B$  input. The  $A = B$  output of the second comparator will go high only when there is an 8-bit address match.  $\overline{SEL}$  will go low if the match exists when  $\overline{IO/M}$  is high.  $\overline{RD}$  and  $\overline{WR}$  can be combined with  $\overline{SEL}$  to get the required I/O read and write signals.

### Partial Port Address Decoding

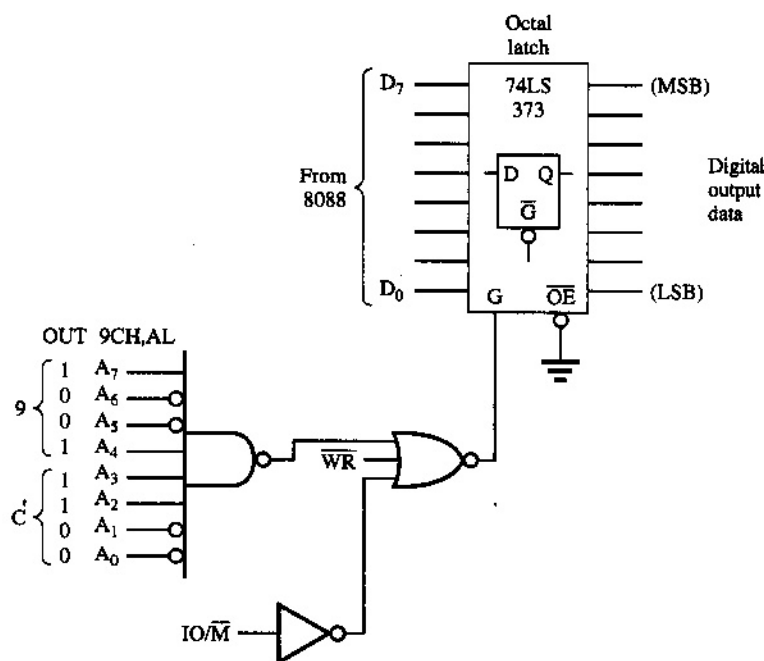
In a minimal system requiring only a handful of I/O ports, we can use **partial port address decoding** to reduce the hardware needed to produce I/O read and write signals. As we saw with partial-address decoders for memory systems, many address lines are not included in the decoder, setting the stage for groups of addresses having the ability to enable the same I/O port. For example, in Figure 10.6(a) a two-input NAND gate is used to decode any port address with  $A_7$  high. This would correspond to port addresses 80 through FF. Any reference to a port in this range will cause  $\overline{SEL}$  to go low. Inverting  $A_7$  before it gets to the NAND gate will decode ports in the range 00 to 7F. Figure 10.6(b) uses a four-input NAND gate to generate  $\overline{SEL}$  for a 16-bit port address in the range 8000 to 9FFF. Any one of the 8192 port addresses in this range will cause  $\overline{SEL}$  to go low. For practice, redesign the decoder of Figure 10.3 so that the eight port addresses are decoded whenever the decoder sees a port between 40 and 7F.

## 10.4 OPERATION OF A BUFFERED INPUT PORT

An **input port** is used to gate data from the outside world onto the CPU data bus, where it is captured by the processor and stored in the accumulator. It is not possible to simply place the digital data onto the data bus any time we please, for this will surely interfere with the program instruction and data bytes constantly being read from memory. The processor indicates the appropriate time to do the I/O read by activating  $\overline{IO/M}$  and  $\overline{RD}$  (or with  $\overline{IORC}$ ). In addition to the port address decoder, we then need a device to gate the input data onto the bus during the read operation, and to isolate the input data from the bus when the processor is not reading the input port. A tri-state buffer is used for this purpose. Figure 10.7 shows a design for an 8-bit input port located at the fully decoded port address 9C. The 74LS244 octal buffer is a tri-state device, meaning that its outputs are capable of going into a high-impedance state, effectively disconnecting the 244 from the processor's data bus. The high-impedance state can be thought of as an open switch. This is the normal state of the 244. When a valid port address appears on the address bus, the 244 will be enabled

**FIGURE 10.7** An 8-bit input port (address 9CH)

**FIGURE 10.8** An 8-bit output port  
(address 9CH)



## 10.6 SIMPLE I/O APPLICATIONS

In this section, we will examine two applications of I/O ports. The first application involves a binary counter, whose 8-bit count is displayed with light-emitting diodes connected to an output port. An input port wired to a set of switches is used to control the speed of the binary counter (all switches down is the fastest, all up is the slowest).

The second application shows how a data table containing sampled data from a sine wave can be output sequentially to a digital-to-analog converter to recreate the sine wave.

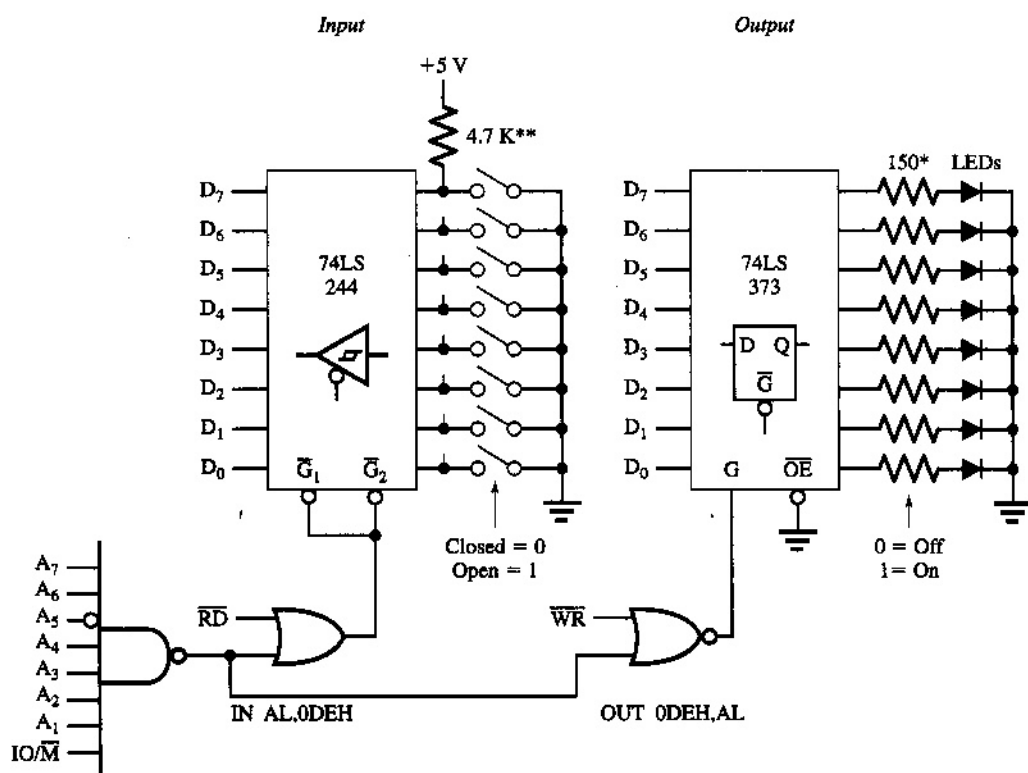
### The Binary Counter

Figure 10.9 shows the logic needed to implement an input/output port with a base address of DE. Because  $A_0$  is not used in the port address decoder, accesses to port DF will also work. The input port uses the 74LS244 octal buffer to read a set of DIP switches. An open switch makes a 1. The information present on the DIP switches is read with `IN AL,0DEH` and then moved into register CX. Register CX is used in a delay subroutine that is called between updates of the binary counter. Thus, the DIP switches have control over how fast or slow the binary counter counts. The count is displayed on a set of light-emitting diodes by outputting the count to port DE with `OUT 0DEH,AL`, which stores the count in the 74LS373.

The software required for this application is as follows:

```

BINCNT:  MOV     AL,0           ;start count at 00
DISPCNT: OUT     0DEH,AL       ;send count to LEDs
          CALL   FAR PTR DELAY ;pause
          INC    AL            ;increment counter
          JMP    DISPCNT       ;repeat forever
  
```



\*Use 150 in all positions

\*\*Use 4.7 K in all positions

FIGURE 10.9 An input/output port

```

DELAY      PROC      FAR
            MOV       BL,AL                ;save copy of counter
            IN        AL,0DEH              ;read DIP switches
            MOV       CH,AL                ;save speed byte
            MOV       CL,1                 ;ensure at least one loop
WAIT:      NOP
            NOP
            LOOP      WAIT                 ;waste a little time
            MOV       AL,BL                ;get counter value back
            RET
DELAY      ENDP

```

With the DIP switches all closed, a 0 is read into the accumulator, giving CX an initial value of 0001. This will cause DELAY to execute the WAIT loop once before returning. The counter is updated at the fastest rate in this case. Opening all of the DIP switches causes FF to be read into AL, giving CX an initial value of FF01. This will cause over 65,000 WAIT loops to be executed, giving the longest possible delay between updates, and thus the slowest count.

A useful exercise to complete this application would be to determine the amount of time between outputs to the display port (the LEDs) for a particular setting of the input switches.

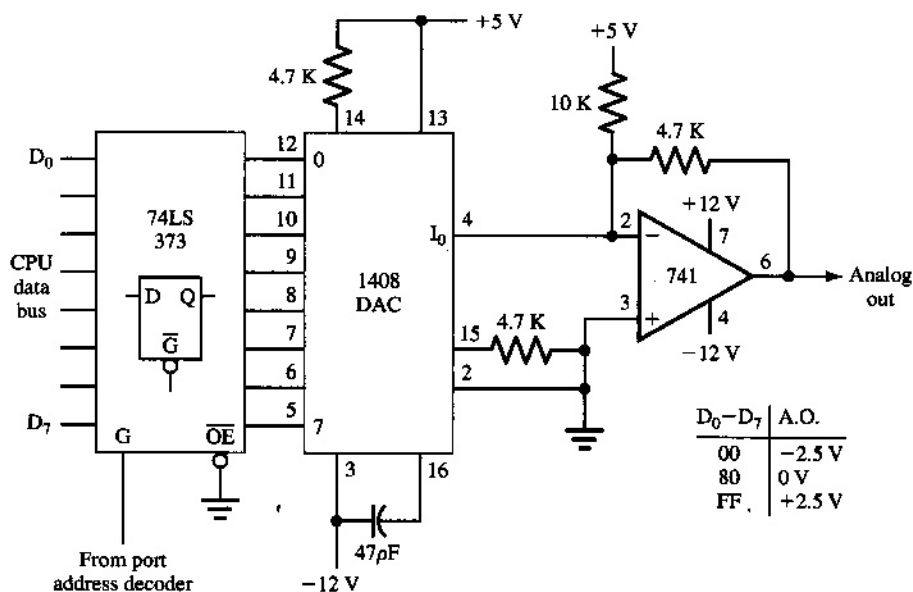


FIGURE 10.10 An 8-bit digital-to-analog converter

## Sine Wave Generator

In this application, an 8-bit digital-to-analog converter is connected to an output port. The circuitry of Figure 10.10 shows how the 1408 DAC is wired to a 74LS373. The 1408 is designed to sink current at its output, the amount of current between 0 mA and some maximum value (not to exceed 2 mA), depending on the binary number present on its inputs. The output current is converted into voltage by the 741 op-amp so that a  $\pm$  range is generated. We will see that  $-2.5$  V is created when 00 is at the DAC input, a value of 80 produces 0 V, and FF gives  $+2.5$  V.

Square waves can be generated by sending out two alternating binary values. For example, outputting 00...FF...00...FF... produces a square wave with a peak-to-peak voltage of 5 V. Sending 20...60...20...60... also produces a square wave, but with a different overall voltage. The software shown here generates a sine wave by outputting each value from the SINE data table. One pass through the table produces one cycle of a sine wave at the DAC output. The information in the SINE data table was computed by breaking one cycle of a sine wave into 256 equal parts of 1.4 degrees. At each degree increment (0, 1.4, 2.8, etc.), the value of  $\sin(x)$  is computed and multiplied by 128. The resulting value is then converted into binary. Try a few conversions for yourself and you should get the same pattern that appears in the table.

The frequency of the sine wave can be altered by playing with the DELAY subroutine.

In current data segment...

```

SINE    DB    82H,85H,88H,8BH,8EH,91H,94H,97H
        DB    9BH,9EH,0A1H,0A4H,0A7H,0AAH,0ADH,0AFH
        DB    0B2H,0B5H,0B8H,0BBH,0BEH,0C0H,0C3H,0C6H
        DB    0C8H,0CBH,0CDH,0D0H,0D2H,0D4H,0D7H,0D9H
        DB    0DBH,0DDH,0DFH,0E1H,0E3H,0E5H,0E7H,0E9H
        DB    0EBH,0ECH,0EEH,0EFH,0F1H,0F2H,0F4H,0F5H
        DB    0F6H,0F7H,0F8H,0F9H,0FAH,0FBH,0FBH,0FCB
  
```

```

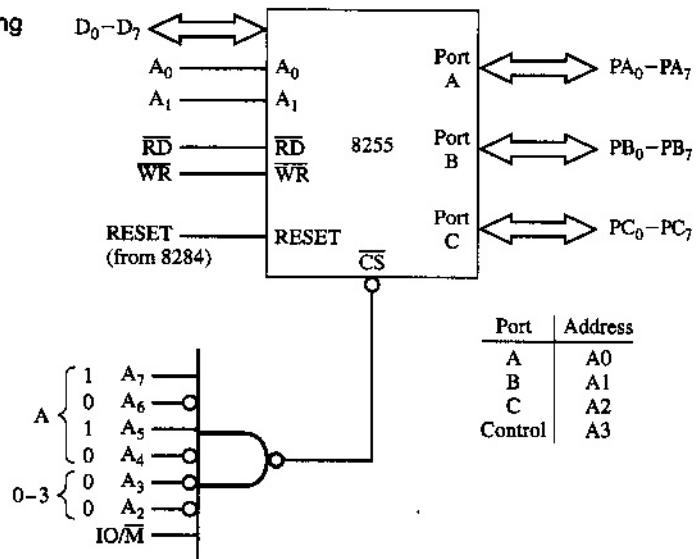
DB      0FDH,0FDH,0FEH,0FEH,0FEH,0FEH,0FEH,0FFH
DB      0FEH,0FEH,0FEH,0FEH,0FEH,0FDH,0FDH,0FCH
DB      0FBH,0FBH,0FAH,0F9H,0F8H,0F7H,0F6H,0F5H
DB      0F4H,0F2H,0F1H,0EFH,0EEH,0ECH,0EBH,0E9H
DB      0E7H,0E5H,0E3H,0E1H,0DFH,0DDH,0DBH,0D9H
DB      0D7H,0D4H,0D2H,0D0H,0CDH,0CBH,0C8H,0C6H
DB      0C3H,0C0H,0BEH,0BBH,0B8H,0B5H,0B2H,0AFH
DB      0ADH,0AAH,0A7H,0A4H,0A1H,9EH,9BH,97H
DB      94H,91H,8EH,8BH,88H,85H,82H,7EH
DB      7BH,78H,75H,72H,6FH,6CH,69H,66H
DB      62H,5FH,5CH,59H,56H,53H,50H,4EH
DB      4BH,48H,45H,42H,3FH,3DH,3AH,37H
DB      35H,32H,30H,2DH,2BH,29H,26H,24H
DB      22H,20H,1EH,1CH,1AH,18H,16H,14H
DB      12H,11H,0FH,0EH,0CH,0BH,9,8
DB      7,6,5,4,3,2,2,1,0,0,0,0,0,0,0,0
DB      0,0,0,0,0,0,0,1,2,2,3,4,5,6,7,8
DB      9,0BH,0CH,0EH,0FH,11H,12H,14H
DB      16H,18H,1AH,1CH,1EH,20H,22H,42H
DB      26H,29H,2BH,2DH,30H,32H,35H,37H
DB      3AH,3DH,3FH,42H,45H,48H,4BH,4EH
DB      50H,53H,56H,59H,5CH,5FH,62H,66H
DB      69H,6CH,6FH,72H,75H,78H,7BH,7FH
.
.
.
WAVER:  MOV    CX,256                ;init loop counter
        MOV    SI,0                  ;and pointer to data
SINOUT:  MOV    AL,SINE[SI]           ;read sine wave data
        OUT    0DEH,AL               ;send to 1408 DAC
        INC    SI                    ;point to next item
        CALL   FAR PTR DELAY         ;pause between outputs
        LOOP   SINOUT               ;repeat forever
        JMP    WAVER

```

The OUT instruction uses the same port address as the one implemented in the binary counter application. An analysis of the instruction cycles required for one pass through the SINOUT loop, *not including* the CALL to DELAY, gives 17 cycles. Multiplying this by 256 gives 4,352 CLK cycles used in the creation of one sine wave! If the processor is running at 5 MHz, this gives a sine wave frequency of over 1,100 Hz. A DELAY subroutine requiring 50 additional cycles produces a sine wave with a frequency of only 290 Hz. These estimates should give you an indication as to the limits of the sine wave generator. High-frequency waveforms will have to be created by other methods. Even so, this simple circuit can be used to create interesting audio effects by connecting the output of the 1408 to an amplifier. It is only a matter of sending the right data to the 1408.

## 10.7 PARALLEL DATA TRANSFER: THE 8255 PPI

In the previous section, we interfaced switches, lights, and a digital-to-analog converter to an I/O port. The number of applications is endless, with each one performing a different function. Even so, they all have at least one thing in common: they all use **parallel I/O**. In parallel I/O, all data bits are sent or received at the same time, as a group. This is very

**FIGURE 10.11** Interfacing the 8255 PPI

necessary in many applications! What would the binary counter look like if each LED were not updated at the same time as every other LED?

Many applications require more than one I/O port to get the job done. Peripheral designers realized this years ago and came up with a parallel I/O peripheral containing three separate I/O ports, all of which are programmable. This device is the 8255 programmable peripheral interface. In this section, we will see how the 8255 is interfaced to the 8088 and programmed.

### Interfacing the 8255

Figure 10.11 shows a diagram of the 8255 and its I/O and control signals. Twenty-four of the 8255's 40 pins are dedicated to the three programmable ports A, B, and C. These three ports, and a fourth one called a control port, are accessed via  $\overline{RD}$ ,  $\overline{WR}$ ,  $\overline{CS}$ , and address lines  $A_0$  and  $A_1$ . A RESET input is included to initialize the 8255 when power is first applied. Figure 10.11 shows how an 8-input NAND gate is used to decode port addresses  $A_0$  through  $A_3$ . When the address bus contains one of these four port addresses during an I/O access,  $\overline{CS}$  will be pulled low. The 8255 will internally decode the states of  $A_0$  and  $A_1$  and determine which port to access. In this example, port A has port address  $A_0$ . Ports B and C are accessed through ports  $A_1$  and  $A_2$ , respectively, and the control port is at  $A_3$ . It is very easy to determine the four port addresses by adding 0, 1, 2, and 3 to the base port address. The base port address is found by picking the upper six address lines ( $A_2$  through  $A_7$ ) to be what you need and assuming 0 for  $A_1$  and  $A_0$ .

The nicest feature of the 8255 is that different hardware circuits can be connected to ports A, B, and C, with the direction (input or output) of each port configured with initial programming. This allows an 8088-based system with an 8255 in it to be used for many different purposes.

### Programming the 8255

The 8255 has three modes of operation. The first is *mode 0: basic input/output*. In this mode, ports A, B, and C can be individually programmed as input or output ports. Port C is

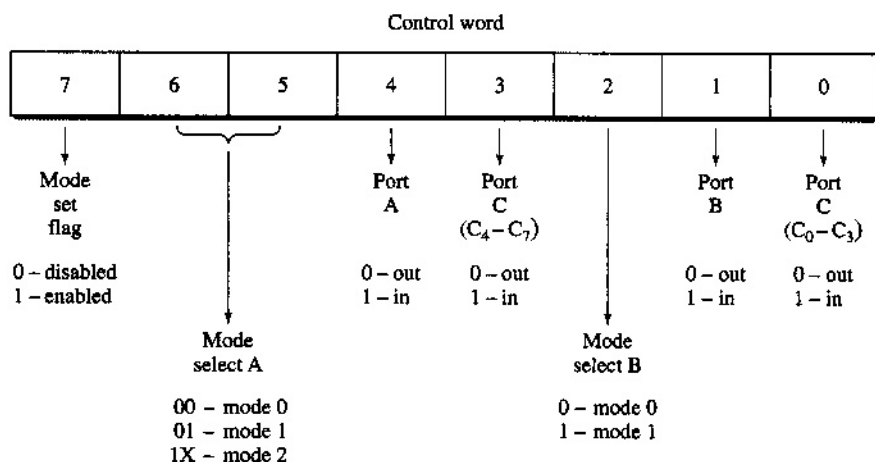


FIGURE 10.12 8255 mode word format

divided into two 4-bit halves, directionally independent from each other. So, there are sixteen combinations of input and output configurations available with this mode. A RESET automatically causes the 8255 to enter mode 0 with all ports programmed for input.

Input data is not latched. Data must be present when the port is being read by the processor. Output data is latched, as we would normally expect in an output port.

To program the 8255 for mode 0 operation and set the direction of each port, a mode word must be output to the control port. The definition of the mode word is shown in Figure 10.12. The MSB is the mode-set flag, which must be a 1 to program the 8255. Bits 5 and 6 are used to select the 8255's mode. 00 selects mode 0, 01 selects mode 1, and mode 2 is selected when bit 6 is high. Bit 2 is also used as a select bit for modes 0 and 1. The other 4 bits set the direction of ports A and B and both halves of C. A 0 indicates an output port and a 1 indicates an input port. To configure the 8255 for mode 0, all ports programmed for input, the mode word must be 10011011 (9BH). This byte must be output to the control port to configure the 8255. The following two instructions will initialize the 8255 after a reset:

```
MOV    AL, 9BH
OUT    0A3H, AL
```

Remember that the 8255 of Figure 10.11 has its control port at address A3.

Once the 8255 is programmed, the ports can be accessed with the appropriate IN instruction, such as IN AL, 0A0H (which reads port A). What mode word is needed to program port A for input, port B for output, and both halves of port C for input? You should get 99H when using the mode word format of Figure 10.12.

Assume that the 8255 has a DIP switch wired to port A and a set of LEDs wired to port B. The following code can be used to repeatedly read the switches and send their states to the LEDs.

```
READEM: MOV    AL, 99H        ;configure 8255 for Ain, Bout, mode 0
          OUT    0A3H, AL
GETSW:  IN     AL, 0A0H        ;read switches
          OUT    0A1H, AL        ;send data to lights
          JMP    GETSW
```

In this case, a closed switch turns an LED off.



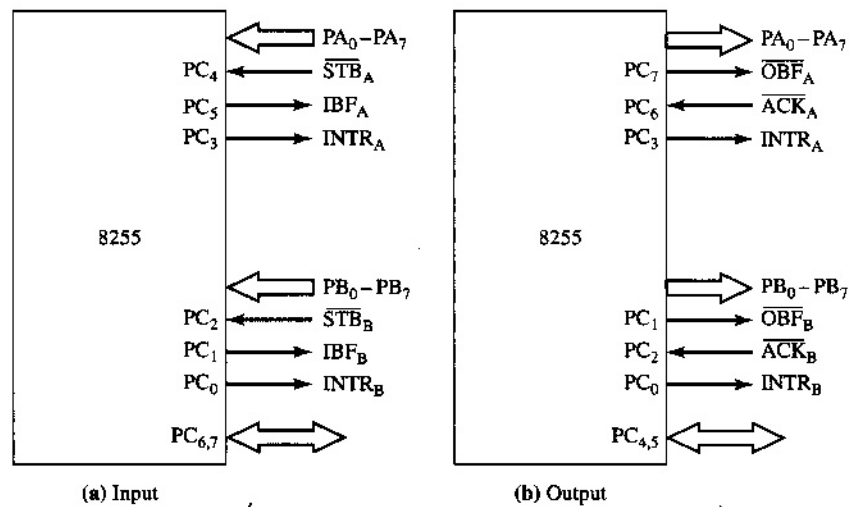


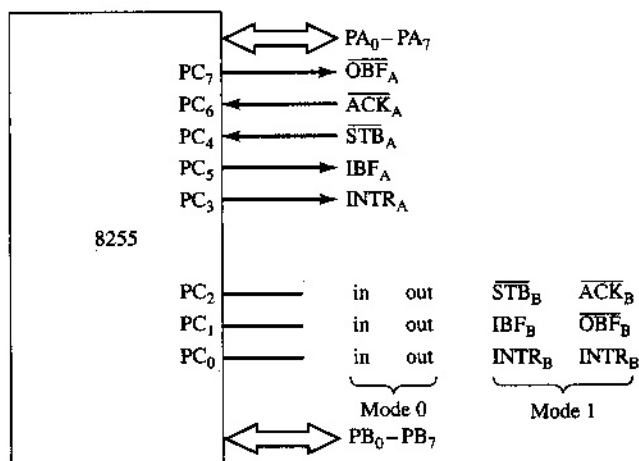
FIGURE 10.13 Mode 1 port definitions

The next mode is *mode 1: strobed input/output*. In this mode, the 8255 uses port C as a *handshaking* port. Handshaking signals are commonly used in printers to sense the status of the paper-out sensor and the printer's readiness to accept new data. Ports A and B can be programmed for input or output. Data are latched in both directions. If port A is programmed for input, a strobe signal is needed on PC<sub>4</sub> to write data into port A. The 8255 will acknowledge the new input data by outputting a 1 on PC<sub>5</sub>. These two signals on port C are defined as shown in Figure 10.13(a). PC<sub>5</sub> is IBF<sub>A</sub>, input buffer A full. IBF is cleared when the processor reads port A. Port B operates in the same way, using PC<sub>2</sub> and PC<sub>1</sub> as handshaking signals. Both ports have the capability of causing an interrupt when data is strobed into them. The INTR output will go high when IBF goes high and the internal interrupt-enable bit is set. PC<sub>4</sub> and PC<sub>2</sub> make up the interrupt-enable bits for ports A and B. Setting PC<sub>4</sub> will cause INTR<sub>A</sub> to go high when data is strobed into port A. Reading the input port will clear the interrupt request. This interrupt mechanism is a useful alternative to using software to constantly poll the input port. Polling wastes a lot of time waiting for input data that may not be there. Interrupting the processor only when new data has arrived results in more efficient program execution. This is one of the advantages of mode 1.

As Figure 10.13(a) shows, PC<sub>6</sub> and PC<sub>7</sub> are available for general purpose I/O when port A is programmed for input. The mode word needed to program the 8255 for mode 1, Ain, Bin, and PC<sub>6</sub> and PC<sub>7</sub> out is B6.

Figure 10.13(b) shows how the 8255 is configured for output in mode 1. Here we have *output buffer full* (OBF) and *acknowledge* (ACK) signals used to handshake with the output circuitry. OBF will go low when the processor writes to port A or B. This signal will remain low until a low pulse arrives on ACK. ACK is used to indicate that the new output data was received. ACK, together with interrupt-enable, can be used to generate an interrupt with INTR. This would interrupt the processor when the new output data has been read and avoid the need to poll the ACK signal. To program the 8255 for mode 1, Aout, and Bout, use mode word A4.

You may notice that the port C bits are assigned differently in the output configuration. For example, PC<sub>4</sub> and PC<sub>5</sub> are now used for general purpose I/O. The type of hardware configuration must be decided on, and then connected to the appropriate bits in port C.

**FIGURE 10.14** Mode 2 operation

The last mode is *mode 2: strobed bidirectional I/O*. This mode allows port A to operate as an 8-bit bidirectional bus. This is needed to allow the 8255 to be interfaced with 8-bit peripherals such as UARTS, which require a bidirectional data bus. Bits in port C are again used for handshaking and general purpose I/O, as indicated by Figure 10.14. Port B can operate as an input port or output port in mode 0 or mode 1. When operating port B in mode 0 (with port A in mode 2), PC<sub>0</sub> through PC<sub>2</sub> are available for general purpose I/O. The definitions for PC<sub>0</sub> through PC<sub>2</sub> in mode 1 apply when port B is operated in mode 1 with port A in mode 2.

As before, the INTR output can be used to interrupt the processor for input or output operations. When the CPU writes data to port A,  $\overline{\text{OBF}}$  will go low. To enable the output buffer on port A and read the data,  $\overline{\text{ACK}}$  must be pulled low. Data is written into port A by pulling  $\overline{\text{STB}}$  low. This will cause IBF to go high until the data is read by the processor.

### An 8255 Application: A/D Conversion

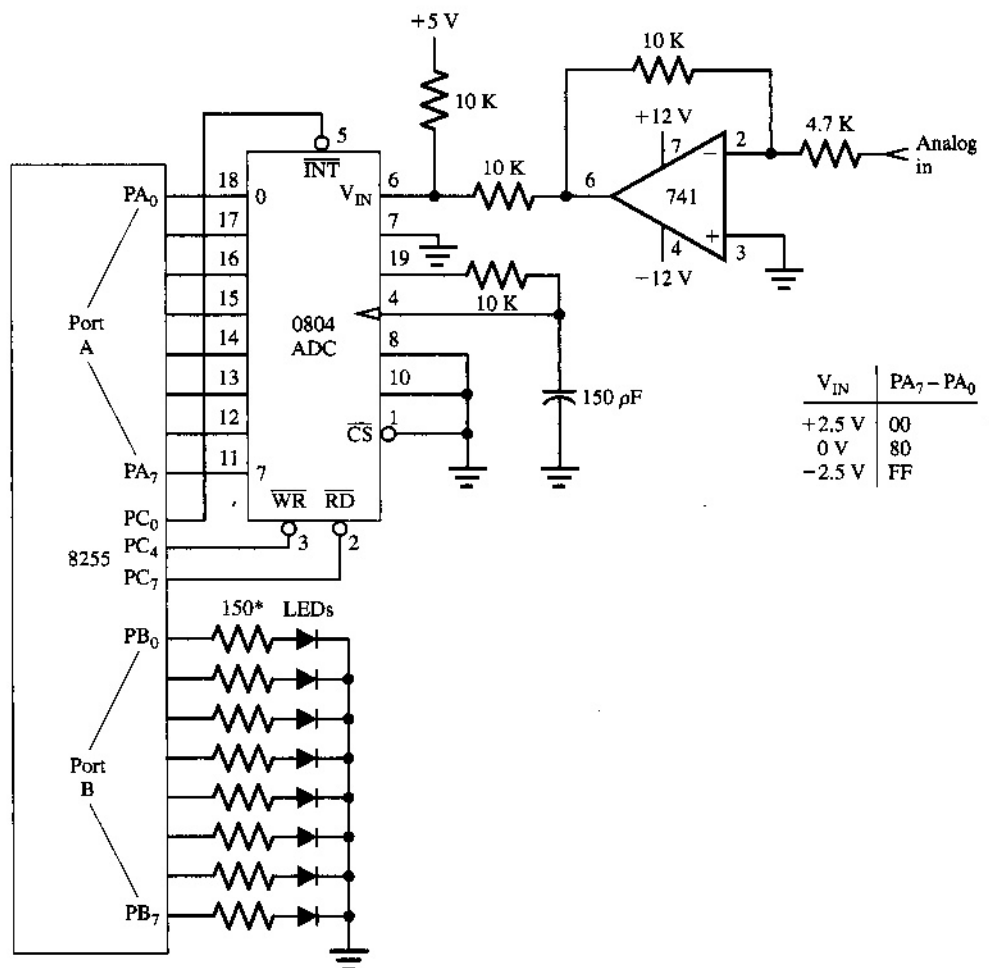
The 8255 in Figure 10.15 is configured for mode 0, port A in, port B out, PC<sub>0</sub> through PC<sub>3</sub> in, and PC<sub>4</sub> through PC<sub>7</sub> out. The 8255 has a base address of 40H. An 8-bit analog-to-digital converter (ADC0804) is connected to ports A and C. Port C is used to control the  $\overline{\text{RD}}$  and  $\overline{\text{WR}}$  inputs on the 0804, and to read the end-of-conversion status. The 741 op-amp converts a  $\pm$  input voltage swing into a 0–5 V signal that can be digitized by the 0804. To start a conversion, the 0804's  $\overline{\text{WR}}$  line must be pulled low. This will force the  $\overline{\text{INT}}$  output high until the conversion is complete. When  $\overline{\text{INT}}$  goes low, the 0804 can be read by pulling the  $\overline{\text{RD}}$  input low and reading port A.

The following subroutine is called to perform a single conversion and return the results in AH:

```

VCON  PROC  FAR
        MOV  AL,80H      ;start a conversion by pulling
        OUT  42H,AL      ;WR low
        MOV  AL,90H      ;now pull WR high
        OUT  42H,AL
EOC:   IN    AL,42H      ;read port C
        AND  AL,1        ;test bit-0 (INT)
        JNZ  EOC         ;wait for end of conversion
        MOV  AL,10H      ;enable RD
        OUT  42H,AL

```



\*Use 150 in all positions

**FIGURE 10.15** 8-bit analog-to-digital converter

```

IN      AL,40H      ;read port A (the 0804)
OUT     41H,AL      ;echo data to port B
MOV     AH,AL       ;return result in AH
MOV     AL,90H      ;get RD back to normal
OUT     42H,AL
RET
VCON    ENDP

```

The data read from the 0804 is sent out to the LEDs on port B. This gives a visual indication that everything is working properly. Being able to split up port C makes the interface with the 0804 easy to accomplish.

The 8255 is configured and initialized in this way:

```

MOV     AL,91H      ;mode 0, Ain, Bout, CLin, CHout
OUT     43H,AL      ;send to control port
MOV     AL,90H      ;RD and WR both high
OUT     42H,AL      ;send to port C

```

A routine to digitize a waveform presented to the analog input would require successive CALLs to VCON, storing AH in a data table each time VCON returns. Once the waveform has been digitized, the data bytes that represent it can be altered and then output to a digital-to-analog converter for playback.

### The Centronics Parallel Printer Interface

Another application involving parallel data transfer is the use of a parallel printer. A parallel printer connection, such as the Centronics™ standard, provides for communication between the computer and the parallel printer. ASCII codes are output to the printer, and printer status is monitored by the computer through signals to a DB25 connector, as shown in Figure 10.16. On the PC, three ports are used to interface with the printer. A data port (address 378H) outputs 8-bit ASCII information to the printer. A control port (address 37AH) supplies a number of control signals (such as *strobe* and *initialize printer*), and a status port (address 379H) monitors printer status. In general, to print a character, the following sequence must take place:

- Output ASCII code to data port
- Output a low-going strobe pulse
- Wait for a low-going acknowledge pulse

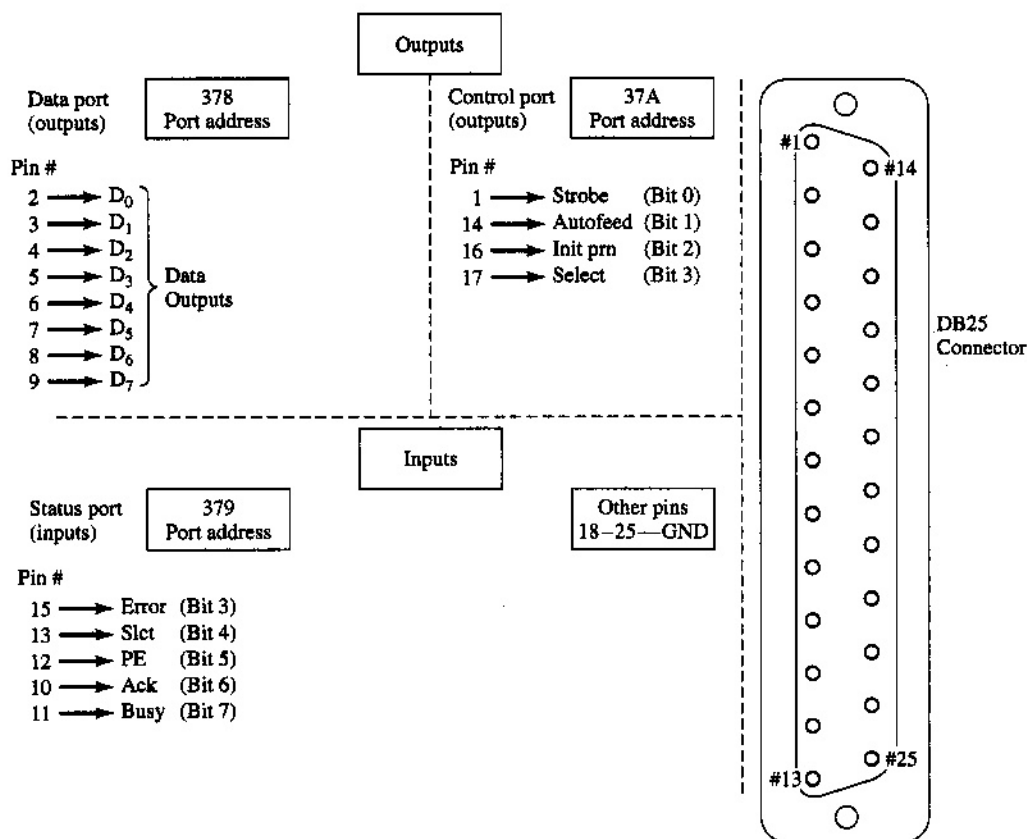
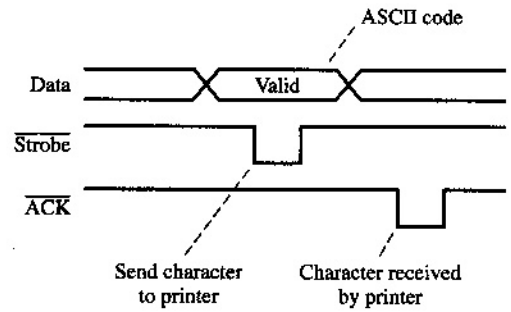


FIGURE 10.16 Centronics parallel printer connections

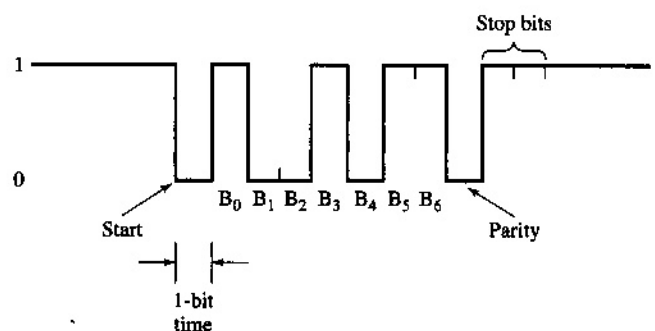
**FIGURE 10.17** Printer/computer handshaking

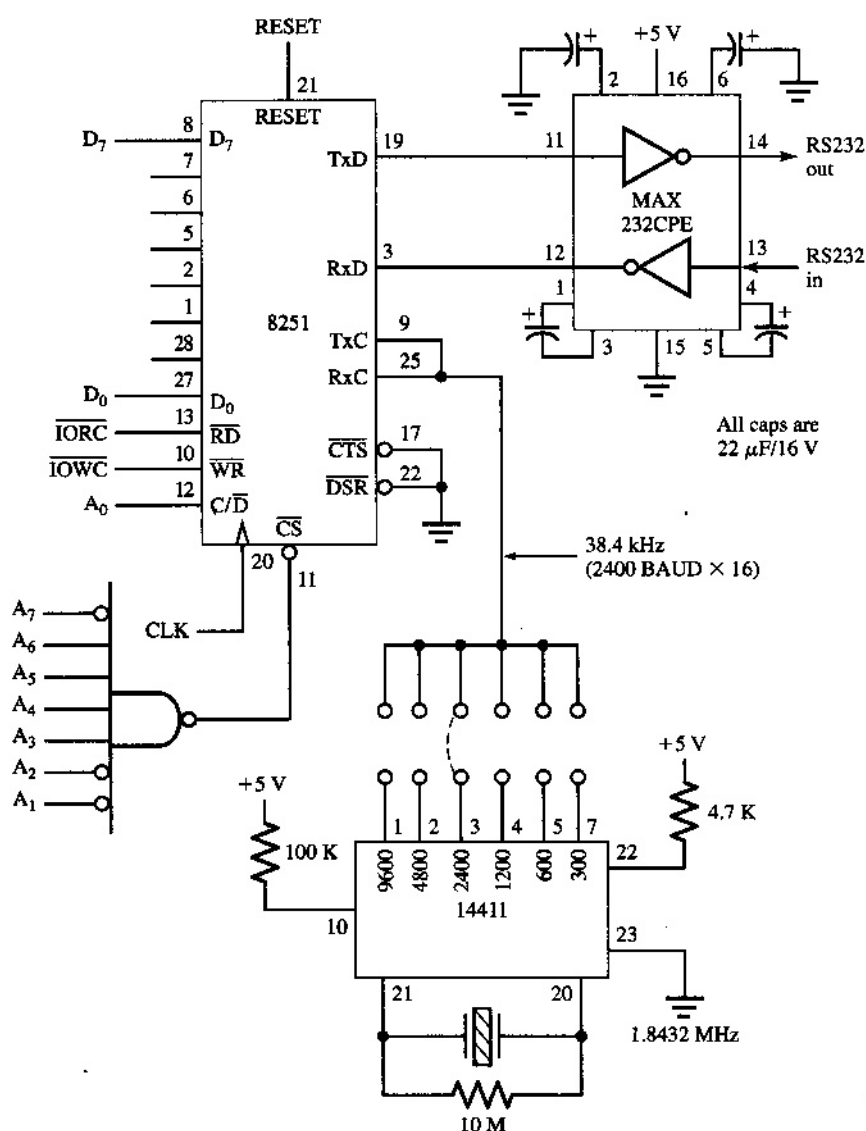
This sequence, illustrated in Figure 10.17, performs handshaking between the printer and the computer and guarantees that the computer does not send data to the printer faster than the printer can accept it.

## 10.8 SERIAL DATA TRANSFER: THE 8251 UART

Serial I/O offers the convenience of running a small number of wires between two points (three will do the job in most cases), while at the same time being very reliable. Although we must wait longer to receive our data because it is transmitted only 1 bit at a time, we are able to place our communication devices (computers, terminals), far away from each other. Worldwide networks now exist, connected via satellites, based on serial data transmission. The peripheral covered in this section, the 8251 UART, implements serial data transmission in a variety of formats. The standard serial data transmission waveform for any UART is depicted in Figure 10.18.

The normal state of the line is a logic 1. This level indicates that no activity is present (that is, no data is being transmitted). When the line level falls to a logic 0 (the start bit), the receiving UART knows that a new character is being transmitted. The data bits representing the character (or data) being transmitted are clocked out in the order shown, least significant to most significant. Following the data bits is the parity bit, which will be used by the receiving UART to determine the accuracy of the data it received. The parity bit in Figure 10.18 shows that the data has even parity. The last bits in any transmission are the stop bits, which are always high. This gets the line back into its inactive state. We are able to set the number of data bits, the type of parity used, the number of stop bits, and other parameters through software. Before we consider how to do this, let us examine the hardware operation of the 8251.

**FIGURE 10.18** Standard TTL serial data waveform



**FIGURE 10.19** 8251 to 8088 interface

## Interfacing the 8251

The 8251 was originally designed to be used with the 8080 and 8085 microprocessors, 8-bit machines that preceded the 8088. The 8088 interfaces with the 8251 easily, requiring the usual address decoder and a few control signals. Figure 10.19 shows a complete serial data circuit for the 8088. The 8251 is connected to the processor's data bus and  $\overline{\text{IORC}}$  and  $\overline{\text{IOWC}}$  signals. Because these signals are active only during I/O operations, the address decoder need only examine the state of the address bus. The NAND gate recognizes port addresses 78H and 79H. Address line  $A_0$  is not used in the decoder. Instead, it is connected to the 8251's  $C/\overline{D}$  input. This pin selects internal *control* or *data* registers. So, if  $A_0$  is low (port 78H) and  $\overline{\text{IORC}}$  is active, the UART's receive register will be read. If  $\overline{\text{IOWC}}$  is active with  $A_0$  low, the UART's transmitter register will be written to and a new transmission started.

The control functions are selected when  $A_0$  is high (port 79H). Reading from port 79H gets 1 byte of status information; writing to it selects functions such as data size and parity.  $\overline{CTS}$  (clear to send) and  $\overline{DSR}$  (data set ready) are handshaking signals normally used when the 8251 is connected to a modem. They are grounded to keep them enabled.  $TxC$  and  $RxC$  are the transmitter and receiver clocks. The frequency of the TTL signal at these inputs determines the bit rate and time of the transmitter and receiver. It is common to run the UART at a clock speed sixteen times greater than the baud rate. So, a 2,400 baud transmission rate requires a 38.4 kHz clock (multiply 2,400 by 16). This frequency and other standard baud rate frequencies are generated automatically by the 14,411 baud-rate generator. All that is needed is a 1.8432 MHz crystal.

Attempting to transmit a digital (0–5 V) signal over a long length of wire causes distortion in the signal shape due to the line capacitance. It was discovered that making the signal switch from a positive voltage to a *negative* voltage helps to eliminate the distortion. Higher baud rates are possible using the  $+/-$  swinging signal. A standard was developed for this type of signal, called the RS232C standard. Take another look at the waveform in Figure 10.18. This is the TTL waveform that comes out of the UART's transmitter. The RS232 waveform that gets transmitted over the wires is inverted and swings plus and minus. So, a high level on the TTL waveform creates a low (negative) level on the RS232C waveform. An integrated circuit capable of performing the RS232C-to-TTL conversions is the MAX232CPE. This chip is especially useful because, by adding four 22  $\mu$ F electrolytic capacitors, the MAX232 generates its own  $+/-$  10 V supply while needing only the standard 5 V. Older chips, such as the 1488 line driver and 1489 line receiver, required additional external power supplies. The MAX232 has two separate RS232C drivers/receivers for systems requiring two serial data channels.

## Programming the 8251

Because the 8251 is connected to RESET, we are assured that the 8251 is functional after a power-on. It is still necessary to program the 8251 to ensure that the correct number of data bits will be used, that the parity will be generated as expected, and so on. To program the 8251, a series of bytes are output to the control port (79H from our example). The first byte is called the mode instruction. The format of this byte is shown in Figure 10.20. The 8251 can operate in asynchronous mode or synchronous mode. In asynchronous mode, the baud rate is determined by the lower 2 bits in the mode instruction. If these 2 bits are low, synchronous mode is selected.

The number of data bits used in a transmission is selected by bits 2 and 3. To enable generation of a parity bit, bit 4 must be set. Odd or even parity is chosen by the setting of bit 5. Finally, the number of stop bits is chosen by the upper 2 bits in the mode instruction. The waveform of Figure 10.18 contained 7 data bits, an even parity bit, and 2 stop bits. The required mode instruction byte is FA. To program the 8251, use:

```
MOV    AL, 0FAH
OUT    79H, AL
```

Because an X16 clock was selected, the 8251 will operate in asynchronous mode. Synchronous mode is used for high-speed data transmission (not usually needed for communication with a serial display terminal). Synchronous mode is selected by making the lower two mode instruction bits 0. In this case, the upper two mode instruction bits do not set the number of stop bits, but rather the number of sync characters transmitted and the function of the SYNDET pin.

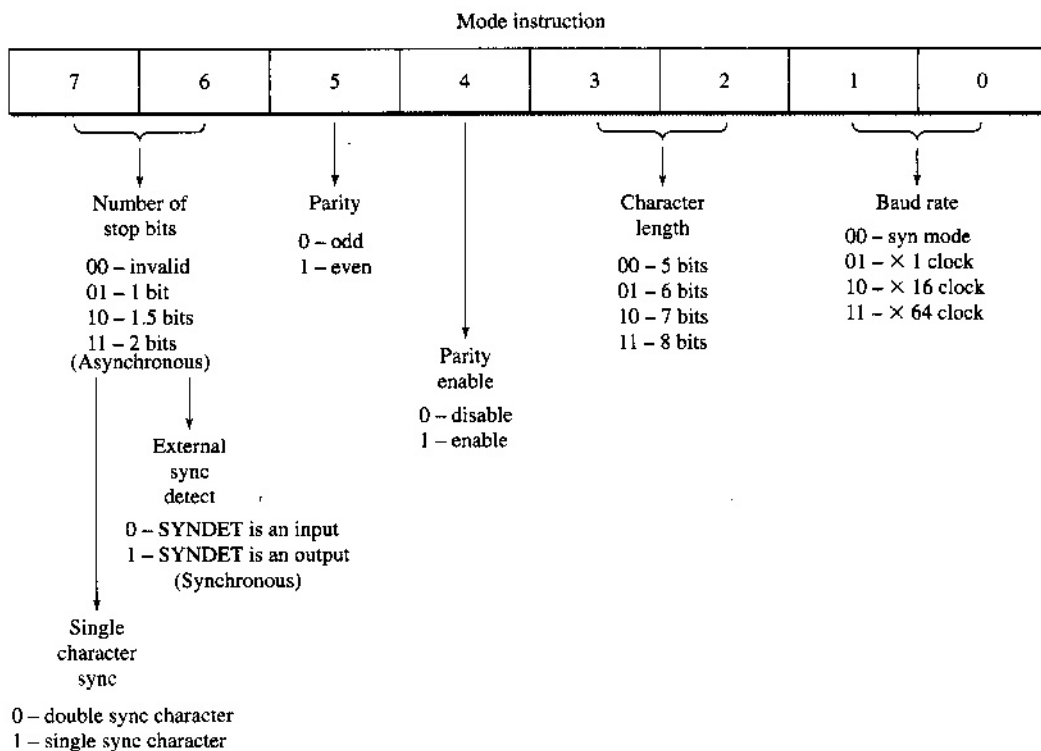


FIGURE 10.20 8251 mode instruction format

A second byte must be output to the control port to complete the initialization of the 8251. This byte is called the *command* instruction. The bits are assigned as shown in Figure 10.21; they have the following meanings:

Bit 0: transmit enable. Enable transmitter when this bit is set.

Bit 1: data terminal ready. Setting this bit will force the  $\overline{DTR}$  output low.

Bit 2: receive enable. Enable receiver when this bit is set.

Bit 3: send break character. Setting this bit forces  $TxD$  low.

Bit 4: error reset. Setting this bit clears the PE, OE, and FE error flags.

Bit 5: request to send. Setting this bit forces the  $\overline{CTS}$  output low.

Bit 6: internal reset. To reset the 8251 and prepare for a new mode instruction, this bit must be set.

Bit 7: enter hunt mode. Setting this bit enables a search for SYNC characters (in synchronous mode only).

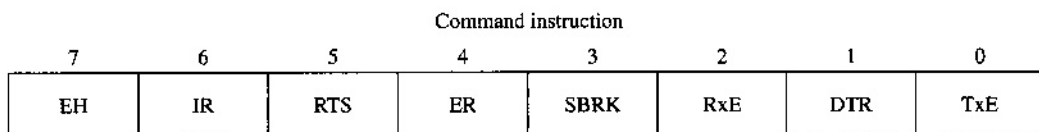


FIGURE 10.21 8251 command instruction format



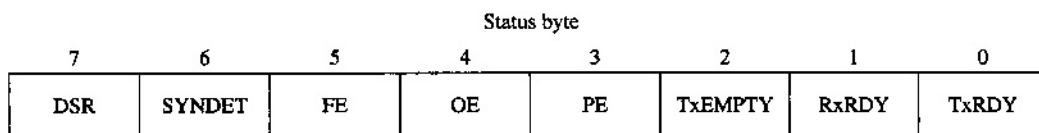


FIGURE 10.22 8251 status byte

The command instruction needed to enable the transmitter and receiver and ignore all other functions is 05H. This byte must be output to the control port after the mode instruction. So, to totally initialize the 8251 for operation in the circuit of Figure 10.19, we need these instructions:

```

MOV    AL,0FAH    ;mode instruction
OUT    79H,AL
MOV    AL,5        ;command instruction
OUT    79H,AL

```

Once the UART has been programmed, we have no need for the control port. Instead, we use the 8251's *status* port to help control the way data are transmitted and received. Figure 10.22 shows the bit assignments in the 8251's status port. Particularly important are the TxRDY (*transmitter ready*) and RxRDY (*receiver ready*) flags. They tell us when the transmitter is ready to transmit a new character and when the receiver has received a complete character. A number of error bits are included to show what may have gone wrong with the last reception. PE is parity error and will go high if the parity of the received character is wrong. OE is overrun error and will be set if a new character is received before the processor read the last one. FE stands for framing error and goes high when stop bits are not detected. SYNDET (*sync character detected*) will go high when a sync byte is received in synchronous mode. DSR (*data set ready*) will go high whenever  $\overline{\text{DSR}}$  is low.

The programmer must use the 8251's status bits to ensure proper serial data communication. Figure 10.23 shows how the first two bits are used to implement a simple serial input/output procedure.

Both flowcharts indicate that repeated testing of the RxRDY/TxRDY bits may be necessary. For example, to show the importance of this repeated testing, consider the following case. Suppose that an 8251 is configured to transmit and receive data at 1,200 baud, with 7 data bits, odd parity, and 1 stop bit. How long does it take to fully transmit or receive a character? At 1,200 baud, the bit time is just over 833  $\mu\text{s}$ , and the selected word length of 10 bits makes the total time to receive or transmit a single character roughly 8.3 ms.

It is not difficult to imagine how many instructions the 8088 might be able to execute in 8.3 ms. Would a few thousand be unreasonable? Probably not. Therefore, we use the status bits to actually slow down the 8088, so that it does not try to send or receive data from the 8251 faster than the 8251 can handle.

The two short routines that follow show how a character input and a character output routine might be written in 8088 code.

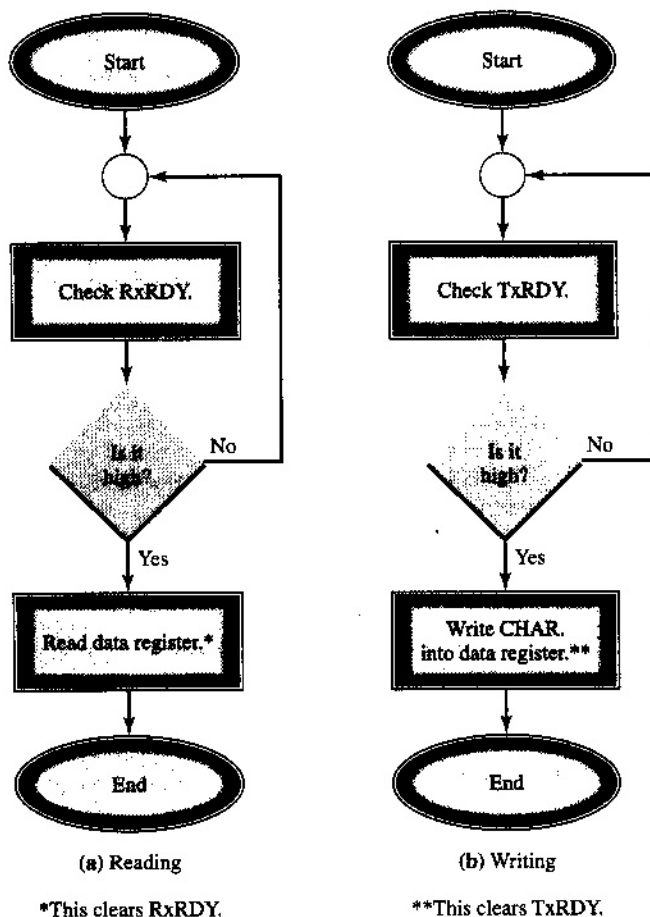
### Character Input

The character input routine is used to read a character from the receiver, returning it in AL. It is necessary to check the state of RxRDY before reading the receiver.

```

CHARIN  PROC    FAR
RSTAT:  IN      AL,79H    ;read UART status
        AND     AL,02H    ;examine RxRDY

```

**FIGURE 10.23** I/O flowcharts

```

JZ     RSTAT      ;wait until receiver is ready
IN     AL,78H     ;read receiver
AND    AL,7FH     ;ensure 7-bit ASCII code
RET

```

```
CHARIN  ENDP
```

The data byte received is ANDed with 7F to clear the MSB. Because the UART is being used for character transmission, only a 7-bit ASCII code is required.

## Character Output

This routine checks to see if the transmitter is ready for a new character. If it is, the ASCII character stored in AL is output to the transmitter.

```

CHAROUT  PROC      FAR
MOV      AH,AL      ;save character
TSTAT:   IN        AL,79H ;read UART status
AND      AL,1       ;examine TxRDY
JZ       TSTAT      ;wait until transmitter is ready
MOV      AL,AH      ;get character back
OUT      78H,AL     ;output to transmitter
RET
CHAROUT  ENDP

```

## An 8251 Application: Video Typewriter

Once the routines to communicate with the UART are in place, we can begin using them in applications. A video typewriter requires that a serial display terminal be connected to the UART that is controlled by CHARIN and CHAROUT. The simple programming loop that follows is used to echo every character received (presumably from the keyboard) to the screen. Line-feed characters are inserted when a carriage return is seen.

```
TVT:  CALL  FAR  PTR CHARIN    ;get keyboard character
      CALL  FAR  PTR CHAROUT   ;send it to screen
      CMP   AL,0DH             ;carriage return?
      JNZ   TVT
      MOV   AL,0AH             ;output a linefeed character
      CALL  FAR  PTR CHAROUT
      JMP   TVT
```

This simple routine can be modified to allow editing and other features and is left for you to think about on your own.

## READCOM1: A Serial I/O Application for the Personal Computer

The personal computer provides serial I/O through its COM1 connector. A mouse or modem commonly uses COM1 to communicate. The single-board computer presented in Chapter 12 is also capable of interfacing with the PC through COM1, which uses a UART similar to the 8251. Port addresses 3F8H through 3FFH are normally reserved for use with COM1 (with COM2 through COM4 using other ranges). Only two of these addresses are needed to perform simple serial I/O through COM1 once its UART has been initialized. These are

3F8H: Data port (receiver/transmitter data registers)

3FDH: Status port (receiver/transmitter ready flags)

Characters received by COM1 can be read by inputting from port 3F8H. Outputting to port 3F8H transmits a character.

Two status bits on port 3FDH must be used to synchronize the I/O operations:

Bit 0: receiver ready when high

Bit 5: transmitter ready when high

These status bits are tested in a manner similar to that used in the CHARIN and CHAROUT routines.

The READCOM1 program that follows reads characters received by COM1 and displays their 8-bit binary equivalents on the display. If a character is a printable ASCII character code, the actual character is output as well. The receiver-ready bit (bit 0 of port 3FDH) is examined through the use of the TEST instruction. Once the UART status has been placed in register AL, the instruction

```
TEST  AL,1
```

is used to check the state of bit 0.

;Program READCOM1.ASM: Read and display COM1 data.

```
;
      .MODEL  SMALL
      .DATA
XMSG  DB      'Press any key to exit...',0DH,0AH,'$'
DSTR  DB      ' is '
```

```

CDATA    DB    20H        ;reserved for COM1 data
          DB    0DH,0AH,'$'

          .CODE
          .STARTUP
          LEA    DX,XMSG        ;set up pointer to exit message
          MOV    AH,9           ;display string function
          INT    21H           ;DOS call
COM1:     MOV    DX,3FDH        ;set up status port address
MORE:     IN     AL,DX          ;read UART status
          TEST   AL,1           ;has a character been received?
          JNZ    READ          ;yes, go get it
AKEY:     MOV    AH,0BH        ;check keyboard status
          INT    21H
          CMP    AL,0FFH        ;has any key been pressed?
          JNZ    MORE
          JMP    BYE
READ:     MOV    DX,3F8H        ;set up data port address
          IN     AL,DX          ;read UART receiver
          MOV    CDATA,AL       ;save character
          CMP    AL,20H        ;test for printable ASCII
          JC     NONPRT
          CMP    AL,80H
          JC     OKVAL
NONPRT:   MOV    CDATA,20H      ;use a blank when non-printable
OKVAL:    CALL   DISPBIN        ;display data in binary
          LEA    DX,DSTR        ;set up pointer to data string
          MOV    AH,9           ;display string function
          INT    21H           ;DOS call
          JMP    AKEY          ;and repeat
BYE:      .EXIT

DISPBIN   PROC    NEAR
          MOV    CX,8           ;set up loop counter
NEXT:     SHL    AL,1           ;move bit into Carry flag
          PUSH   AX             ;save number
          JC     ITIS1          ;was the LSB a 1?
          MOV    DL,30H        ;load '0' character
          JMP    SAY01          ;go display it
ITIS1:    MOV    DL,31H        ;load '1' character
SAY01:    MOV    AH,2           ;display character function
          INT    21H           ;DOS call
          POP    AX            ;get number back
          LOOP   NEXT          ;and repeat
          RET
DISPBIN   ENDP

          END

```

READCOM1 executes until any key on the keyboard is pressed. It might be interesting to watch the values returned by a mouse connected to COM1. What patterns are received when the mouse moves left, right, up, and down? What happens when a mouse button is pressed?

Be sure to initialize COM1 to the proper baud rate before using READCOM1. This is easily accomplished with the use of the MODE command. For example, to initialize COM1 for 2,400 baud, no parity, 8 data bits, and 1 stop bit, use this command:

```
C> MODE COM1:2400,N,8,1
```

This is especially important when using COM1 to communicate with the single-board computer of Chapter 12.

## 10.9 TROUBLESHOOTING TECHNIQUES

Finding the cause of a faulty I/O device can be tricky. Here are suggestions of some things to try when you encounter an I/O problem.

- Write a short loop that continually accesses the I/O device. This should allow you to use an oscilloscope to look for a stream of pulses on the output of the address decoder. Use something like this to test an 8-bit output port:

```

MOV    DX,<I/O address>
SUB    AL,AL
PTEST: OUT    DX,AL
        INC    AL
        JMP    PTEST

```

In addition to the steady stream of pulses that should appear on the address decoder output, there should be a binary count appearing at the output port. It is easy to see with an oscilloscope whether the waveform periods double (or halve) as you step from bit to bit. This is a good way to check for stuck or crossed outputs.

When checking an input port, use these instructions (or something similar):

```

MOV    DX,<I/O address>
PTEST: IN     AL,DX
        JMP    PTEST

```

This loop is good for checking the operation of the address decoder. If possible, combine both loops so that data read from the input port is echoed to the output port.

- Check for easily overlooked mistakes, such as using AD<sub>0</sub> through AD<sub>7</sub> to connect to the I/O device, but not using IO/M in the address decoder.
- Verify that the enable signals on the I/O device all go to their active states when accessed.
- For a serial device, examine the serial output for activity. Check for valid transmitter and receiver clocks. If the serial device is connected to a keyboard, press the keys and watch the serial input of the device. Make sure the TTL-to-RS232 driver is working correctly.

Other I/O devices may require you to test the interrupt system, or write a special initialization code to program a peripheral. Keep track of the new software and hardware designs you develop or troubleshoot; this will save you time and effort in the future.

## SUMMARY

In this chapter we examined the design and operation of input and output ports. We saw that the 8088 has a smaller port addressing space than memory space. Two types of port addresses (and associated I/O instructions) may be used. One set of port addresses is 1 byte wide, from 00 to FF. The second set of port addresses, which is 2 bytes wide, ranges from 0000 to FFFF and must be placed into register DX before use.

The hardware details of full and partial port address decoders were covered, along with examples of actual input and output ports. These ports were expanded into applications

dealing with controlled time delays and waveform generation through a digital-to-analog converter. The programming and interfacing requirements of two I/O-based peripherals, the 8255 PPI and 8251 UART, were also covered. Additional peripherals for the 8088 will be covered in the next chapter.

## STUDY QUESTIONS

1. Show two ways of reading input port 40H. What instructions are required?
2. Explain the processor bus activity when OUT 20, AL executes. Assume a minmode system.
3. Design a minmode full port address decoder for input/output port B0. You must generate active-low  $\overline{RDB0}$  and  $\overline{WRB0}$  signals.
4. Design a minmode full port address decoder for input ports B0 through B7. Eight individual port select outputs should be generated.
5. What changes must be made to the designs of Questions 3 and 4 for a maxmode system?
6. Design a minmode partial-address decoder for an output port whose binary address is 10X01X1X1111111, where X is a *don't care* bit. What are all possible port address ranges for this decoder?
7. Use the magnitude comparators of Figure 10.5 to design a partial-address decoder with selectable port ranges. The base port address is 1011CCCCCCCC0100, where C represents the 8-bit number matched by the comparators.
8. What is the first selectable port address in Question 7? What is the last? How far apart is each port address (in locations)?
9. What are the decoded port address ranges for each circuit in Figure 10.24?
10. Why are input ports buffered rather than latched?
11. Modify the design of Figure 10.7 so that the port address is 409C.
12. Modify the design of Figure 10.7 so that the port is 16 bits wide (lower 8 at port 9C and upper 8 at 9D).
13. Repeat Questions 11 and 12 for the output port in Figure 10.8.
14. Modify the BINCNT program so that the lower 7 bits of the input port adjust the counting speed, while the most significant bit controls the direction, up or down, of the count.
15. Write a program called KATERPILAR, which will output a sequence of rotating bit patterns to the LEDs of Figure 10.9. The sequence might look like this:

```

XXX-----
-XXX-----
--XXX---
---XXX--
----XXX-
-----XXX

```

The sequence should change direction when all LEDs on the right are lit. Use the input port to control the speed of the display.

16. Modify BINCNT and the design of Figure 10.9 to allow a 16-bit counter to be displayed.
17. The data table in the WAVER routine does not have to be 256 bytes long to get decent looking waveforms. What frequency is possible if only 10 bytes are used to generate a waveform? What about 50 or 100 bytes?

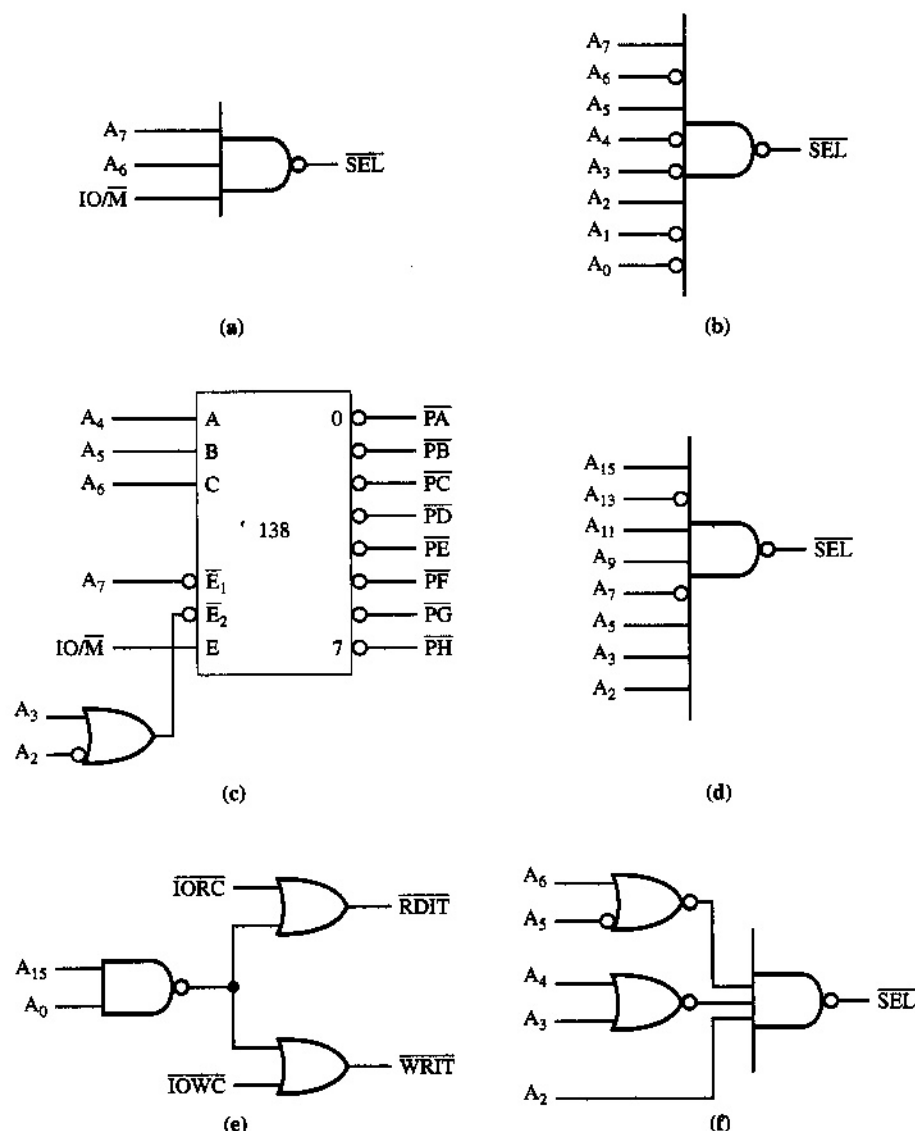


FIGURE 10.24 For Question 9

18. Write a subroutine called TONES that generates two different frequency square waves and alternates them. TONES should output 100 cycles of the low-frequency signal and 50 cycles of the high-frequency signal. This should be repeated ten times before TONES returns. The high-frequency signal should have four times the frequency of the low-frequency one.
19. Redesign the port address decoder of Figure 10.11 so that port A is at address 8A. What are the other three port addresses?
20. Determine the mode words for each 8255 configuration:
  - (a) Mode 0, Ain, Bout, Cin
  - (b) Mode 1, Aout, Bin
  - (c) Mode 2, Bin (Mode 0)

21. Redo the BINCNT program by adapting it for use with an 8255 with a base address of 48H. Assign ports A, B, or C as you like. Draw a schematic of your design.
22. Use the VCON procedure to assist you in writing a subroutine called ONEWAVE, which will do the following:
  - (a) Wait for the analog signal to cross 0 (80H).
  - (b) Read and store samples from VCON until the signal crosses 0 twice.
 The signal samples should be stored in a data table pointed to by register DI. Do not write more than 1,024 samples into the table.
23. Rewrite VCON so that the  $\overline{RD}$  input of the 0804 is always 0, and the  $\overline{INT}$  output is fed back to the  $\overline{WR}$  input. Note that only a software connection will exist between  $\overline{INT}$  and  $\overline{WR}$ .
24. Sketch the 11-bit transmission code for an ASCII "K." Use even parity and 2 stop bits.
25. How long does the character in Question 24 take to transmit if the BAUD rate is 1,200?
26. Use partial-address decoding to put the 8251 of Figure 10.19 into port space at address 40H.
27. Show the instructions needed to program the 8251 for:
  - (a) Asynchronous mode, 7 data bits, odd parity, X16 clock, 2 stop bits.
  - (b) Asynchronous mode, 8 data bits, no parity, X16 clock, 1 stop bit.
  - (c) Synchronous mode, 7 data bits per character.
28. Modify the CHARIN and CHAROUT routines so that bit shift/rotate instructions are used in place of AND to check the status.
29. Modify the CHAROUT routine so that any ASCII code between 01 and 1A will be displayed as a two-character sequence. The first character will always be "^." The second character is found by adding 40H to the original character byte. For example, if the character to display is 03H, CHAROUT should output "^C." All other codes should be directly output.
30. Modify the TVT program so that all characters including the carriage return are stored in a buffer pointed to by register BP. The length of the buffer (not to exceed 255 characters) is returned in CL. When carriage return is hit, or when the buffer is full, call the far routine SCANBUFF.
31. Modify the buffered TVT program so that it detects special control characters. The first is the backspace character (ASCII 08H). If a backspace is received, back up one position in the buffer. Next is Control-C (03H). Reset BP to the beginning of the buffer when this character is received. Last is Control-R (12H). This code causes the entire contents of the current buffer to be displayed on a new line.
32. Write a program that sends the message
 

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

to the COM1 serial port, until "x" is entered on the keyboard.



---

# CHAPTER 11

---

## Interfacing with the 80x86

---

### OBJECTIVES

In this chapter you will learn about:

- Hardware interrupt handling
- Programmable time delays using an interval counter
- Floating-point coprocessor functions
- Interfacing with the Personal Computer

### KEY TERMS

Floating-point unit  
Initialization command  
word  
Interrupt-driven I/O

Interrupt request register  
Operation command word  
Polled I/O  
Priority resolver

Programmable interrupt  
controller  
Programmable interval timer

---

### 11.1 INTRODUCTION

The power of a microprocessor can be increased by the use of peripherals designed to implement special functions, functions that may be very difficult to implement via software. A good example of this principle would be in the use of a coprocessor. The coprocessor comes equipped with the ability to perform complex mathematical tasks, such as logarithms, exponentials, and trigonometry. The 8088, although powerful, would require extensive programming to implement these functions, and even then would not compute the results with the same speed. Thus, we see that there are times when we have to make a hardware/software trade-off. In this chapter we will concentrate on applications that employ the use of standard peripherals, designed specifically for the 8088. In each case, we will examine the interfacing requirements of the peripheral, and then see how software is used to control it.

You are encouraged to refer to the datasheets available on the companion CD as you read the chapter to get a more detailed look at each peripheral. Section 11.2 covers the 8259 programmable interrupt controller. Section 11.3 shows how accurate time delays can be generated with the 8254 programmable interval timer. Section 11.4 explains the operation of the floating-point unit (80x87 coprocessor). Section 11.5 provides several interfacing examples for the personal computer. Troubleshooting tips for new peripherals and devices are presented in Section 11.6.

---

## 11.2 THE 8259 PROGRAMMABLE INTERRUPT CONTROLLER

We saw in the previous chapter that a microprocessor must be interfaced with an I/O device to communicate with the outside world. Software support is required for each I/O device to ensure its proper operation. For example, the receiver status of a UART must be frequently examined to ensure that no received characters are lost. If a loop is used to test the receiver status, the processor may end up spending a great deal of time waiting for the chance to send the next character. While it is doing this, it cannot do anything else! An efficient solution to this situation is accomplished by adding an interrupt signal to the processor. Whenever a character is received by the UART, the UART will interrupt the processor. A special interrupt service routine will be used to read the UART and process the new character. When the UART is interfaced in this way, the processor is free to execute other codes during the times when the UART has not yet received a character. In this interrupt-driven I/O scheme, the processor accesses the UART only when it has to. This example illustrates the basic differences between **polled I/O** and **interrupt-driven I/O**. For some systems, polling is a good solution. This is especially true when the system is dedicated to doing one task over and over again. When a system is used in a more general way, the processor cannot afford to spend its time constantly polling each I/O device. In this case, interrupts provide a simple way to service all peripherals only when they need the processor's attention.

If we expand the idea of interrupt-driven I/O to an entire system, the number of interrupts required quickly adds up. Separate interrupts may be used for real-time clock/calendars, floppy and hard disk drives, the computer's keyboard, serial and parallel interfaces, video displays, and many other devices. Each device will require its own interrupt handler. The interrupts may also be assigned individual priorities to ensure that they get serviced in a manner desired by the programmer. How is it possible to use this many interrupts on the 8088, which has only two external hardware interrupt inputs (NMI and INTR)? The answer is the **8259 programmable interrupt controller**, a special peripheral designed to support eight levels of prioritized hardware interrupts. The 8259 is considered an I/O device on the system bus and can be written and read like an I/O port. For systems requiring more than eight levels of prioritized interrupts, it is possible to cascade 8259s to obtain up to 64 levels of interrupts. The 8259 is configured after power-on through software and may be reconfigured at any time. The 8259 is designed to interface with the early 8-bit machines (the 8080 and 8085) and also the 16-bit 8088 and 8086. We will examine only its operation with respect to the 8088.

Figure 11.1 shows a simplified block diagram of the 8259. Eight levels of hardware interrupts are provided by inputs  $IR_0$  to  $IR_7$ . These eight inputs go directly to the **interrupt request register (IRR)**. This register keeps track of what interrupts have requested service. The 8259 can be programmed to allow level-sensitive or edge-sensitive interrupt inputs.

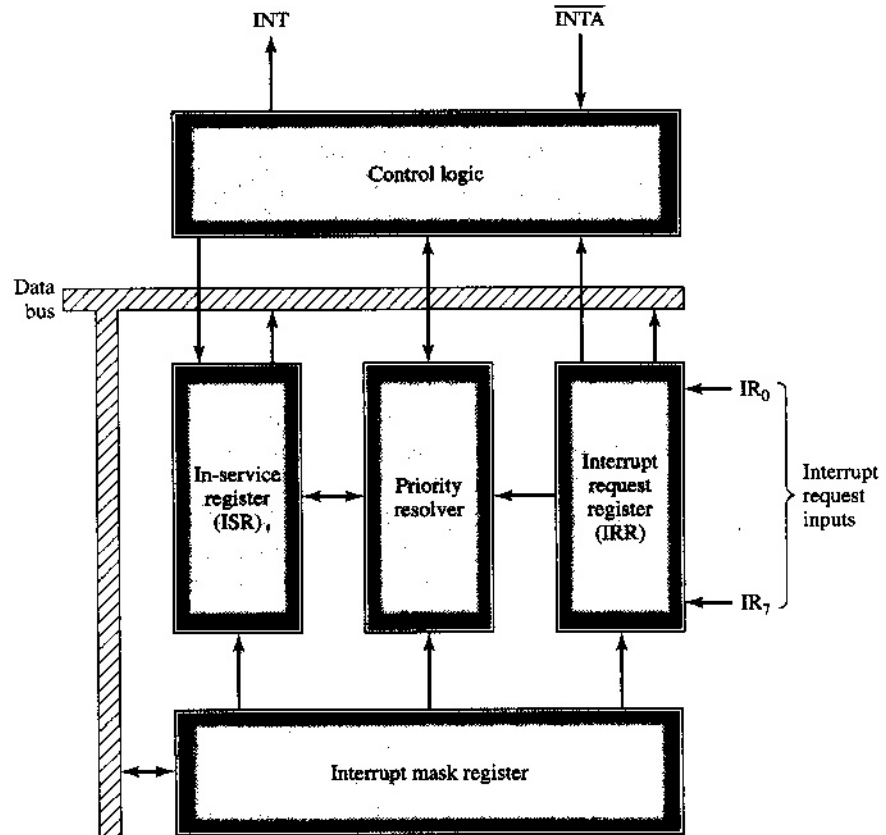


FIGURE 11.1 8259 block diagram

The output of the IRR feeds the priority resolver, which selects the highest priority interrupt from those requesting service. For example, if  $IR_2$  and  $IR_5$  both request service simultaneously,  $IR_5$  will be selected first by the priority resolver.

The output of the priority resolver is used by the *in-service register* (ISR). The ISR indicates which interrupts are being serviced.

All three of these registers communicate with the control logic section, which performs the handshaking with the processor's  $INTR$  and  $INTA$  signals.

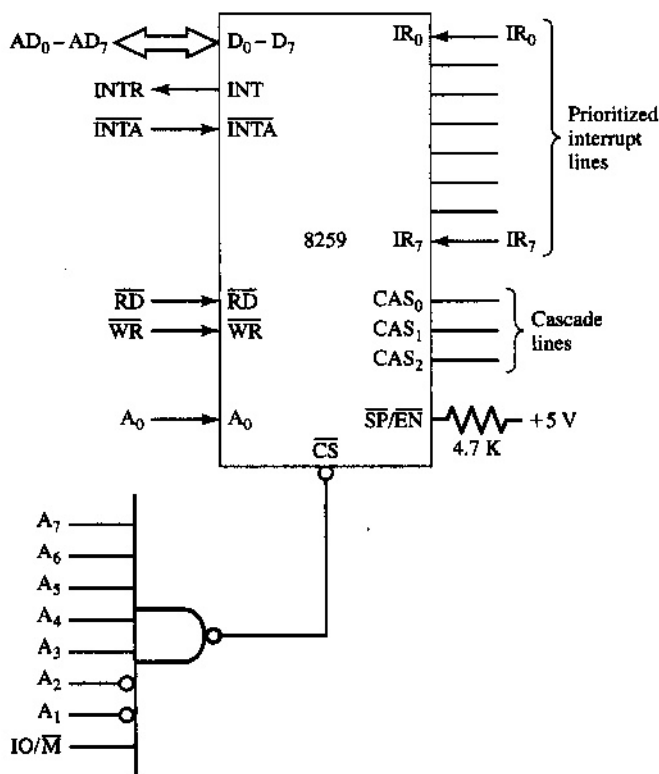
Individual interrupts may be disabled by data written to the *interrupt mask register*. Any combination of interrupts can be disabled (or *masked*) without affecting the priority of the remaining interrupts.

The 8259 can be programmed by writing the appropriate data values to a set of internal registers that are used to control the operation of each of the 8259's functional blocks. Before we see how this is done, let us examine how the 8259 interfaces with the processor.

### Interfacing the 8259

The 8259 connects to the system bus like an ordinary I/O device. Figure 11.2 shows an example of the 8259 mapped to I/O ports F8 and F9 in a minmode system. A port-address

**FIGURE 11.2** 8259 interfaced to the 8088 in minimum mode



decoder using an eight-input NAND gate is used to detect port addresses F8 and F9. The processor's  $A_0$  address line connects directly to the 8259 to select internal registers, along with  $\overline{RD}$  and  $\overline{WR}$ . The 8259 takes over the use of the processor's INTR input, which it activates during an interrupt request. Up to eight devices may request an interrupt from the 8259, via interrupt inputs  $IR_0$  through  $IR_7$ . These eight lines may be programmed for edge-sensitive or level-sensitive operation, and are prioritized, with  $IR_7$  having the highest priority.

Three cascade lines,  $CAS_0$  through  $CAS_2$ , are used to expand the 8259 in a system requiring more than eight levels of interrupts. These signals may be left unconnected when there is a single 8259 in a system. Multiple 8259s require  $CAS_0$  through  $CAS_2$  to be connected in parallel.

Finally, the  $\overline{SP/EN}$  line is used to select master/slave operation in the 8259. When a single 8259 is used, it must be operated as a master. This can be selected by placing a 1 on  $\overline{SP/EN}$ . When two or more 8259s are used, only one 8259 may be a master. The remaining 8259s must be operated as slaves (with  $\overline{SP/EN}$  grounded).

The operation of a properly programmed 8259 is as follows:

1. One or more of the interrupting devices connected to the 8259 requests an interrupt by activating the appropriate IR input. The corresponding bit in the IRR is set.
2. The 8259 examines the interrupts requested and issues an INTR signal to the 8088 for the highest priority active interrupt input.
3. The processor responds with a pulse on  $\overline{INTA}$ .

|   |   |   |   |      |   |      |     |
|---|---|---|---|------|---|------|-----|
| 7 | 6 | 5 | 4 | 3    | 2 | 1    | 0   |
| 0 | 0 | 0 | 1 | LTIM | 0 | SNGL | IC4 |

FIGURE 11.3 Initialization command word 1

4. The 8259 sets the bit for the highest priority active interrupt in the in-service register and clears the corresponding bit in the interrupt request register (to remove the request).
5. The 8088 outputs a second pulse on  $\overline{INTA}$  (as it usually does with *any* interrupt caused by INTR).
6. The 8259 outputs an 8-bit interrupt vector number on the data bus. This number is captured by the processor and used to select the corresponding interrupt service routine address from the vector table.

This process illustrates the usefulness of the 8259. Individual vector numbers are assigned to each of the IR inputs connected to a device. When a particular device issues an interrupt request to the 8259, the 8259 sends the corresponding vector number to the processor.

In the next section we will see how a single 8259 is programmed to provide these vector numbers to the CPU.

## Programming the 8259

The port-address decoder shown in Figure 11.2 maps the 8259 to ports F8 and F9. These port addresses must be used to access the 8259 during initialization and also during normal operation. It is important to note that the 8259 can be reconfigured whenever necessary by issuing initialization commands to it. This may be useful to some designers who may wish to modify the interrupt mechanism of their system as necessary.

When a single 8259 is used in a system, operating in master mode, it requires two types of control information from the CPU: **initialization command words (ICW)** and **operation command words (OCW)**. Once these words are written to the 8259, it is capable of servicing its eight levels of prioritized interrupts. If more than one 8259 is used, each must be programmed individually.

Initialization is accomplished by sending a sequence of 2–4 bytes (ICWs) to the 8259. Figure 11.3 shows the makeup of the first ICW. Zeros are inserted in bit positions that control the 8259's operation in an 8085 system (bits 2 and 5–7). For the purposes of this discussion, only the bits that directly affect the operation of the 8259 in an 8088 system are explained. IC4 (bit 0), when set, indicates that ICW4 must be read during initialization.

SNGL (bit 1), when set, indicates that only one 8259 is being used in a system. To allow cascaded 8259s, this bit must be cleared (and ICW3 issued).

LTIM (bit 3), when set, indicates that the 8259 should operate the IR inputs in level-sensitive mode. When LTIM is cleared, the eight IR inputs will be edge-sensitive.

### ■ EXAMPLE 11.1 What is the interpretation of ICW1 when it contains 1AH?

**Solution:** The bit pattern produced by 1AH is 00011010. This indicates that both LTIM and SNGL are set, and IC4 is cleared. This will select level-sensitive inputs and inform the 8259 that it is the only interrupt controller in the system, and that ICW4 is not required. ■

|                |                |                |                |                |   |   |   |
|----------------|----------------|----------------|----------------|----------------|---|---|---|
| 7              | 6              | 5              | 4              | 3              | 2 | 1 | 0 |
| T <sub>7</sub> | T <sub>6</sub> | T <sub>5</sub> | T <sub>4</sub> | T <sub>3</sub> | 0 | 0 | 0 |

**FIGURE 11.4** Initialization command word 2

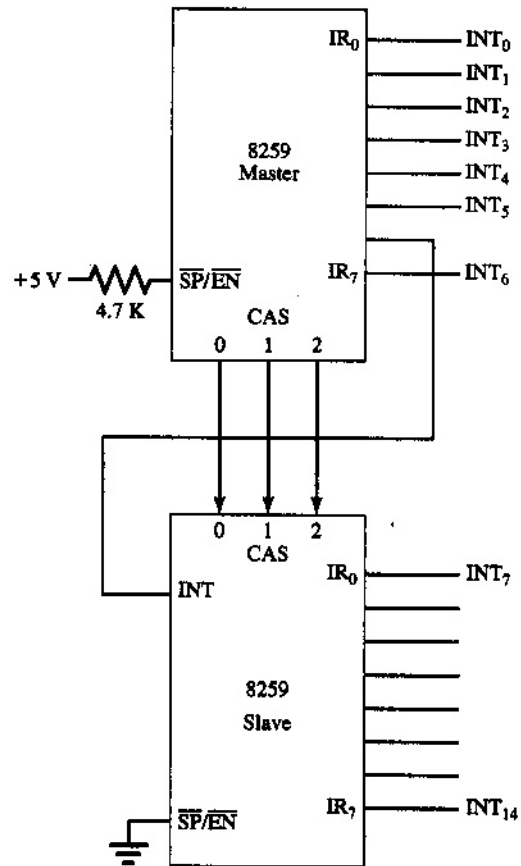
Using the circuit of Figure 11.2 as an example, ICW1 must be output to port F8 ( $A_0$  equals 0) to be properly received. The 8259 recognizes ICW1 by the 1 seen in bit position 4. Reception of this control word also causes the 8259 to perform some internal housekeeping.

Once ICW1 is processed, the 8259 awaits ICW2, which must be output to port F9 ( $A_0$  equals 1). This control word is shown in Figure 11.4, and is used to program the eight interrupt vector numbers that will be associated with the IR inputs.  $T_3$  through  $T_7$  become the five most significant bits of the vector number supplied by the 8259 during an interrupt acknowledge cycle. The lower 3 bits are generated by the IR input that has been selected for service. Figure 11.5 shows the makeup of the interrupt vector number. Do you see the relationship between bits 0–2 and the corresponding IR signal? If  $IR_0$  requests service, the vector number generated will contain three 0s in the lower bits. Suppose that ICW2 is issued with the data byte 68. This indicates that  $T_7$  through  $T_3$  are assigned the values 01101. What are the eight possible interrupt vector numbers, using the information from Figure 11.5?  $IR_7$  will create vector number 6F.  $IR_0$  will create vector number 68. Vector numbers 69 through 6E are generated by  $IR_1$  through  $IR_6$ , respectively.

If a single 8259 is used in a system, ICW3 is not needed. For multiple 8259s, ICW3 serves two functions. When an 8259 is used as a master, each bit in ICW3 is used to indicate a slave connected to an IR input. When the 8259 is used as a slave, only the lower 3 bits of ICW3 are used, and they set the slave's cascade number (0 to 7). Figure 11.6 shows two 8259s connected as master and slave. The INT output on the slave device connects to the  $IR_6$  input on the master device. Thus, any requests on interrupt lines  $INT_7$  through  $INT_{14}$  will cause  $IR_6$  to be activated on the master. The master will then examine bit 6 in ICW3 to see if it is set (indicating that a slave is connected to  $IR_6$ ). If so, it will output the cascade number of the slave (110 in this case) on  $CAS_0$  through  $CAS_2$ . These cascade bits are received by the slave device, which examines its ICW3 to see if there is a match. The programmer must have previously programmed 110 into the slave's ICW3. If there is a match between the cascade number and ICW3, the slave device will output the appropriate vector number during the second  $\overline{INTA}$  pulse.

**FIGURE 11.5** Generating the interrupt vector number

|        | Interrupt      |                |                |                |                | Vector number  |                |                |
|--------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
|        | D <sub>7</sub> | T <sub>7</sub> | T <sub>6</sub> | T <sub>5</sub> | T <sub>4</sub> | T <sub>3</sub> | D <sub>1</sub> | D <sub>0</sub> |
| $IR_7$ |                |                |                |                |                |                | 1              | 1              |
| $IR_6$ |                |                |                |                |                |                | 1              | 0              |
| $IR_5$ |                |                |                |                |                |                | 1              | 0              |
| $IR_4$ |                |                |                |                |                |                | 0              | 1              |
| $IR_3$ |                |                |                |                |                |                | 0              | 1              |
| $IR_2$ |                |                |                |                |                |                | 0              | 0              |
| $IR_1$ |                |                |                |                |                |                | 0              | 0              |
| $IR_0$ |                |                |                |                |                |                | 0              | 0              |

**FIGURE 11.6** Two 8259s cascaded

To get this scheme to work, the first 8259 must have bit 6 set in its ICW3, and the second 8259 must have the bit pattern 110 in the lower 3 bits of its ICW3. Both forms of ICW3 are shown in Figure 11.7.

■ **EXAMPLE 11.2** How many slave devices are required if ICW3 in a master 8259 contains 10010010? What IR inputs are connected to each slave device?

**Solution:** Because ICW3 contains three 1s, three slave devices are being used. The INT outputs on each slave connect to IR inputs IR<sub>7</sub>, IR<sub>4</sub>, and IR<sub>1</sub> on the master. Each of the three slaves must have its own ICW3 programmed with a unique cascade code. The three codes are 111 (for IR<sub>7</sub>), 100, and 001. All of the slave devices will receive the same cascade code from the master, so using unique codes for each slave will guarantee that only one slave responds during the interrupt acknowledge cycle.

ICW3 must be output to port F9 (A<sub>0</sub> equals 1) to be properly received. ■

The last initialization command word is ICW4 (also output to port F9). Remember that this word is only needed when bit 0 of ICW1 is set. The format of ICW4 is shown in

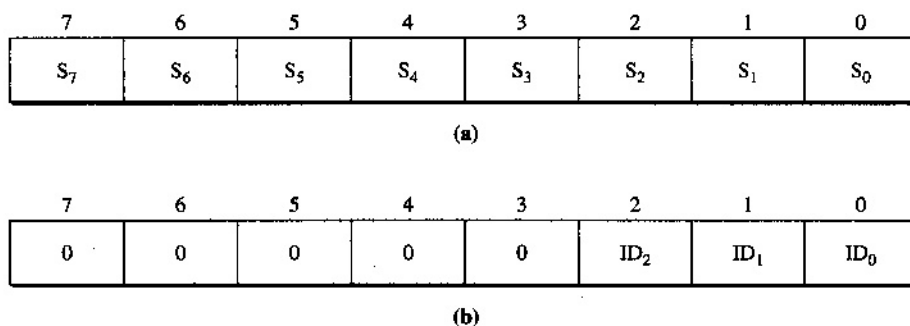
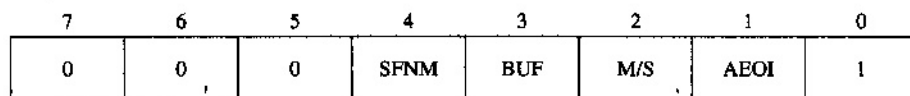
**FIGURE 11.7** ICW3 format: (a) master mode; (b) slave mode**FIGURE 11.8** Format of ICW4

Figure 11.8. The operation of each bit is as follows:

- AEOI (bit 1) is used to program automatic end-of-interrupt mode when high. When AEOI mode is set, the 8259 automatically clears the selected bit in the in-service register. When not operated in AEOI mode, the in-service bit must be cleared manually through software.
- M/S (bit 2) is used to set the function of the 8259 when operated in buffered mode. If M/S is set, the 8259 will function as a master. If M/S is cleared, the 8259 will function as a slave.
- BUF (bit 3), when set, selects buffered mode. When the 8259 operates in buffered mode, the  $\overline{SP/EN}$  pin becomes an output that can be used to control data buffers connected to the 8259's data bus. When BUF is cleared,  $\overline{SP/EN}$  is used as an input to determine the function (master/slave) of the 8259.
- SFNM (bit 4), when set, causes the 8259 to operate in special fully nested mode. This mode is used when multiple 8259s are cascaded. The master must operate in special fully nested mode to support prioritized interrupts in each of its slave 8259s.

Let us look at an example of how a single 8259 can be programmed.

**EXAMPLE 11.3** Tell what instructions are needed to program a single 8259 to operate as a master and provide the following features:

1. Edge-sensitive interrupts
2. ICW4 needed
3. A base interrupt vector number of 40H
4. No special fully nested mode
5. No buffered mode
6. AEOI mode enabled



**Solution:** First, the data values for each ICW must be determined. To set SNGL and IC4 and clear LTIM, ICW1 must contain 13H. This takes care of conditions 1 and 2. Placing 40H into ICW2 will program the desired base-interrupt vector of condition 3. This will allow generation of interrupt vectors 40H through 47H. Because SNGL will be set in ICW1, there is no need to write to ICW3 during initialization. The remaining three conditions are met by placing 03H into ICW4.

Initialization can be performed by the instructions shown here:

```
MOV  AL, 13H
OUT  0F8H, AL    ; output ICW1
MOV  AL, 40H
OUT  0F9H, AL    ; output ICW2
MOV  AL, 03H
OUT  0F9H, AL    ; output ICW4
```

A system designed with flexibility in mind may require that the initialization bytes come from a data table placed somewhere in memory. In this case, ICW1's byte could be tested by the initialization routine to see if it needs to output ICW4 (and similar reasoning applies to ICW3). ■

Once the 8259 has received all of its ICWs, it is ready to begin processing interrupt requests. The exact way the 8259 handles each interrupt is programmed with three OCWs. This information must be output to the 8259 after initialization.

The first OCW is used to mask off selected interrupts by altering the bit pattern in the interrupt mask register. Any of the eight inputs can be disabled by setting its corresponding bit in OCW1.

#### ■ EXAMPLE 11.4 What interrupts are disabled by writing 10100001 into OCW1?

**Solution:** Interrupts IR<sub>7</sub>, IR<sub>5</sub>, and IR<sub>0</sub> are masked out by this control word. Interrupt requests on these inputs will be ignored until a different pattern is output to OCW1. ■

The second OCW is illustrated in Figure 11.9. We saw earlier that the AEOI bit in ICW4 is used to enable/disable *AEOI mode*. AEOI mode supports repetitive interrupts of the same priority by automatically resetting bits in the in-service register. It may be desirable for an interrupt service routine to complete before allowing a second interrupt of the same type or level. In this case, AEOI will be set to 0, and the interrupt service routine is responsible for resetting the in-service register bit for a particular interrupt. This can be done by outputting a *specific EOI command* to OCW2. The level of the interrupt being reset must be placed in the lower 3 bits of OCW2. Other features, such as automatic rotation of the interrupt priorities, can also be selected. When many devices in a system have equal priority, rotating interrupt priorities ensures that all devices get serviced in a round-robin fashion.

The third OCW is used to enable/disable *special mask mode*. Remember that OCW1 allows us to mask off individual interrupts. Normally, when an interrupt of a certain priority is processed by the 8259, the in-service register bit for that interrupt is set. This automatically disables additional interrupts of the same and lower level. For example, an IR<sub>3</sub> request will disable further IR<sub>3</sub> through IR<sub>0</sub> requests. In special mask mode, setting a bit in OCW1 will only disable the associated IR input, leaving the lower priority interrupts enabled.

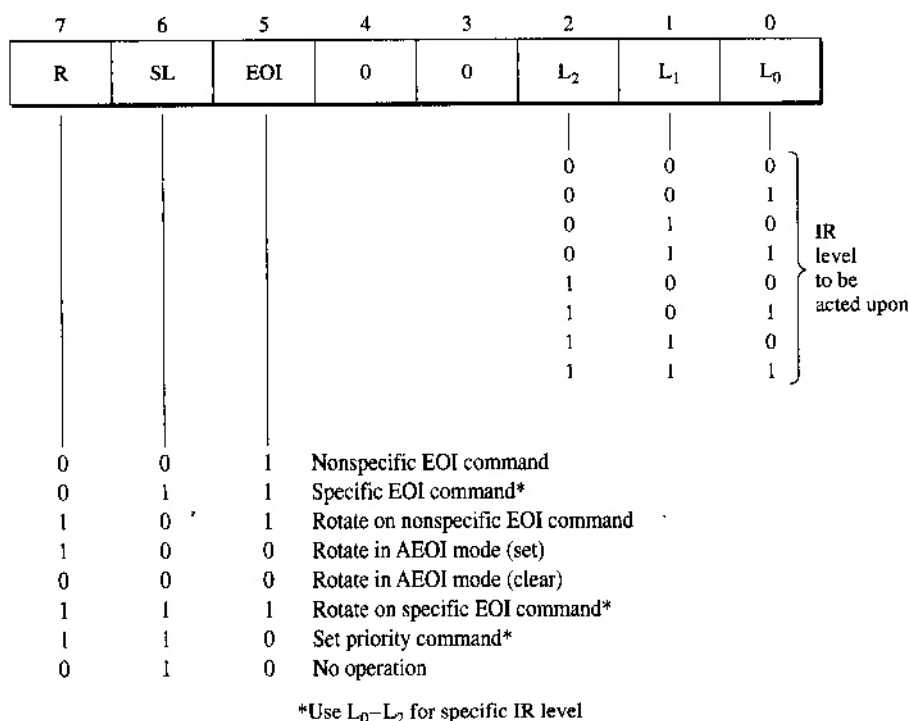


FIGURE 11.9 Format of OCW2

OCW3 is also used to allow the interrupt request and in-service registers to be read by the processor. This is accomplished by writing the necessary pattern into the lower two bits of OCW3 and then reading port F8 (A<sub>0</sub> equals 0).

A final function of OCW3 is to enable/disable *poll mode*. In this mode, the 8259 does not generate interrupt requests on INT. When read, the 8259 will output the level of the highest priority device requesting service on the lower 3 bits of the data bus. The format of OCW3 is shown in Figure 11.10.

There are times when many of the special features provided by the 8259 will not be needed. However, the introduction to this device should show you that a very complex interrupt system, expandable to 64 inputs, is possible with a handful of 8259s and port-select logic.

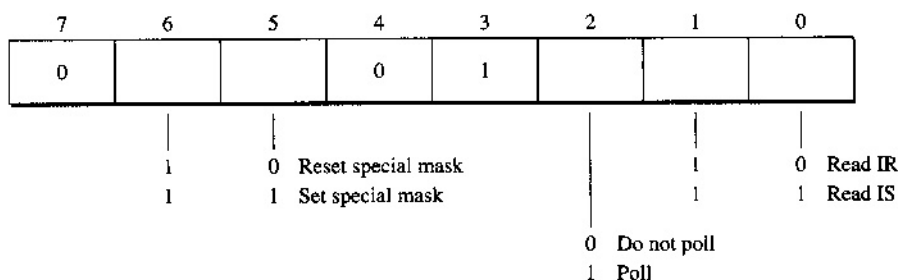


FIGURE 11.10 Format of OCW3

### 11.3 THE 8254 PROGRAMMABLE INTERVAL TIMER

Many applications require the processor to perform an accurate time delay between a set of operations. For example, a microprocessor might be dedicated to reading a custom keypad or driving a multiplexed display. Both applications require a small time delay between repeated input or output operations. A programmer may decide to use a software delay loop, such as:

```
DELAY:  MOV  CX,4000      ;init delay counter
WAIT:   LOOP WAIT        ;LOOP until CX=0
```

The total amount of delay involves 4,000 executions of the LOOP instruction. This time can be estimated by multiplying the processor's clock period by the total number of states required to execute the LOOP instructions. This type of delay loop has two main disadvantages. While the loop is executing, the processor is not able to do anything else, such as execute instructions not related to the loop. Also, the time delay becomes inaccurate if the processor is interrupted. For these reasons some designers (and programmers) prefer to do their timing with hardware. Software is used to program the hardware for a specific time delay. At the end of the time delay, the processor is interrupted. This frees up the processor for other kinds of execution while the hardware is performing the time delay.

A peripheral designed to implement the type of time delay just described (and many others) is the 8254 **programmable interval timer**. Figure 11.11 shows the signal groups of the 8254. The 8254 is interfaced through a group of I/O ports. Three internal counters can be programmed in a variety of formats, including, 4-digit BCD or 16-bit binary counting, square-wave generation, and one-shot operation. These formats allow the 8254 to be used for a number of different timing purposes. A short list of these applications includes real-time clocks (and/or calendars), specific time delay generation, frequency synthesis, frequency measurement, and pulse-width modulation.

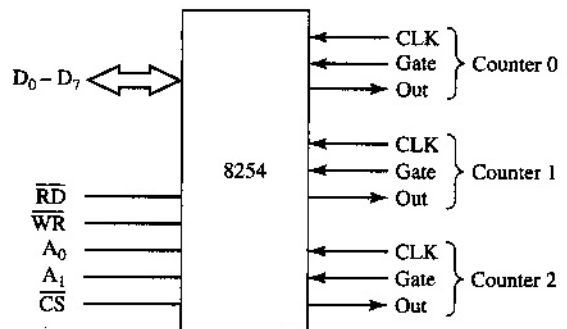
First let us see how the 8254 is interfaced to the 8088.

#### Interfacing the 8254

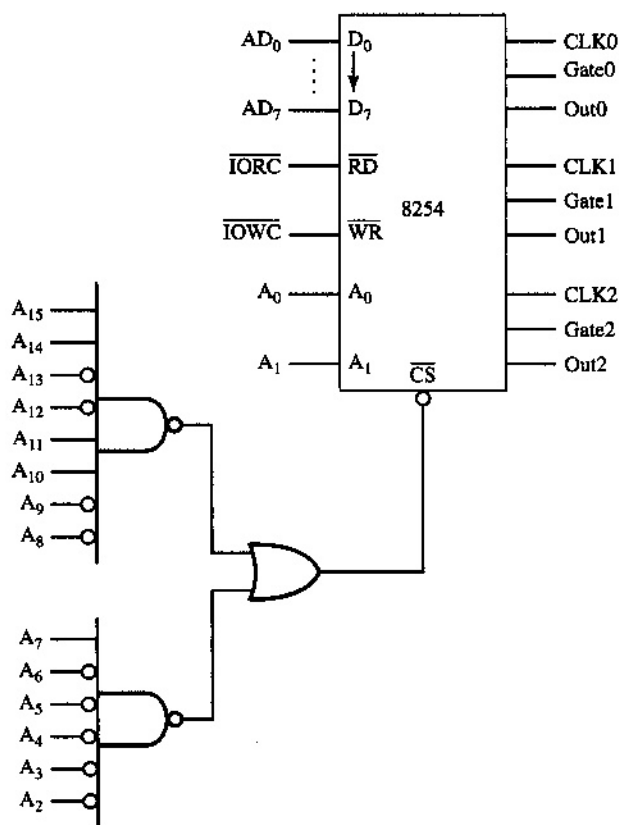
The port-address decoder needed to connect the 8254 to the processor's address bus is shown in Figure 11.12. Here, the 8254 is interfaced with an 8088 operating in maximum mode (indicated by the  $\overline{\text{IORC}}$  and  $\overline{\text{IOWC}}$  command signals).

The two 8-input NAND gates in the port-address decoder map the 8254 into four I/O locations, CC80H through CC83H. The first port (CC80H) is used to read and write counter 0. The second two ports (CC81H and CC82H) access counters 1 and 2, respectively.

**FIGURE 11.11** The 8254 programmable interval timer



**FIGURE 11.12** 8254  
Interfaced to the 8088



The fourth port (CC83H) is used to control the 8254. Each counter contains CLK and GATE inputs, and one output, OUT. Counters may be cascaded by connecting the OUT output of one to the GATE of the other.

### Programming the 8254

Each of the 8254's three counters can be programmed and operated independently of the others. This discussion will concentrate on programming and using counter 0 only.

Counter 0 is a 16-bit synchronous down counter that can be preset to a specific count, and decremented to 0 by pulses on the CLK0 input. Counter 0 may operate in any of six different modes (selected with the use of a control word). Each mode supports four-digit BCD and 16-bit binary counting. Thus, counts may range from 9999 to 0000 BCD or FFFF to 0000 hexadecimal. Programming the counter requires outputting a control word and an initial count to the 8254.

Figure 11.13 shows the bit assignments in the 8254's control word. Two bits (7 and 6) are used to select the counter being programmed. Another two bits (5 and 4) are used to control the way the counter is loaded with a new count. Bits 3, 2, and 1 are used to select one of six modes of operation. The least significant bit is used to select BCD or binary counting.

To begin a counting operation, the control word must be output, followed by the 1- or 2-byte initial count. The initial count value output to the 8254 goes into a *count register*. The count register is cleared when the counter is programmed (upon reception of the control word) and transferred to the actual down counter after it gets loaded with the 1- or 2-byte initial count. The counter may be loaded with a new count at any time, without the need for a new control word.

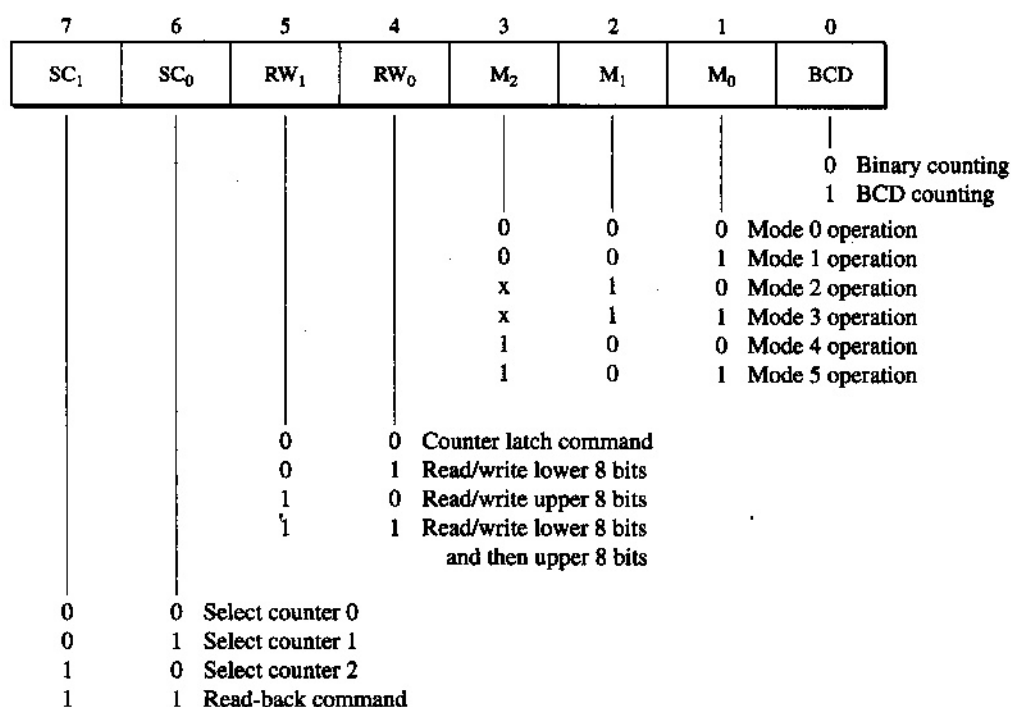


FIGURE 11.13 8254 control word

■ **EXAMPLE 11.5** What instructions are needed to program counter 0 for BCD counting in mode 4? The initial count is 4788.

**Solution:** Using the port addresses assigned by the hardware in Figure 11.12, it is necessary to output the control word to port CC83H and the initial count to port CC80H. The control word needed to program counter 0 for BCD counting in mode 4 and a 16-bit initial count is 00111001 (39H). This value must be output to the control port. The initial count is represented by hex-pairs 47H and 88H.

The counter is initialized by the following instructions:

```
MOV DX,0CC83H
MOV AL,39H
OUT DX,AL      ;output control word
MOV DX,0CC80H
MOV AL,88H
OUT DX,AL      ;output lower 8 bits
MOV AL,47H
OUT DX,AL      ;output upper 8 bits
```

To load a new BCD count at any time, the last five instructions must be repeated. ■

Some time delays may be very short, but still require the use of the 8254. If the counter value is less than 256 for binary counting (or 100 for BCD counting), then the counter may be programmed by outputting the lower 8 bits only.

■ **EXAMPLE 11.6** What control word is needed to program counter 2 for binary counting in mode 1, with an initial count of A0H?

**Solution:** The control word required is 10010010 (92H). Because the count register is cleared when the counter is programmed, writing A0H to the lower half results in an initial count of 00A0H. Notice that the control word indicates that only the lower 8 bits are needed to load the counter.

The counter is initialized by the following instructions:

```
MOV DX,0CC83H
MOV AL,92H
OUT DX,AL          ;output control word
MOV DX,0CC82H
MOV AL,0A0H
OUT DX,AL          ;output initial count ■
```

It may be necessary to read the state of a counter while counting is in progress. The 8254 allows this to be done three different ways.

The first method employs a read from the input port associated with the counter. This technique will not be accurate unless the counter is temporarily paused by stopping the CLK signal or placing a 0 on the GATE input.

The second technique uses the 8254's *counter latch command*. This command is selected by clearing bits 5 and 4 in the control word for a particular counter. This causes the 8254 to transfer a copy of the selected count register into an output latch. The output latch can then be read at any time by reading the desired 8254 port.

■ **EXAMPLE 11.7** What instructions are needed to latch the count in counter 1 and save it in register BX?

**Solution:** The control word necessary to latch counter 1 is 01000000 (40H). This must be output to the control port (CC83H). The latched count may then be read from port CC81H.

These instructions will read the count and save it in register BX:

```
MOV DX,0CC83H
MOV AL,40H
OUT DX,AL          ;latch counter 1
MOV DX,0CC81H
IN AL,DX           ;read lower byte
MOV BL,AL          ;save lower byte
IN AL,DX           ;read upper byte
MOV BH,AL          ;save upper byte ■
```

Note that if the software fails to read the latched counter value before a new counter latch command is issued for the same counter, no new latching takes place. The count will remain latched and unchanged until it is read.

The third technique uses a *read-back command* to read the state of a counter. The read-back command is issued by setting bits 7 and 6 in the control word. The remaining bits take on new meanings, as shown in Figure 11.14. Bits 1, 2, and 3 are used to select any

|   |   |                           |                            |                  |                  |                  |   |
|---|---|---------------------------|----------------------------|------------------|------------------|------------------|---|
| 7 | 6 | 5                         | 4                          | 3                | 2                | 1                | 0 |
| 1 | 1 | $\overline{\text{COUNT}}$ | $\overline{\text{STATUS}}$ | CNT <sub>2</sub> | CNT <sub>1</sub> | CNT <sub>0</sub> | 0 |

FIGURE 11.14 8254 read-back command word

|        |               |                 |                 |                |                |                |     |
|--------|---------------|-----------------|-----------------|----------------|----------------|----------------|-----|
| 7      | 6             | 5               | 4               | 3              | 2              | 1              | 0   |
| OUTPUT | NULL<br>COUNT | RW <sub>1</sub> | RW <sub>0</sub> | M <sub>2</sub> | M <sub>1</sub> | M <sub>0</sub> | BCD |

FIGURE 11.15 Status word format

combination of counters (from none to all three). Bit 4,  $\overline{\text{STATUS}}$ , causes the status to be latched for any selected counter. Bit 5,  $\overline{\text{COUNT}}$ , causes the count to be latched in the same fashion.  $\overline{\text{STATUS}}$  and  $\overline{\text{COUNT}}$  are active low control bits.

■ **EXAMPLE 11.8** What control word is needed to latch the count of counters 0 and 2?

**Solution:**  $\overline{\text{COUNT}}$  must be low to latch the count of any counter. CNT<sub>0</sub> and CNT<sub>2</sub> must be high to select counters 0 and 2. The required control word is 11011010 (DAH). Once this control word is output, counter 0 and counter 2 may be read from ports CC80H and CC82H. ■

Do you see how a single read-back command can be used to eliminate multiple counter latch commands?

When the read-back command is used to read status information, a 1-byte status word is generated for the selected counter. The format of the status word is shown in Figure 11.15. The lower 6 bits indicate the mode and counting scheme currently being used by the counter. NULL COUNT (bit 6) goes low when the new count written to a counter is actually loaded into the counter. This will occur at different times, depending on the mode selected. OUTPUT (bit 7) reflects the current state of the OUT signal for the counter. When counters are assigned and programmed dynamically, it is useful to be able to read the operating parameters by latching the status.

■ **EXAMPLE 11.9** What instructions are necessary to latch the count and status of counter 1? The status must be returned in AH and the count in BX. If the status byte indicates that BCD counting is being used, return 01 in CL, otherwise return with CL = 00.

**Solution:** The command word needed to latch the count and status for counter 1 is 11000100 (C4H). After this byte is output to the control port, the status of counter 1 may be read (from counter 1's input port). Then the 2-byte count may be read.

These instructions will read counter 1's status and count and test for BCD counting:

```
MOV  DX,0CC83H
MOV  AL,0C4H
OUT  DX,AL           ;latch counter 1 data
MOV  DX,0CC81H
IN   AL,DX           ;read status
```

```

MOV  AH,AL
IN   AL,DX      ;read lower byte of count
MOV  BL,AL
IN   AL,DX      ;read upper byte of count
MOV  BH,AL
MOV  CL,AH      ;get status back
AND  CL,01H     ;test BCD bit

```

The AND instruction is used to test the BCD bit in the status byte. Other bits may be tested in a similar way, using different bit patterns in the immediate byte of the AND instruction. ■

The remaining discussion is devoted to an explanation of the six modes of operation possible with the 8254.

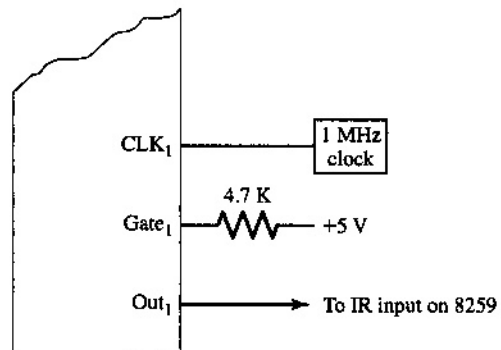
**Mode 0: Interrupt on Terminal Count.** This mode can be used to interrupt the processor after a certain time period has elapsed, or after a number of events have occurred. Its operation is as follows. Initially, the programmer writes the control word specifying mode 0 to the control port of the 8254. This forces the selected counter's OUT signal to go low. Next, the programmer outputs the initial count, which is loaded into the counter during the next falling edge of CLK. Each successive falling edge of CLK will decrement the counter. When the counter gets to 0, OUT will go high. OUT can be used to interrupt the processor (possibly via the 8259 programmable interrupt controller). OUT remains high until a new count (or control word) is issued.

The count can be paused at any time by placing a 0 on the GATE input. Bringing GATE high again resumes counting. If a new count value is output while GATE is low, the next falling edge of CLK will still load it into the counter. When a 2-byte count is output, the first byte output causes the count to terminate and OUT to immediately go low. The second byte output enables the counter to be loaded on the next falling edge of CLK.

■ **EXAMPLE 11.10** Show how the 8254 can be used to generate a time delay of 5 ms. A 1-MHz clock is connected to the CLK input of counter 1.

**Solution:** Figure 11.16 shows the connections to counter 1's input and output signals. GATE is tied high to enable counting all the time. OUT is connected to an IR input on the 8259 programmable interrupt controller. Assuming the IR inputs are edge-sensitive, when OUT goes high at the end of the count, an interrupt will be generated. The trick is to get

**FIGURE 11.16** Generating a 5-ms delay with the 8254





OUT to go high 5 ms after we start the counter. The period of the 1-MHz clock is 1  $\mu$ s. Dividing 5 ms by 1  $\mu$ s gives 5,000, the number of CLK pulses required to get a 5-ms delay. Because one of these CLK pulses will be used to load the counter, the initial counter value must be 4,999. Thus, the counter can operate in binary or BCD mode.

These instructions can be used to implement the 5-ms time delay in mode 0, with BCD counting:

```
MOV  DX,0CC83H
MOV  AL,71H
OUT  DX,AL                ;counter 1, mode 0, BCD
MOV  DX,0CC81H
MOV  AL,99H
OUT  DX,AL                ;lower byte of count
MOV  AL,49H
OUT  DX,AL                ;upper byte of count ■
```

In general, a count of  $N$  requires  $N + 1$  CLK pulses to complete. One CLK pulse is used to load the counter and  $N$  are used to count it down to 0. Also note that an initial BCD count of 0000 will require 10,000 CLK pulses to count down to 0000, and an initial binary count of 0000 will require 65,536!

**Mode 1: Hardware Retriggerable One-Shot.** In this mode, the 8254 is programmed to output a low-level pulse on OUT for a predetermined length of time. The length of the pulse is obtained by multiplying the CLK period by the initial count value. OUT goes high when the control word is written. The 8254 is triggered by transitions on GATE and may be retriggered during counting by a high-going pulse on GATE. Triggering the counter causes OUT to go low while the counter decrements to 0.

■ **EXAMPLE 11.11** Counter 2 is to be programmed to generate a 63- $\mu$ s pulse when triggered. A 1-MHz clock is connected to CLK. What instructions are needed to use counter 2 as a one-shot?

**Solution:** The control word for counter 2 must select mode 1, BCD or binary counting, and a load method. Because only sixty-three CLK pulses will be required for the one-shot pulse, we can get by without having to output a 2-byte count. Only the lower byte value (63 for binary counting, 63H for BCD counting) needs be output.

The required instructions are:

```
MOV  DX,0CC83H
MOV  AL,92H
OUT  DX,AL                ;counter 2, mode 1, binary counting
MOV  DX,0CC82H
MOV  AL,63
OUT  DX,AL                ;output lower byte of count ■
```

**Mode 2: Rate Generator.** This mode of operation is designed to be *periodic*. Instead of generating a single pulse on OUT, this mode generates a pulse on OUT every  $N$  CLK cycles. Thus, the *rate* of output pulses depends on the CLK frequency and the initial count value  $N$ . Mode 2 really simulates a modulo- $N$  counter. A 0 on the GATE input can be used to suspend counting.

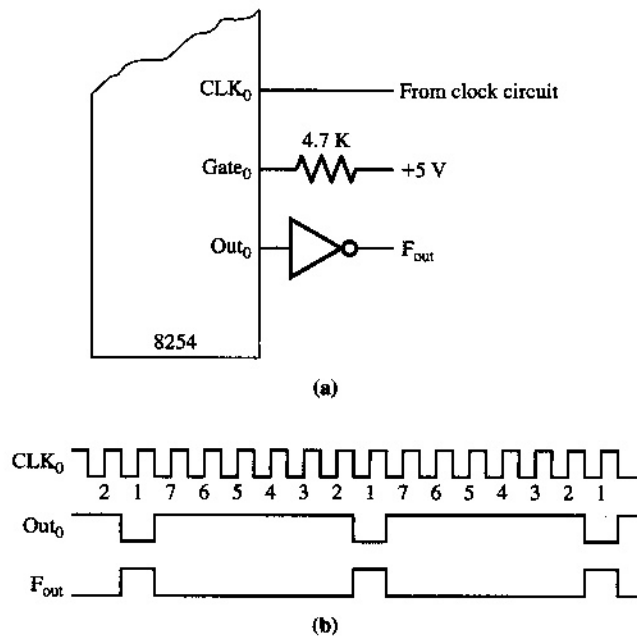
OUT goes high when the control word is written and remains high until the counter decrements to 1. OUT then goes low for one CLK pulse and the counter is reloaded with the initial count. OUT goes high again at the beginning of the next counting cycle.

■ **EXAMPLE 11.12** A programmer needs to generate a waveform that goes high once every seven CLK<sub>0</sub> cycles. What instructions are required? What does the timing diagram for CLK<sub>0</sub> and OUT<sub>0</sub> look like? How is the waveform generated?

**Solution:** Counter 0 must be programmed for mode 2 counting in either binary or BCD. If binary counting is used, these instructions will create the required pulses on OUT<sub>0</sub>:

```
MOV  DX,0CC83H
MOV  AL,14H
OUT  DX,AL           ;counter 0, mode 2, binary counting
MOV  DX,0CC80H
MOV  AL,7
OUT  DX,AL           ;lower byte of count
```

**FIGURE 11.17** Modulo-7 counter: (a) circuit diagram; (b) timing diagram



Mode 2 operation will cause OUT<sub>0</sub> to go low once every seven CLK pulses. An inverter must be used to get the desired output waveform. Figure 11.17 shows the circuit diagram and waveforms for the modulo-7 counter. ■

Because the counters are limited to 16-bit values (either FFFFH or 9999 BCD), what options are available to the programmer who needs to divide by 250,000, or count groups of 10,000 pulses? These kinds of numbers require counters with more bits than those available in the 8254. One simple way to solve this problem is to *cascade two or more counters*. Cascading two counters can result in binary counts of over 4 billion (and BCD counts of up

to 100 million). Cascading counters that are operating in different modes can lead to the creation of some interesting and complex waveforms.

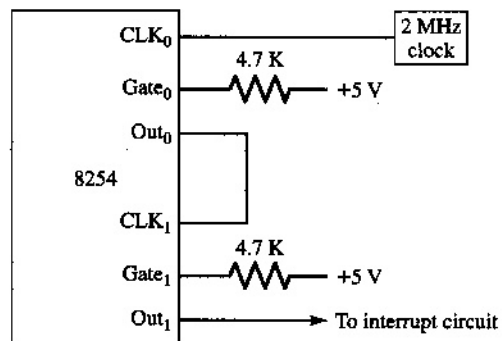
■ **EXAMPLE 11.13** A 2-MHz clock is available for timing in a system that needs to be interrupted once every 4 seconds. How can two counters be cascaded to obtain this interrupt rate?

**Solution:** Figure 11.18 shows how counters 0 and 1 are connected to implement the 0.25-Hz interrupt clock. The 2-MHz clock connected to CLK<sub>0</sub> will create output pulses on Out<sub>0</sub> when counter 0 is programmed in mode 2. These output pulses serve as the clock for counter 1 (also programmed in mode 2). Dividing 2 MHz by 0.25 Hz gives 8,000,000! This is the count that must be simulated by both counters. Many different counting schemes are possible. One scheme requires that counter 0 be loaded with 50,000 and counter 1 with 160. Note that the product of these two numbers is 8,000,000. Counter 0 will output one pulse for every 50,000 CLK<sub>0</sub> pulses. Counter 1 will output one pulse for every 160 CLK<sub>1</sub> pulses.

The instructions needed for this interrupt timing circuit are:

```
MOV DX,0CC83H
MOV AL,34H
OUT DX,AL           ;counter 0, mode 2, binary counting
MOV AL,54H
OUT DX,AL           ;counter 1, mode 2, binary counting
MOV DX,0CC80H
MOV AX,50000
OUT DX,AL           ;output lower byte of count-0
MOV AL,AH
OUT DX,AL           ;output upper byte of count-0
INC DX              ;point to counter 1
MOV AL,160
OUT DX,AL           ;output lower byte of count-1
```

**FIGURE 11.18** Cascading two 8254 counters



Note the order of the OUT instructions. It is possible to output all control words before sending any counter values. This leads to simpler code and some reuse of registers. ■

**Mode 3: Square Wave Mode.** When a 50 percent duty cycle is required in a timing circuit, mode 3 can be used. The operation of mode 3 is similar to mode 2's rate generation in that it is also periodic. The difference lies in the use of the counter.

When the counter is loaded, the 8254 operating in this mode will decrement it by 2 every CLK pulse. When the counter gets to 0, OUT will change state and the counter will

be reloaded. The counter will decrement by 2 again for each CLK pulse. When it reaches 0 a second time, OUT will go back to its original high state. Thus, one complete cycle at OUT requires  $N$  clock pulses:  $N/2$  pulses for the first countdown and  $N/2$  pulses for the second. When  $N$  is an odd number, OUT will be high for  $(N + 1)/2$  CLK pulses and low for  $(N - 1)/2$  CLK pulses.

As always, a low on GATE will disable counting.

**EXAMPLE 11.14** The CLK2 input of the 8254 is connected to a 2.4576-MHz clock (a standard baud-rate generation frequency). OUT2 will be used to drive the transmitter and receiver clock inputs of a UART operating at 2,400 baud with an X16 clock. How must counter 2 be programmed to operate the UART correctly?

**Solution:** A UART operating at 2,400 baud with an X16 clock requires transmitter and receiver clocks of 38.4 kHz. Dividing 2.4576 MHz by 38.4 kHz gives 64. If counter 2 is programmed for mode 3 with an initial count of 64, the correct frequency will be generated.

The instructions for this application are:

```
MOV  DX,0CC83H
MOV  AL,97H
OUT  DX,AL      ;counter 2, mode 3, BCD counting
DEC  DX         ;point to counter 2
MOV  AL,64H
OUT  DX,AL      ;output lower byte of count ■
```

**Mode 4: Software Triggered Strobe.** In this mode of operation, the 8254 generates a low-going pulse on OUT (lasting one CLK pulse) when the counter has decremented to 0. OUT will go low  $N + 1$  CLK pulses after the initial count has been written. The extra CLK pulse is needed to load the counter. Only one pulse will be generated on OUT. To get additional pulses, the counter must be reloaded by outputting the initial count value again.

**EXAMPLE 11.15** A 500-kHz clock is connected to CLK1. The initial count written to counter 1 is 40. How much time expires before OUT1 goes low? How long does OUT1 remain low?

**Solution:** Counter 1 will be loaded with 40 on the first CLK pulse and decremented to 0 over the next forty pulses. It will take a total of forty-one CLK pulses of time before OUT1 goes low. This corresponds to 82  $\mu$ s of time. Because OUT1 will remain low for only one CLK period, its duration is 2  $\mu$ s. ■

**Mode 5: Hardware Triggered Strobe.** This final mode provides the ability for a hardware generated timing pulse. After writing the control word and initial count, OUT will go high. A rising edge on GATE will trigger the 8254 and begin the countdown sequence (after one CLK pulse has been used to load the initial count into the counter). When the counter gets to 0 (after  $N + 1$  CLK pulses), OUT will go low for one CLK pulse. A rising edge on GATE during counting will cause the 8254 to retrigger and begin a new counting sequence (that requires an additional  $N + 1$  pulses to complete).

**EXAMPLE 11.16** An 8254 will be used to generate a strobe pulse on OUT0 fifty CLK cycles after it is triggered by GATE. What instructions are needed to implement this timing need?

**Solution:** Counter 0 in the 8254 must be programmed for mode 5, and either BCD or binary counting. Because the strobe cannot be issued for fifty CLK cycles, it is necessary to use 49 as the initial count.

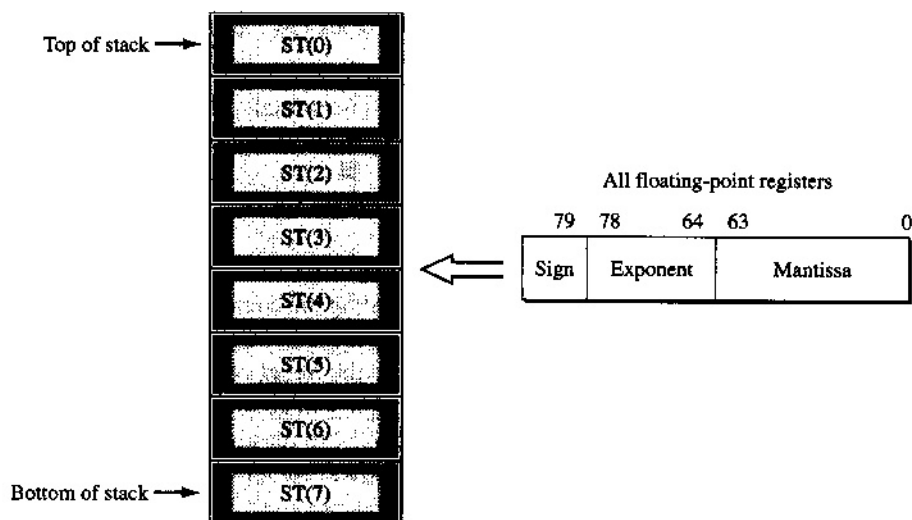
These instructions will program counter 0:

```
MOV  DX,0CC83H
MOV  AL,1AH
OUT  DX,AL      ;counter 0, mode 5, binary counting
MOV  DX,0CC80H
MOV  AL,49
OUT  DX,AL      ;output lower byte of count ■
```

These six modes of operation provide the programmer (or system designer) with many ways of generating timing pulses, delays, and waveforms. Spend a few moments writing down additional timing applications for the 8254. What modes are needed to implement your applications? What is possible with two or more 8254s?

## 11.4 THE FLOATING-POINT UNIT (80x87 COPROCESSOR)

Previously we have covered instructions that perform standard mathematical operations, such as multiplication (MUL and IMUL) and division (DIV and IDIV), but the range of values used with these instructions is limited (often to 64-bit integers). The **FPU (floating-point unit)** improves on this by using 80-bit floating-point numbers, allowing more precision and a huge range of values. A floating-point number, as defined by the IEEE Standards 754 and 854, consists of up to 64 bits of mantissa, a sign bit, and a 15-bit exponent. This 80-bit number does not fit into any of the processor's registers (which are only 16/32 bits wide). For this reason, the FPU contains a set of eight internal floating-point registers. The registers are organized into a stack, with ST(0) referring to the register on top of the stack, and ST(7) to the register on the bottom. Figure 11.19 shows this organization. When data



**FIGURE 11.19** Floating-point register and stack organization in the FPU

is read in from memory, it is *pushed* onto the register stack. Thus, ST(7) becomes ST(6), ST(6) becomes ST(5), and so on, with ST(0) getting replaced by the data from memory. Data written to memory is *popped* off the register stack.

The FPU contains many instructions for manipulating data on the stack. These instructions include the standard add, subtract, multiply, and divide operations (but now with 80-bit precision), as well as logarithms, power functions, square roots, loading useful constants such as pi, and others. The sheer size of the floating-point registers gives the FPU a much higher degree of computational ability over the standard instructions. Computations are performed quickly, because the FPU was specifically designed for floating-point number crunching.

The FPU for the original 8086 and 8088 machines was the 8087. The 80286 and 80386 had their own 80x87 coprocessors as well. Beginning with the 80486, Intel moved the floating-point hardware onto the same chip as the processor, making it internal. There is no need to perform any interfacing anymore. Even so, let us see how the 8087 communicated with the 8088, to get a feel for what goes on between the CPU and the FPU.

### Interfacing the 8087

The 8087 is not interfaced the same as the other peripherals we have examined in this (and earlier) chapters. No port-address decoder is used to enable the 8087. Instead, the 8087 operates in parallel with the 8088, sharing access to the address and data buses, and using a handful of signals for control purposes.

Figure 11.20 shows the connections between the 8088 and the 8087. Note that the 8088 must be operated in maximum mode. This allows the 8087 to use  $\overline{RQ}/\overline{GT}_0$  to initiate a DMA operation. DMA is needed to enable the 8087 to access system memory.

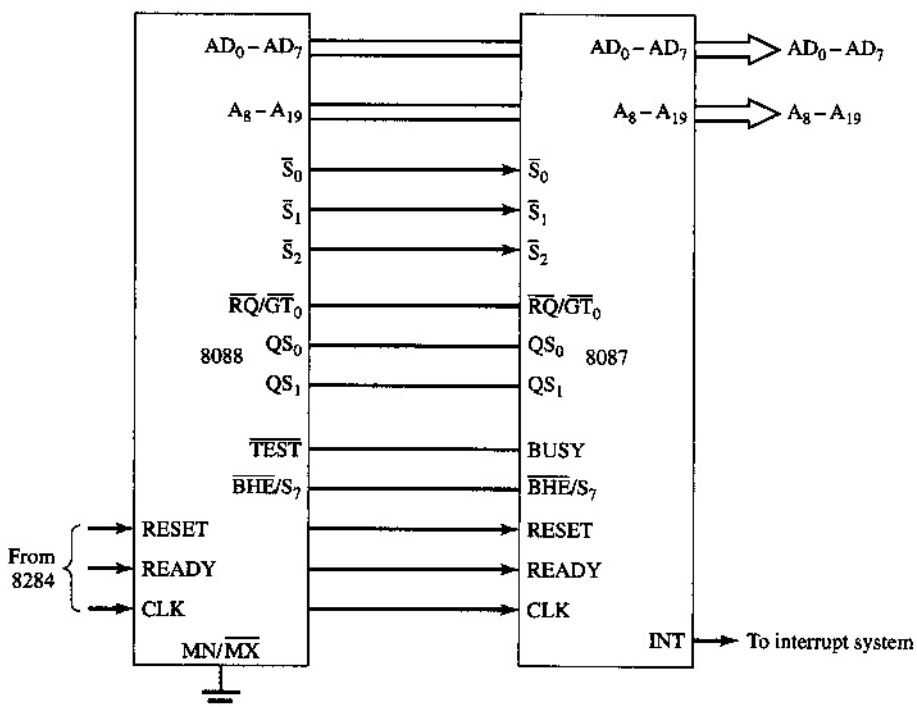


FIGURE 11.20 Connecting the 8087 to the 8088

When coprocessor instructions are encountered on the data bus, the 8087 will capture them and initiate processing. Because they are not part of the 8088's instruction set, the processor will ignore them and proceed with a fetch of the next instruction from memory. If it is necessary to pause the 8088 while the 8087 performs a computation, an *FWAIT* instruction (floating-point WAIT) must be used. This instruction causes the processor to examine the level of its  $\overline{\text{TEST}}$  input. The 8088 will enter a wait state if  $\overline{\text{TEST}}$  is high and remain there until  $\overline{\text{TEST}}$  goes low.  $\overline{\text{TEST}}$  is controlled by the *BUSY* output of the 8087. If the 8087 is in the middle of a computation, *BUSY* will be high, which makes  $\overline{\text{TEST}}$  high. An *FWAIT* instruction will then force the processor into a wait state until the 8087 completes execution.

The 8087 may encounter an unexpected result during computation (such as divide by 0, overflow, or precision errors) and generate an interrupt. The *INT* output must be connected to the system's interrupt circuitry (or directly to *INTR* if available).

With its shared address and data buses, the 8087 is capable of directly accessing memory to read and write floating-point operands.

## FPU Data Types

The FPU implements over seventy different types of floating-point operations on a wide variety of data types. Let us first examine the FPU's data organization, and then proceed to the instruction set.

Seven different data types are available with the FPU. Three of these deal exclusively with positive and negative integers. An integer may be coded as a *word* integer, a *short* integer, or a *long* integer. Word integers are 16 bits long, with the MSB acting as a sign bit. Two's complement notation is used for storage of negative integers.

Short integers are 32 bits long (with the MSB as sign bit) and long integers occupy 64 bits. All bits except the MSB are used to represent the magnitude of the integer. This leads to three different ranges of integers. Figure 11.21 shows the format of the three integer data types and their ranges. Integers can be defined within a source file by using certain assembler directives. The assembler will code numbers in the operand field, into the format used by the FPU.

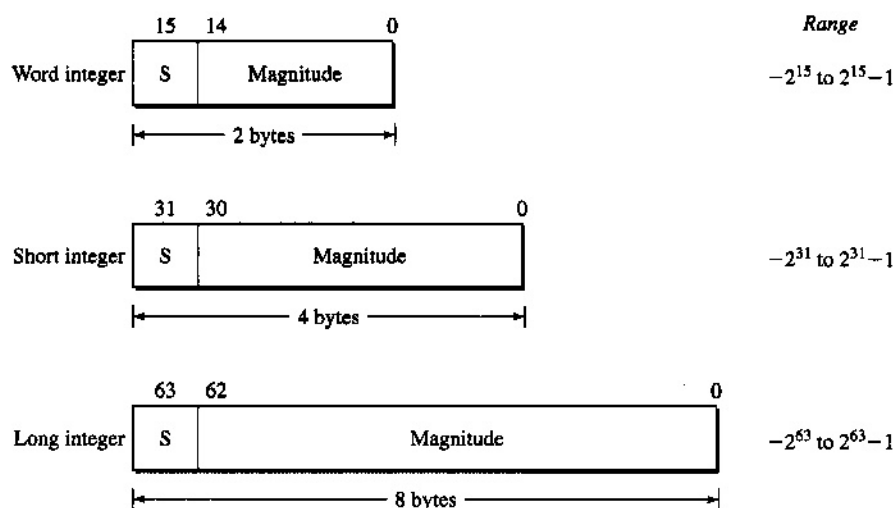


FIGURE 11.21 Integer formats in the FPU

■ **EXAMPLE 11.17** The six source lines shown here indicate how the assembler converts decimal numbers into the three integer formats used by the FPU.

```

03 E8                                WORDINT  DW  1000
FC 18                                DW  -1000
00 6A CF C0                          SHORTINT DD  7000000
FF 95 30 40                          DD  -7000000
00 00 00 6A DE 88 2E 00  LONGINT  DQ  4590000000000
FE FE FE 94 20 76 D1 00  DQ  -4590000000000

```

The DW directive is used for word integers and generates 2 bytes. The DD (define double) and DQ (define quad) directives generate 4 and 8 bytes, respectively. Note the change in the first byte for each group of numbers when the “-” sign is used. ■

The next data type is *packed BCD*. In this format, 10 bytes are used to represent a BCD number. Nine bytes are used to store the magnitude, with each byte holding two BCD digits. The last byte contains only the sign bit. Figure 11.22 shows the organization of this data type. The packed BCD number really represents an integer, with a range roughly equivalent to that of the long integer format.

■ **EXAMPLE 11.18** The packed decimal numbers shown here were created by the assembler in response to the DT (define tens) directive.

```

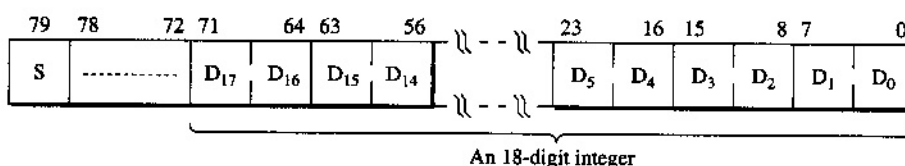
00 00 00 00 00 12 34 56 78 90  PACKBCD  DT  1234567890
80 00 00 00 00 12 34 56 78 90  DT  -1234567890

```

The effect of the minus sign in the second number is easy to spot by examining the most significant byte of both numbers. Also, the packed decimal numbers are right justified within the 10-byte data area. Familiarity with the format of any data type aids in interpreting its value. ■

Three data types are devoted to the use of *real* numbers. *Short* real format (single-precision) uses a 32-bit data block composed of 23 magnitude bits, 8 exponent bits, and 1 sign bit. *Long* real numbers (double-precision) use a 64-bit format composed of 52 magnitude bits, 11 exponent bits, and 1 sign bit. Extended-precision reals have 15 bits of exponent and 64 magnitude bits. These formats are shown in Figure 11.23. A brief introduction to floating-point numbers is needed to fully understand these new formats.

Figure 11.24 gives a general equation for the definition and representation of a short real binary number. The first part of the equation generates the sign of the number. A positive number is generated when the sign bit is 0. Negative numbers require the sign bit to be set.



**FIGURE 11.22** Packed BCD format



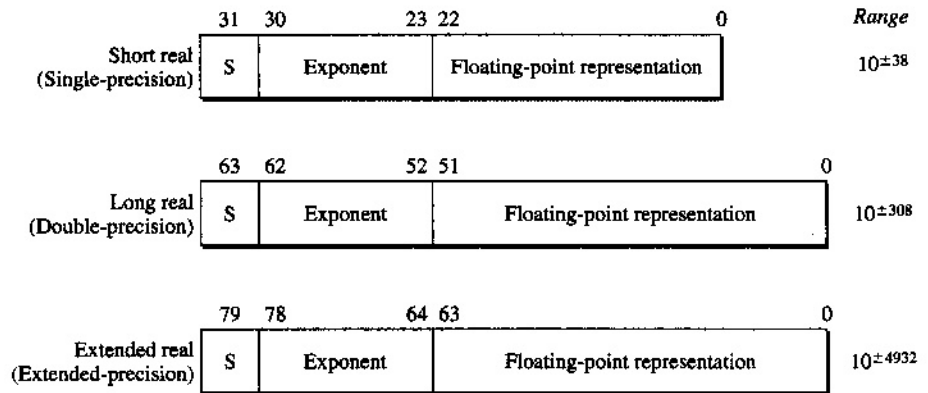


FIGURE 11.23 Short real, long real, and extended real formats

FIGURE 11.24 Equation for short real format representation

$$N = -1^S \times 2^{E-127} \times 1.F$$

Number
Sign
Exponent
Fraction

The exponent part of the number is used as a multiplier for the fractional part. The  $E-127$  term moves the decimal point in the fraction to the left (making the number smaller) or to the right (making the number larger) a certain number of places. The  $E$  variable is controlled by the 8 exponent bits, which are interpreted as an unsigned number in the range 0 to 255. This gives an exponent range for the short real format of  $-126$  to  $+128$ .

The fraction represents the *normalized* binary equivalent of the decimal number. Normalization is a process that converts any binary number into standard format by adjusting the exponent of the number until the mantissa is in the form  $1.F$ . For example,  $1010.1111$  becomes  $1.0101111$  with a power-of-2 exponent of 3. Similarly,  $0.000011011$  becomes  $1.1011$  with an exponent of  $-5$ . Because every number will begin with 1, it is only necessary to store the  $F$  part of the number. In the short real format,  $F$  is 23 bits wide. Thus, a 24-bit number is actually represented via normalization.

■ **EXAMPLE 11.19** What are the steps involved in converting 209.8125 into the normalized short real format?

**Solution:** The first step is to convert 209.8125 into binary. This gives a result of  $11010001.1101$ .

Next, the binary result is normalized by shifting the decimal point seven places to the left. This gives a normalized result of  $1.10100011101$ , with an exponent of  $+7$ . Because 23 bits are used to represent the fraction, we end up with a fraction of  $10100011101000000000000$ . Note that the leading 1 (which is always there) is not stored.

The third step creates the required exponent bits by adding 127 to the exponent obtained during normalization ( $+7$ ). The result is 134, or  $10000110$ .

The final step clears the sign bit so that it represents a positive number. Putting all 32 bits together gives  $01000011\ 01010001\ 11010000\ 00000000$ , or  $43\ 51\ D0\ 00$ . Checking

our conversion with the assembler gives:

```
43 51 D0 00    SHORTREAL    REAL4    209.8125
```

The `REAL4` directive is used to encode a short real number in the format used by the FPU. ■

The same technique that was used in Example 11.19 will work on long real and extended real formats, the only differences being the number of bits used to represent the exponent (11 or 15) and the fraction (52 or 64). Exponents are generated by adding 1,023 or 16,383 to the exponent found during normalization.

A number stored in real format can be converted back into decimal by reversing the steps outlined in Example 11.19. This can be done without much difficulty in software, and you are encouraged to think about how the conversion could be accomplished. Example 11.20 may give you some ideas.

■ **EXAMPLE 11.20** Convert the `REAL4` number `C5 5A 57 00` into its decimal equivalent.

**Solution:** In binary, we have 11000101 01011010 01010111 00000000.

Grouping the data bits into sign, exponent, and fraction bits gives 1 10001010 10110100101011100000000. The 1 in the sign bit indicates a negative result. The 8-bit pattern in the exponent evaluates to 138. To get the exponent of the normalized fraction, we subtract 127. This gives a normalized exponent of 11. The decimal point in the fraction must be moved eleven places to the right.

The fraction (with the leading 1 added) is 1.10110100101011100000000. Multiplying by the normalized exponent results in 110110100101.0111 (with trailing 0s left off). Converting the integer and fractional parts into decimal (and multiplying by  $-1$ ) gives  $-3493.4375$ .

None of the above steps require an extensive amount of software, with rotate or shift instructions most likely implementing the conversion within the fraction. ■

Assembler directives `REAL8` and `REAL10` are used for long real and extended real numbers, respectively.

The instruction set of the FPU consists of a number of functional groups. These groups are the data transfer, arithmetic, compare, transcendental, constant, and processor control instructions. Each group will be introduced briefly.

## FPU Instructions

A wide variety of instructions are available with the FPU. Let us take a brief look at the different groups of instructions used by the FPU.

**Data Transfer Instructions.** These instructions are used to move data around inside the FPU, and between the FPU and system memory. Source and destination operands are not required on many of the FPU's instructions. When no operands are explicitly included in an instruction, the FPU will use the number (or numbers) located on the top of its floating-point stack. Results are written back onto the stack or into memory. Calculations involving repeated loops are best written in a way that uses the stack to avoid endless delays due to the access time of the system memory.

Source and destination operands are indicated by <src> and <dst>, respectively. The data transfer instructions are:

|       |       |                        |
|-------|-------|------------------------|
| FLD   | <src> | ;load real             |
| FILD  | <src> | ;load integer          |
| FBLD  | <src> | ;load BCD              |
| FST   | <dst> | ;store real            |
| FIST  | <dst> | ;store integer         |
| FBSTP | <dst> | ;store BCD and pop     |
| FSTP  | <dst> | ;store real and pop    |
| FISTP | <dst> | ;store integer and pop |
| FXCH  | <dst> | ;exchange registers    |

Note that all floating-point instructions begin with F. This helps to differentiate them from ordinary processor instructions.

**Arithmetic Instructions.** The arithmetic instructions are designed to perform a wide variety of computations on all of the data types supported by the FPU. The hardware performing the operations produces results much faster than an ordinary program could.

The arithmetic instructions are:

|         |              |                                  |
|---------|--------------|----------------------------------|
| FADD    | <dst>, <src> | ;add real                        |
| FADDP   | <dst>, <src> | ;add real and pop                |
| FIADD   | <src>        | ;add integer                     |
| FSUB    | <dst>, <src> | ;subtract real                   |
| FSUBP   | <dst>, <src> | ;subtract real and pop           |
| FSUBR   | <dst>, <src> | ;subtract real reversed          |
| FSUBRP  | <dst>, <src> | ;subtract real reversed and pop  |
| FISUB   | <src>        | ;subtract integer                |
| FISUBR  | <src>        | ;subtract integer reversed       |
| FMUL    | <dst>, <src> | ;multiply real                   |
| FMULP   | <dst>, <src> | ;multiply real and pop           |
| FIMUL   | <src>        | ;multiply integer                |
| FDIV    | <dst>, <src> | ;divide real                     |
| FDIVP   | <dst>, <src> | ;divide real and pop             |
| FDIVR   | <dst>, <src> | ;divide real reversed            |
| FDIVRP  | <dst>, <src> | ;divide real reversed and pop    |
| FIDIV   | <src>        | ;divide integer                  |
| FIDIVR  | <src>        | ;divide integer reversed         |
| FABS    |              | ;absolute value                  |
| FNCHS   |              | ;change sign                     |
| FPREM   |              | ;partial remainder               |
| FPREMI  |              | ;IEEE standard partial remainder |
| FRNDINT |              | ;round to integer                |
| FSCALE  |              | ;scale                           |
| FSQRT   |              | ;square root                     |
| FXTRACT |              | ;extract exponent and fraction   |

Some instructions (like FSQRT and FABS) require no operands and will perform their indicated operation on the number stored on top of the stack. The programming applications that follow will use some of these mathematical operations and show their various operand forms.

**Compare Instructions.** It is useful during a computation to be able to check the results before continuing on to a new set of calculations. The compare instructions allow the programmer to examine the value of the number stored on top of the stack. The compare instructions affect flags stored in the FPU's *status register*. The status register is 16 bits wide and contains a number of flags and status indicators as indicated in Figure 11.25. One

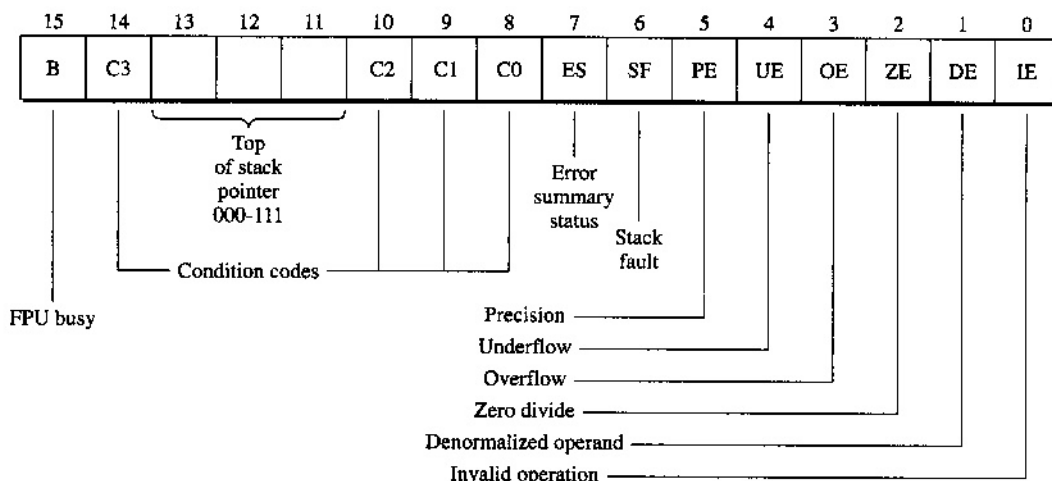


FIGURE 11.25 FPU status register

bit indicates that division by 0 has occurred. Another is set when an exponent overflow is detected. A programmer can look for these conditions by testing specific bits within the status register.

The compare instructions are:

```
FCOM    <src>    ;compare real
FCOMP    <src>    ;compare real and pop
FCOMPP   ;compare real and pop twice
FICOM    <src>    ;compare integer
FICOMP    <src>    ;compare integer and pop
FTST     ;test top of stack (compare with 0.0)
FUCOM    ;unordered compare real
FUCOMP    ;unordered compare real and pop
FUCOMPP   ;unordered compare real and pop twice
FXAM     ;examine top of stack
```

**Transcendental Instructions.** The transcendental instructions use numbers stored on top of the stack (the first and second stack elements) and write their results back onto the stack. These instructions can be combined with the arithmetic instructions to form other functions that are not implemented on the FPU.

The transcendental instructions are:

```
FSIN     ;calculate sine
FCOS     ;calculate cosine
FSINCOS  ;calculate sine and cosine
FPATAN   ;partial arctangent
FPTAN    ;partial tangent
F2XM1    ;calculate  $2^X - 1$ 
FYL2X    ;calculate  $Y * \log_2(X)$ 
FYL2XP1  ;calculate  $Y * \log_2(X + 1)$ 
```

**Constant Instructions.** The constant instructions are used to load the top of the stack with a commonly used number. This helps to speed up execution of routines that would otherwise have to store the constants in memory. All constants are stored with the extended real format.

The constant instructions are:

```
FLDZ      ;load 0.0
FLD1      ;load 1.0
FLDPI     ;load PI (3.14159...)
FLDL2E    ;load log2(E) where E = 2.7182818...
FLDL2T    ;load log2(10)
FLDLG2    ;load log10(2)
FLDLN2    ;load logE(2)
```

Note that some constants are useful for conversion between base-10 and base-2, and also between base-E and base-2.

**Processor Control Instructions.** These instructions are useful for controlling the operation of the FPU. Access to the status and control register is provided, and programmer control of error flags and the FPU's internal environment is also available.

The processor control instructions are:

```
FINIT      ;reset the FPU
FCLEX      ;clear error flags
FINCSTP    ;increment stack pointer
FDECSTP    ;decrement stack pointer
FSTSW      <dst> ;store status register
FSTSW      AX    ;store status register to AX
FSTCW      <dst> ;store control register
FLDCW      <src> ;load control register
FSTENV     <dst> ;store environment
FLDENV     <src> ;load environment
FSAVE      <dst> ;save state
FRSTOR     <src> ;restore state
FFREE      <dst> ;free register
FNOP       ;no operation
FWAIT      ;report FPU error
```

## FPU Programming Examples

The brief introduction to the FPU's instruction set should not prevent us from examining some simple programming applications. The examples that follow will show the usage and form of a number of different FPU instructions. Remember that these floating-point instructions would need to be implemented in software, through time-consuming subroutines, if the FPU was not available.

---

■ **EXAMPLE 11.21** The FPU instructions shown here convert the floating-point Celsius temperature value stored in CEL into its corresponding Fahrenheit equivalent. The result is saved in FAHR.

```
In current data segment...
CEL      REAL4      100.0
C5       REAL4      5.0
C9       REAL4      9.0
C32      REAL4      32.0
FAHR     REAL4      ?
.
.
.
```

```

FINIT
FLD    CEL        ;load Celsius temperature
FMUL   C9         ;multiply by 9
FDIV   C5         ;divide by 5
FADD   C32        ;add 32
FST    FAHR       ;save Fahrenheit result
FWAIT  █

```

---

The FMUL, FDIV, and FADD instructions have only one operand specified. This means that the FPU will use the floating-point number on top of its internal stack as the second operand in each instruction. Let us take another look at how the floating-point stack is used in a computation.

---

■ **EXAMPLE 11.22** The following FPU instructions compute the length of the hypotenuse of a right triangle.

In current data segment...

```

SIDEA    REAL4    3
SIDEB    REAL4    4
SIDEDEC   REAL4    ?
.
.
.

```

```

FINIT
FLD      SIDEA     ;load side A length
FMUL     SIDEA     ;square it
FLD      SIDEB     ;load side B length
FMUL     SIDEB     ;square it
FADD                     ;add the squares together
FSQRT                    ;find the square root
FST      SIDEDEC   ;store length of side C
FWAIT

```

The floating-point stack is used to save the partial results obtained when the individual side lengths are squared. By the time the FADD instruction executes, the top two numbers on the stack are the squared side lengths. FADD pops both of these numbers, adds them, and pushes the result back onto the stack (for FSQRT to use). Using the floating-point stack instead of external memory locations results in much faster execution. █

---

The next example shows how one of the FPU's built-in constants can be used in a calculation.

---

■ **EXAMPLE 11.23** The area of a circle is equal to  $\pi R^2$ , where  $\pi$  (pi) is roughly 3.14159265, and R is the radius of the circle. The coprocessor routine shown here implements the area formula using the coprocessor's internal **pi** constant.

In current data segment...

```

RADIUS    REAL4    5.6
AREA      REAL4    ?

```

```

.
.
.
FINIT
FLD    RADIUS      ;load R
FMUL   RADIUS      ;compute R squared
FLDPI                   ;load pi constant
FMUL                   ;compute area
FST    AREA        ;save result
FWAIT  █

```

---

One common limitation of the example programs just examined is that we have no way of viewing the result of each routine because the floating-point result is saved in the standard IEEE format.

One way to view floating-point numbers is to use a program like CodeView, which performs the necessary conversions and displays the floating-point result in a debugging window.

CodeView's MD (Memory Display) and VM (View Memory) commands allow you to specify a display format that controls how a number read from memory is displayed. For example, to display a REAL4 floating-point value, we enter MDR <address>, where address is the memory address to begin displaying data. Let us consider the routine of Example 11.21. Assume the data area begins at offset 000C in the data segment. The VMR 0x000C command will display the floating-point numbers in scientific notation, beginning at offset 000C. The output looks like this:

```

214B:000C +1.0000000E+002
214B:0010 +5.0000000E+000
214B:0014 +9.0000000E+000
214B:0018 +3.2000000E+001
214B:001C +2.1200000E+002

```

The result of 212 (at offset 001C) indicates that the program worked correctly.

An option within the VM command can be used to add the associated hexadecimal data for each floating-point number to the display. The new command is VMR 0x000C/R+ and has the following format:

```

214B:000C 00 00 C8 42 +1.0000000E+002
214B:0010 00 00 A0 40 +5.0000000E+000
214B:0014 00 00 10 41 +9.0000000E+000
214B:0018 00 00 00 42 +3.2000000E+001
214B:001C 00 00 54 43 +2.1200000E+002

```

This feature of CodeView is very handy, but we still do not have a way for the program to display a floating-point number.

Another way to view these numbers is to use an FPU procedure to perform the conversion. This procedure is covered next.

## SQRT: An FPU Application

The SQRT program contains a set of procedures capable of converting an FPU floating-point number into a 12-digit real number that is displayed on the screen. The floating-point number is converted through repetitive division into its individual digit values. For

example, 371.52 is converted in the following manner:

| <i>Divide</i>         | <i>Round</i> | <i>Multiply</i> | <i>Subtract</i> |
|-----------------------|--------------|-----------------|-----------------|
| $371.52/100 = 3.7152$ | 3            | $3 \times 100$  | $371.52 - 300$  |
| $71.52/10 = 7.152$    | 7            | $7 \times 10$   | $71.52 - 70$    |
| $1.52/1 = 1.52$       | 1            | $1 \times 1$    | $1.52 - 1$      |
| $0.52/0.1 = 5.2$      | 5            | $5 \times 0.1$  | $0.52 - 0.5$    |
| $0.02/0.01 = 2.0$     | 2            | $2 \times 0.01$ | $0.02 - 0.02$   |

The FPCON procedure converts six digits at a time, based on the floating-point number stored in NUM. The first time FPCON is called, the initial divisor value is 100,000. So, numbers as large as 999,999 can be converted with FPCON without making any changes to the program. To convert different-size numbers, the values of DIVI and POW must be changed, as well as the storage requirements for DIGITS.

The DPD procedure converts the integer-based floating-point result values saved in DIGITS into their corresponding ASCII decimal counterparts '0' through '9.'

;Program SQRT.ASM: Display floating-point square root using FPU.

```

;
.MODEL SMALL
.586
.DATA
X      DW      200      ;integer input number
FPX    REAL4    200.0    ;floating point input number
TAI    DB      'The square root of 200 is $'
CRLF   DB      13,10,'$'
DIGITS REAL4    6 DUP(0.0) ;leftmost digit storage
        REAL4    6 DUP(0.0) ;rightmost digit storage
DIVI   REAL4    1.0E5     ;initial divisor
POW    DW      6         ;number of digits to display
NUM    REAL4    ?        ;FP working number
TEN    REAL4    10.0     ;used in conversion
CW     DW      ?        ;FPU control word

.CODE
.STARTUP
LEA    DX,TAI          ;set up pointer to square root message
MOV    AH,9            ;display string function
INT    21H             ;DOS call
FINIT                     ;initialize FPU
FLD    FPX             ;load floating-point x value
FSQRT                     ;calculate square root
FST    NUM             ;save result
FWAIT
CALL    FPDISP          ;display floating point result
LEA    DX,CRLF          ;set up pointer to newline string
MOV    AH,9            ;display string function
INT    21H             ;DOS call
.EXIT

FPDISP PROC NEAR
SUB     SI,SI           ;clear index pointer
CALL    FPCON           ;convert leftmost digits
CALL    FPCON           ;convert rightmost digits
LEA     SI,DIGITS       ;set up pointer to digits

```



```

        CALL    DPD                ;display leftmost digits
        MOV     DL, '.'            ;load decimal point character
        MOV     AH, 2              ;display character function
        INT     21H                ;DOS call
        CALL    DPD                ;display rightmost digits
        RET
FPDISP ENDP

FPCON  PROC NEAR
        MOV     CX, POW            ;set up loop counter
DODIG: FINIT                ;initialize FPU
        FNSTCW  CW                ;store control word
        OR      CW, 0C00H          ;set rounding control to truncate
        FLDCW   CW                ;load control word
        FLD     NUM                ;get current value of number
        FDIV    DIVI              ;divide by multiple of 10
        FRNDINT                ;round result to integer
        FST     DIGITS[SI]        ;save result in digit buffer
        FLD     DIVI              ;load divisor
        FMUL    DIGITS[SI]        ;multiply by integer result
        FLD     NUM                ;load current value of number
        FSUBR                   ;calculate remainder
        FST     NUM                ;save new value of number
        FLD     DIVI              ;divide divisor by 10
        FDIV    TEN
        FST     DIVI
        FWAIT
        ADD     SI, 4              ;advance to next digit
        LOOP    DODIG             ;and repeat
        RET
FPCON  ENDP

DPD    PROC NEAR
        MOV     CX, POW            ;set up loop counter
OP:    MOV     AH, [SI+3]          ;load exponent information
        MOV     AL, [SI+2]
        SHL     AX, 1              ;calculate exponent
        SUB     AH, 127
        MOV     DL, [SI+2]        ;load initial bits of number
        OR      DL, 80H           ;always begin with 1 as MSB
        SUB     DH, DH            ;clear DH
SDL:   SHL     DX, 1              ;double number
        CMP     AH, 0              ;is exponent zero?
        JZ      OP2              ;if yes, go display digit
        DEC     AH                ;repeat until exponent equals 0
        JMP     SDL
OP2:   MOV     DL, DH            ;load digit value
        ADD     DL, 30H           ;add ASCII bias
        MOV     AH, 2              ;display character function
        INT     21H                ;DOS call
        ADD     SI, 4              ;advance pointer to next FP number
        LOOP    OP                ;and repeat
        RET
DPD    ENDP
END

```

The execution of Sqrt results in this output

The square root of 200 is 000014.142135

Note that leading 0's are not eliminated by the DPD procedure. You may wish to experiment with larger numbers to investigate the accuracy of the conversion process.

---

**Programming Exercise 11.1:** Write a program that will display the temperature found with the FPU code from Example 11.21. Use the FPCON procedure from Sqrt.ASM to display the result.

---

**Programming Exercise 11.2:** Repeat Programming Exercise 11.1 for the other FPU routines covered in Examples 11.22 and 11.23.

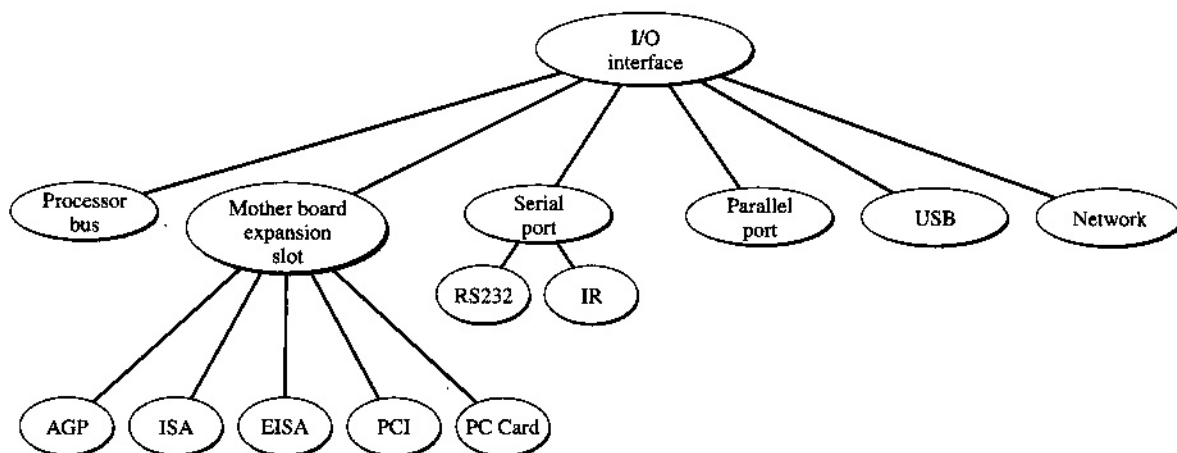
---

**Programming Exercise 11.3:** Modify the Sqrt program so that leading 0's are not displayed.

---

## 11.5 INTERFACING WITH THE PERSONAL COMPUTER

Since its appearance in the early 1980s, the personal computer has been interfaced with many different types of hardware using a wide variety of methods. Beginning with its original ISA bus, the PC has branched out its I/O capabilities, as indicated in Figure 11.26. If you are using an 8088 or 8086 in your design, you may choose to interface directly with the processor bus signals, as we have previously seen earlier in this chapter and in Chapters 9 and 10. This is not really an option for the advanced CPUs available today unless you have sophisticated surface-mount integrated circuit and multi-layer PCB capability. The original ISA bus has been surpassed in bit-width and transfer speed by the PCI and high-speed USB busses. Where you may have once purchased a prototype board for the ISA bus



**FIGURE 11.26** I/O possibilities on the PC

that allowed you to design and build your own ISA bus interface (an I/O port or peripheral interface), that is not the case today. Instead, new devices have emerged to assist with our interfacing needs, with a strong trend to do lots of things using serial I/O.

In this section we will examine several interfacing applications for the personal computer, using a variety of techniques.

### Using the Serial Port

For a while it seemed as if serial communication was getting overlooked, as networking grabbed the headlines and programmable logic became more prevalent. But the simplicity of a two-wire or three-wire communication system for transmitting and receiving 8-bit codes is too attractive to ignore. Today, there are serial peripherals and integrated circuits, such as serial A/D converters, serial EEPROMs, and more. One such device is the BPI-216, a dot-matrix alphanumeric LCD (Liquid Crystal Display) display with two lines of 16 characters each. The BPI-216 is available from Scott Edward's Electronics, Inc (<http://www.seetron.com>). The BPI-216 has these features:

- 5 volt operation.
- RS232 serial communication at 2400 or 9600 baud.
- Two lines of 16 characters each. Characters are in a 5x7 dot-matrix format.
- Control codes can be used to format and control the display.

The BPI-216 is easy to use, just send it an ASCII character string and the characters appear on the display. Figure 11.27 shows the character set of symbols that may be displayed on the BPI-216. It is even possible to create your own 5x7 characters. Send the BPI-216 a printable ASCII character code and that character appears in the current cursor position. Note that the display memory for the BPI-216 is 40 characters wide, so received characters may not be visible on the display until they are scrolled into position.

To send a control code to the BPI-216, you must first send the instruction-prefix code, equal to the value 254 (0xFE). This is not an ASCII character value and the BPI-216 recognizes that the next code it receives will be a command code. These command codes are listed in Table 11.1. So, to clear the display screen of the BPI-216, send the 254 prefix code first, followed by the 1 code. To scroll left, send the 254 prefix code, followed by the 24 code.

|   | 0 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 160 | 168 | 176 | 184 | 192 | 200 | 208 | 216 | 224 | 232 | 240 | 248 |   |
|---|---|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|
| 0 | ▲ |    | (  | 0  | 8  | 0  | H  | P  | X  | `  | h   | P   | X   |     | イ   | ー   | ク   | タ   | ネ   | ミ   | リ   | α   | λ   | ρ   | ×   |   |
| 1 | ▲ | !  | )  | 1  | 9  | A  | I  | Q  | Y  | a  | i   | 4   | 9   |     | ウ   | ア   | ケ   | チ   | ノ   | △   | ル   | ä   | '   | α   | U   |   |
| 2 | ▲ | "  | *  | 2  | :  | B  | J  | R  | Z  | b  | j   | r   | z   |     | 「   | エ   | イ   | コ   | ツ   | ン   | ×   | レ   | e   | i   | θ   | チ |
| 3 | ▲ | #  | +  | 3  | ;  | C  | K  | S  | [  | c  | k   | s   | (   | 」   | オ   | ウ   | サ   | テ   | ヒ   | モ   | ロ   | ε   | *   | ω   | 万   |   |
| 4 | ■ | \$ | ,  | 4  | <  | D  | L  | T  | ¥  | d  | l   | t   |     | 、   | ハ   | エ   | シ   | ト   | フ   | パ   | ワ   | μ   | φ   | Ω   | 円   |   |
| 5 | ■ | %  | -  | 5  | =  | E  | M  | U  | ]  | e  | m   | u   | )   | ・   | ユ   | オ   | ス   | ナ   | ハ   | ン   | シ   | モ   | Ü   | ÷   |     |   |
| 6 | ■ | &  | .  | 6  | >  | F  | N  | V  | ^  | f  | n   | v   | +   | ヲ   | ヨ   | カ   | セ   | ニ   | ホ   | ヨ   | *   | o   | n   | Σ   |     |   |
| 7 | ■ | ?  | /  | 7  | ?  | G  | O  | W  | _  | g  | o   | w   | +   | ア   | ッ   | キ   | ソ   | ヌ   | マ   | ラ   | °   | α   | ö   | π   | ■   |   |

FIGURE 11.27 Character set for the BPI-216

**TABLE 11.1** Control codes for the BPI-216 LCD display

| <i>Code</i> | <i>Instruction/Action</i>                |
|-------------|------------------------------------------|
| 1           | Clear screen                             |
| 2           | Home (set cursor to line 1, character 1) |
| 8           | Blank the display                        |
| 12          | Restore the display / Hide cursor        |
| 13          | Enable blinking-block cursor             |
| 14          | Enable underline cursor                  |
| 16          | Move cursor one character left           |
| 20          | Move cursor one character right          |
| 24          | Scroll display one character left        |
| 28          | Scroll display one character right       |
| 64 + addr   | Set pointer in character generator RAM   |
| 128 + addr  | Set cursor position                      |
| 192 + addr  |                                          |

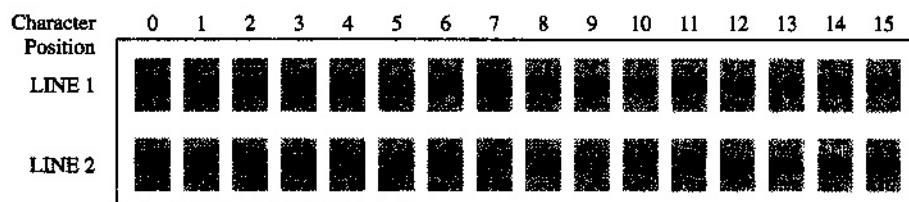
You may want to write characters to specific locations on the two-line display. To do this you must position the cursor at the desired character position before writing the character or characters. Figure 11.28 illustrates the numeric addresses of each character position on the BPI-216's display. To set the cursor position to character position 12 (4<sup>th</sup> character from the right end) on line 1, we send the prefix code 254 followed by the address value 140. To get to character position 2 on line 2, we send the prefix code of 254 followed by the address value 194.

The following C function is used to output the message "System Ok" on the BPI-216 LCD display, with the word "System" centered on line 1 and the word "Ok" centered on line 2.

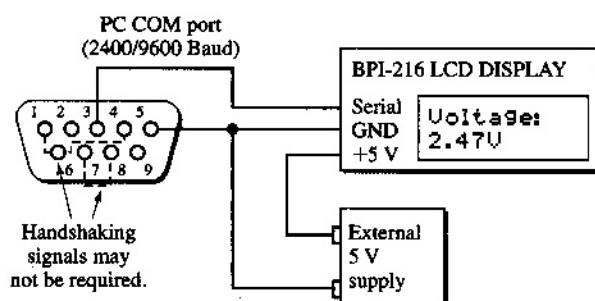
```
void SystemOK()
{
    writeLCD(254);           // Send instruction prefix
    writeLCD(1);             // Clear LCD screen

    writeLCD(254);
    writeLCD(133);           // Set cursor to Line 1, character 5
    sendLCDstr("System");

    writeLCD(254);
    writeLCD(199);           // Set cursor to Line 2, character 7
    sendLCDstr("Ok");
}
```

**FIGURE 11.28** Address assignments for each display character

**FIGURE 11.29** Interfacing the BPI-216 LCD display to the PC



The `writeLCD` function outputs a single character to the BPI-216 LCD display. The `sendLCDstr` function outputs a string of ASCII characters. The code for `sendLCDstr` looks like this:

```
void sendLCDstr(char lcdstr[])
{
    int k;
    for(k = 0; k < strlen(lcdstr); k++)
        writeLCD(lcdstr[k]);
}
```

So, we see that `writeLCD` is the only function that actually talks to the BPI-216 LCD display.

Figure 11.29 shows how the BPI-216 is connected to the PC's COM port. Note that the PC does not provide power for the BPI-216, so an external 5 volt supply is required.

Use the program files contained on the companion CD to experiment with your own LCD messages.

## Using the Parallel Printer Port

In the old days, the personal computer performed limited parallel I/O through its printer port. The 25-pin connector for the parallel port, and its signal assignments, are shown in Figure 11.30. The limited bidirectional capability of the original printer port was understandable; it only needed to control a printer. But, eventually, the printer port became useful for other things, such as external drives and video cameras. As the parallel printer port hardware evolved (EPP Enhanced Parallel Port and ECP Enhanced Capabilities Port modes of operation), so did the operating system watching over it. In fact, Windows NT/2000/XP make it very difficult to get at the hardware of the system, including the I/O ports associated with the parallel port. Even so, clever programmers have found ways around the hardware limitation, developing DLLs that contain kernel-mode device drivers that allow a high-level language to manipulate the hardware through calls to the DLL. There are plenty of resources on the Web for anyone interested in working with the parallel printer port.

One useful parallel application is controlling a stepper motor. As illustrated in Figure 11.31, four parallel output bits are used to control the four coils in a uni-polar stepper motor. A logic one level on any parallel output turns on its associated Darlington driver (within the ULN2003 integrated circuit) and stepper motor coil. Table 11.2 shows the pattern sequence used to energize the stepper motor coils in the proper fashion. The basic process is to send a pattern to the stepper motor, wait for a short delay for the coils to stabilize (and the mechanical parts to settle), and then go to the next pattern sequence and repeat. The stepper motor rotates as pairs of coils take turns being energized. If you output the pattern sequence in the opposite order, the stepper motor steps in the opposite direction.

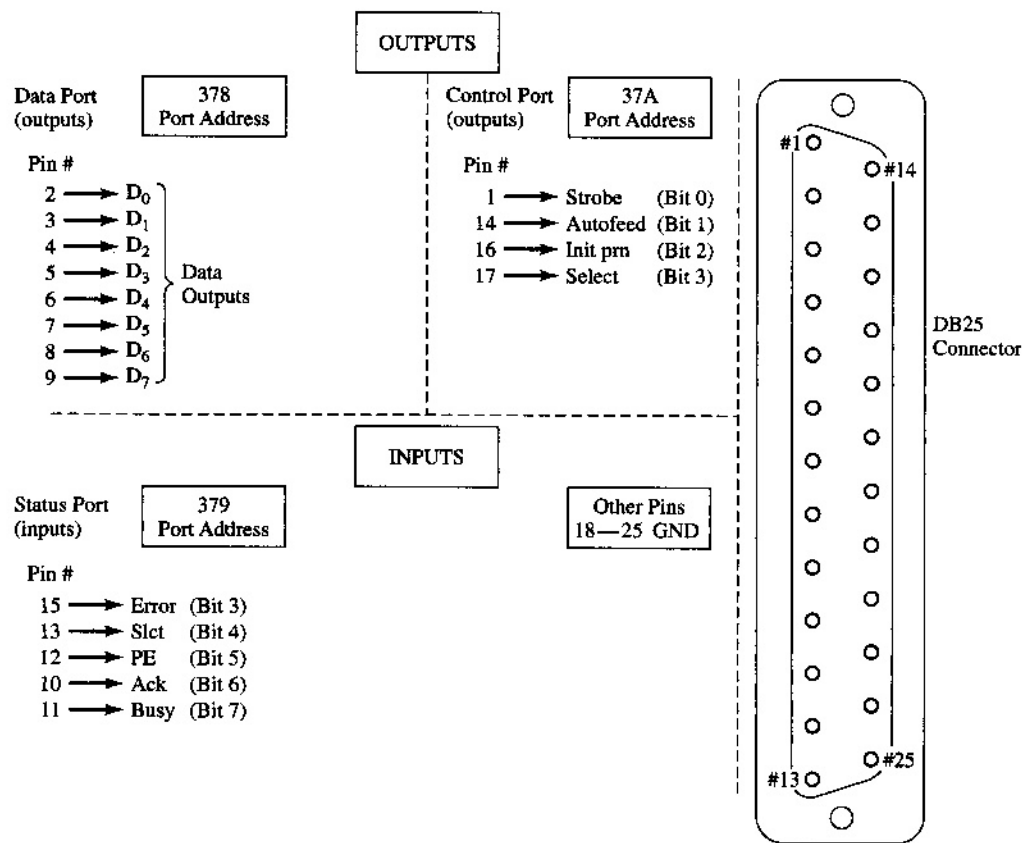


FIGURE 11.30 DB25 parallel (printer) port connections

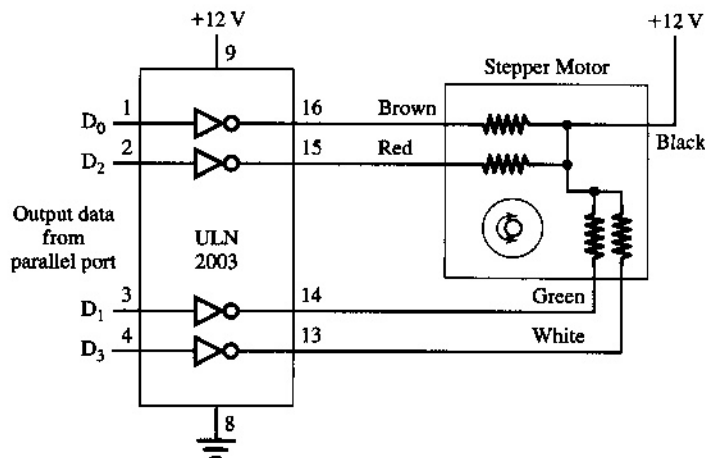


FIGURE 11.31 Parallel port interface for a uni-polar stepper motor

**TABLE 11.2** Stepper motor pattern sequence

| <i>Pattern</i> | <i>D<sub>3</sub></i> | <i>D<sub>2</sub></i> | <i>D<sub>1</sub></i> | <i>D<sub>0</sub></i> | <i>Hex Value</i> |
|----------------|----------------------|----------------------|----------------------|----------------------|------------------|
| 0              | 1                    | 1                    | 0                    | 0                    | 0x0C             |
| 1              | 0                    | 1                    | 1                    | 0                    | 0x06             |
| 2              | 0                    | 0                    | 1                    | 1                    | 0x03             |
| 3              | 1                    | 0                    | 0                    | 1                    | 0x09             |

The StepCW function shown here is used to step the stepper motor 3.6 degrees in the clockwise direction. This is accomplished by outputting all four patterns from Table 11.2 in sequence to the parallel port indicated by PPORT parameter.

```
void StepCW()
{
    int p;
    unsigned char patterns[] = {0x0c, 0x06, 0x03, 0x09};
    unsigned char patt;

    for(p = 0; p < 4; p++)
    {
        patt = patterns[p];
        outport(PPORT, patt);
        stepDelay();
    }
}
```

A little experimentation will be required with the amount of time spent processing the stepDelay() function. You may want to begin with 10 milliseconds and adjust as necessary.

## Using the Universal Serial Bus

When the USB was introduced, many computer users breathed a sigh of relief, because I/O expansion no longer required one to open the chassis of the computer and plug in an expansion card.

Now virtually every device you can imagine is available for the USB. To get a project up and running, you do not have to design a USB interface, which would be very complex. Instead, you can use one of many USB interface modules available. For example, the USB I/O 24 module (available through FTDI Products, <http://www.ftdichip.com>) provides 24 bits of parallel programmable I/O in three 8-bit ports. The USB I/O 24 module connects to the PC via USB and responds as if it were a virtual COM port. Two special drivers are installed when the USB I/O 24 is first plugged in. After that, all communication with the USB I/O 24 is through serial communications to the virtual COM port. The communication parameters are 9600 baud, no parity, 8 data bits, and one stop bit. If the driver and hardware installation is successful, you should be able to open a HyperTerminal session (or other serial communication application) and connect to the virtual COM port. Entering a "?" will interrogate the USB I/O 24 hardware, which will respond with a "USB I/O 24" message in return.

The USB I/O 24 has three 8-bit ports (ports A, B, and C) whose direction (in or out) is programmable. Sending serial data to the USB I/O 24 gives it commands and outputs data to the ports. Reading serial data from the USB I/O 24 gets status and inputs data from the ports. Table 11.3 shows the commands for port A. Port B and C commands are similar.

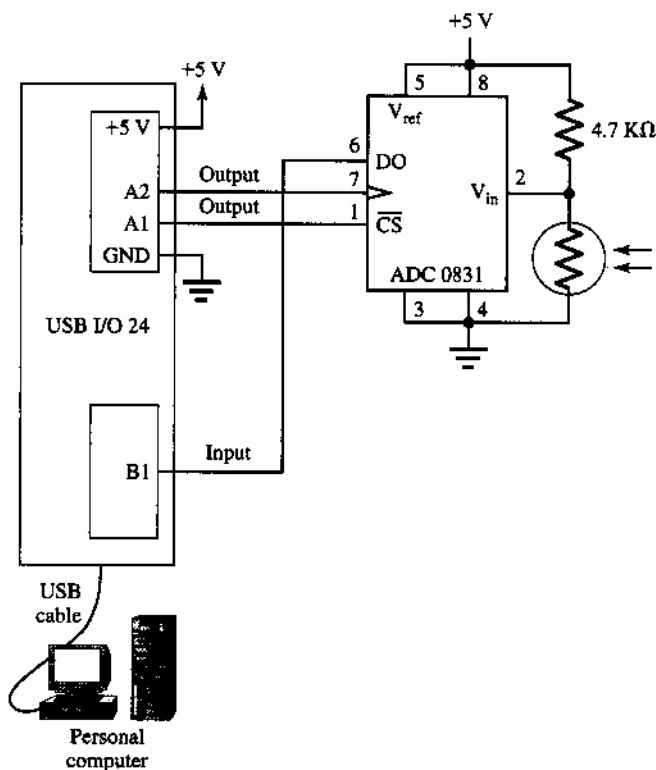
**TABLE 11.3** Port A commands for the USB I/O 24

| Command | Data                                                | Function                           |
|---------|-----------------------------------------------------|------------------------------------|
| "!A"    | One byte of direction bits<br>0 = output, 1 = input | Write to port A Direction register |
| "A"     | One byte of output data                             | Write to port A                    |
| "a"     | One byte of input data                              | Read from port A                   |

To configure port A for output, we send the ASCII characters "!" and "A" to the virtual COM port, followed by a byte of 0s. For an input port we would follow the "!" and "A" with a byte of 1s.

To send data to port A, we send an ASCII "A" to the virtual COM port, followed by the byte of data we want to output. To read data from port A, we send an "a" to the virtual COM port, then read the virtual COM port to get the byte of data from the input port.

Figure 11.32 shows how the USB I/O 24 is used to interface with an 8-bit serial ADC, the ADC0831. Port A is configured for output. Bit A1 controls the chip select ( $\overline{CS}$ ) and bit A2 controls the clock input. Port B is configured for reading and is used to read the data output bit from the ADC0831 into bit B1. The timing sequence required to read the 8-bit ADC value from the ADC0831 is shown in Figure 11.33. A conversion begins when  $\overline{CS}$  is pulled low, and completes after the last bit has been read. Here is a portion of the code used to control the USB I/O 24 (you can find the entire code project on the companion CD).

**FIGURE 11.32** Interfacing a serial A/D converter



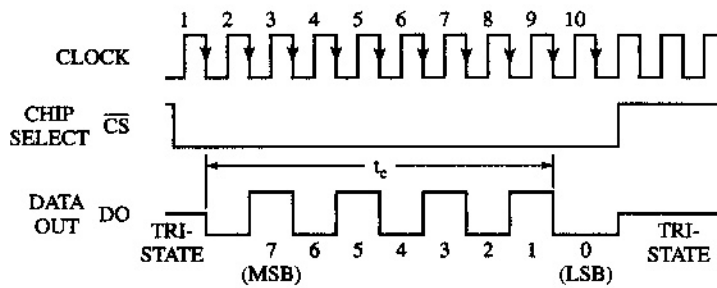


FIGURE 11.33 Timing waveforms for the ADC0831

```

unsigned char Read0831()
{
    unsigned char result, bits, newbit;
    ClearCS();           // Make CS low
    CLKpulse();          // Pulse the 0831s clock
    CLKpulse();
    result = 0;
    for(bits = 0; bits < 8; bits++)
    {
        CLKpulse();      // Clock the 0831
        newbit = ReadDO(); // Read data bit from 0831
        result = result << 1; // Shift current result
        result = result | newbit; // Adjust result
    }
    SetCS();           // Make CS high
    return(result);
}

```

The reason the result variable is shifted right one bit with each pass through the loop is because the order of bits from the ADC0831 is from MSB to LSB. The first data bit read from the ADC0831 will be the MSB. After eight passes through the loop, this initial bit will have been shifted right into the MSB position of the final result value.

Once we have read the ADC0831, determining the original input voltage is straightforward:

```

value = Read0831();
cout << "The voltage is" << 5.0 * value / 255.0 << "volts." << endl;

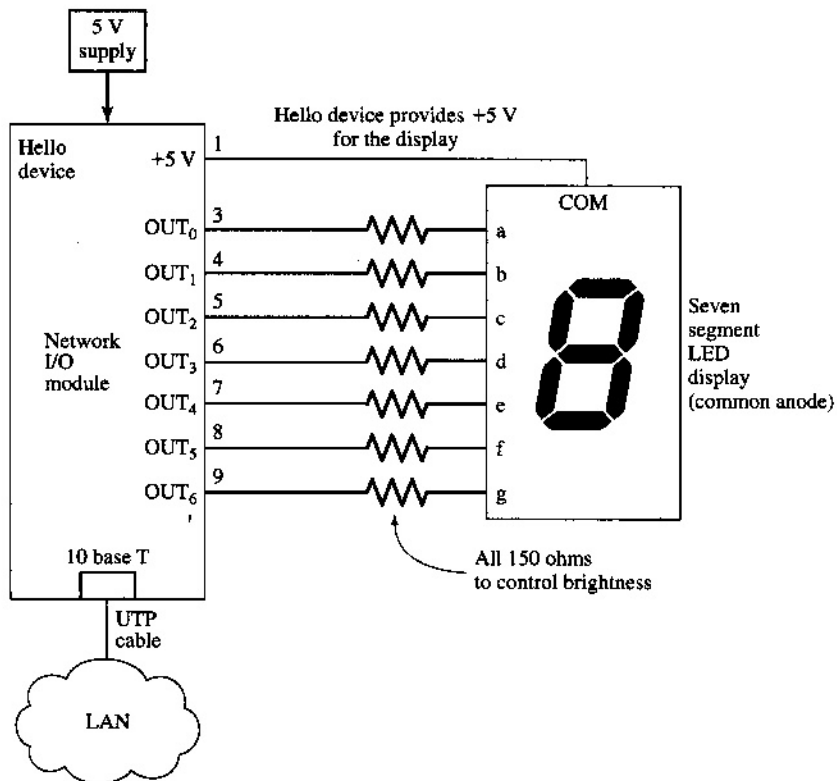
```

Because the result from the ADC0831 is an 8-bit number, its integer value range is 0 to 255. Dividing the ADC0831s value by 255.0 creates a 0 to 100 percent factor, which is then multiplied by 5.0 because the voltage reference for the ADC0831 is 5 volts.

Using serial character strings to control information on the USB is simplicity itself and an attractive reason to invest some effort into learning about this technology.

## Using a Network Interface

Thanks to a wide variety of available hardware and software, interfacing electronic devices to an Ethernet network is accomplished with a minimum of difficulty. Typically the network device has an RJ-45 10baseT Ethernet connector and connections for its I/O signals. A network application communicates with the network device via the UDP or TCP



**FIGURE 11.34** Interfacing the Hello Device to a seven-segment LED display

protocols. The network application we will investigate here involves controlling a seven-segment LED display. The network device used is the Hello Device 1100 from Sena Technologies (<http://www.sena.com>). The Hello Device 1100 has the following features:

- 10baseT RJ-45 connection
- 5 volt operation
- 16 digital inputs
- 16 digital outputs
- Embedded Web server (512K flash memory for content)
- Scenix Sx52 8-bit processor
- TCP/IP protocols: ARP, IP, ICMP, UDP, TCP, HTTP

Figure 11.34 shows the Hello Device interfaced with a common-anode seven-segment display. Seven output signals from the Hello Device control individual segments in the LED display. A zero on any Hello Device output turns the associated segment on (because the display is a common anode display). Series resistors are used to limit the current through the segment and control the segment's brightness.

The Hello Device requires an external 5 volt supply and draws a maximum of 200 mA. When you first configure the Hello Device to work on your network, you must assign it an IP address. The Hello Device will remember its assigned IP address even after being powered off and on again.

The Hello Device runs an embedded Web server. If you enter the IP address of the Hello Device in your browser's address bar, your browser will display the Hello Device's

Web page (which you can change by uploading your own page). A TCP server process running on port 6001 is used to read and write the Hello Device's 16 parallel I/O signals.

Figure 11.35 shows the bit patterns needed to form the numerical digits 0 through 9 on the seven-segment LED display using seven outputs from the Hello Device. These patterns are output by the Hello Device to turn the segments on and off. Examine the following C function to see how these patterns are used.

```
// Output pattern to 7-segment display
void SevenSeg(int cmd, int symbol)
{
    char  cmdBuf[3];
    int   cmdLen;
    int   outdata;
    int   err;
    // Segment patterns for numeric digits 0...9
    unsigned char digits[] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
                             0x02, 0x78, 0x00, 0x10};
    //Initialize output status
    outdata = 0;
    IOSTatus[2] = (outdata & 0x0000ff00)>> 8;
    IOSTatus[3] = (outdata & 0x000000ff);

    //Create TCP command message for Hello Device
    cmdLen = 3;
    cmdBuf[0] = IOSet // IOSet = 0x76
    if(cmd == COUNT)
    {
        cmdBuf[1] = digits[symbol]; // Load segment pattern
        cmdBuf[2] = digits[symbol];
    }
    else
    {
        cmdBuf[1] = 0x7f; // all segments off
        cmdBuf[2] = 0x7f;
    }
    // Send command to HelloDevice
    err = sendto
    (
        sock,
        (char *)&cmdBuf,
        cmdLen,
        0,
        (struct sockaddr*)&clientAddr,
        sizeof(clientAddr)
    );
    if (err == -1)
    {
        perror("\nSend error! Resetting communications.\n");
        TCPSocketClose();
        TCPSocketInit();
    }
}
```

A three-byte command message is sent to the Hello Device using the TCP protocol, which is a reliable communication protocol (compared to UDP, which is unreliable).

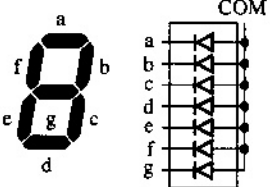
| Hello Device Outputs, Segment Inputs<br>OUT <sub>6</sub> OUT <sub>5</sub> OUT <sub>4</sub> OUT <sub>3</sub> OUT <sub>2</sub> OUT <sub>1</sub> OUT <sub>0</sub><br>g f e d c b a |   |   |   |   |   |   | Hex<br>pattern | Common-Anode<br>7-segment display                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---|---|---|---|---|----------------|------------------------------------------------------------------------------------|
| A zero in any position turns<br>segment on.                                                                                                                                     |   |   |   |   |   |   |                |  |
| 1                                                                                                                                                                               | 0 | 0 | 0 | 0 | 0 | 0 | 40             | 0                                                                                  |
| 1                                                                                                                                                                               | 1 | 1 | 1 | 0 | 0 | 1 | 79             | 1                                                                                  |
| 0                                                                                                                                                                               | 1 | 0 | 0 | 1 | 0 | 0 | 24             | 2                                                                                  |
| 0                                                                                                                                                                               | 1 | 1 | 0 | 0 | 0 | 0 | 30             | 3                                                                                  |
| 0                                                                                                                                                                               | 0 | 1 | 1 | 0 | 0 | 1 | 19             | 4                                                                                  |
| 0                                                                                                                                                                               | 0 | 1 | 0 | 0 | 1 | 0 | 12             | 5                                                                                  |
| 0                                                                                                                                                                               | 0 | 0 | 0 | 0 | 1 | 0 | 02             | 6                                                                                  |
| 1                                                                                                                                                                               | 1 | 1 | 1 | 0 | 0 | 0 | 78             | 7                                                                                  |
| 0                                                                                                                                                                               | 0 | 0 | 0 | 0 | 0 | 0 | 00             | 8                                                                                  |
| 0                                                                                                                                                                               | 0 | 1 | 0 | 0 | 0 | 0 | 10             | 9                                                                                  |
| 1                                                                                                                                                                               | 1 | 1 | 1 | 1 | 1 | 1 | 7F             | Blank display.<br>All segments off.                                                |

FIGURE 11.35 Bit patterns needed for seven-segment display

These three bytes are stored in the `cmdBuf` array. The first byte is the command (IOSet or IOGet). The second and third bytes are the data bytes used to transport the 16 bits of parallel I/O. The `SevenSeg` function builds the TCP command message by filling it with the appropriate output pattern from Figure 11.35. These patterns are stored in the `digits[]` array. The array pattern that is used is selected by the value of the symbol variable passed to `SevenSeg`.

The `SevenSeg` function is called from another function named `Flash`. Here is the code:

```
void Flash()
{
    static int ctr;
    if(ctr == 10)
        ctr = 0;
    SevenSeg(COUNT,ctr);    // Show count on the 7-segment display
    printf("%d...",ctr);    // Show count in the command window
    Sleep(750);
    ctr++;
}
```

`Flash` uses a static integer variable named `ctr` to keep track of a counting sequence that runs from 0 to 9 over and over. A static variable is automatically initialized to zero and its value persists even when the function that defines it is no longer running. So, every time `Flash` is called, the value of the `ctr` variable from the previous execution is used. `Flash` updates the value of the `ctr` variable, automatically resetting it to zero when it reaches 10.

`Flash` itself is called from the main C function as follows:

```
void main()
{
    int stopped = FALSE;
    char inchar;

    TCPSocketInit();

    printf("Counting...\n");
    printf("Press any key to exit...\n\n");

    while (!stopped)
    {
        Flash();
        if(kbhit())
        {
            inchar = getch();
            stopped = TRUE;
        }
    }
    SevenSeg(LEDSON,0);
    TCPSocketClose();
}
```

After the TCP socket is created and initialized with the `TCPSocketInit` function, `Flash` is called over and over again until the user presses any key on the keyboard. Before closing the TCP session, `main` calls `SevenSeg` with the `LEDSON` command to blank the seven-segment display.

The companion CD contains all of the Visual C++ project files for the Hello Device application, in case you want to experiment on your own.

## PC Interfacing Summary

Whether your interfacing application is parallel, serial, or network-based, there are plenty of devices to choose from to assist with the electrical and data interface. All that is necessary to begin is a little money and some imagination.

---

**Programming Exercise 11.4:** Write a C function similar to `SystemOk` that outputs the message "All inputs normal" centered on the display, with the words "All inputs" on line 1 and "normal" on line 2.

---

**Programming Exercise 11.5:** Write the `stepCCW` function to step the stepper motor counterclockwise.

---

**Programming Exercise 11.6:** Write a C function that steps the stepper motor 90 degrees clockwise, followed by 180 degrees counterclockwise, and then by another 90 degrees clockwise. Use the `stepCW` and `stepCCW` functions in your routine.

---

**Programming Exercise 11.7:** Develop a program that reads the ADC0831 and writes the calculated voltage to the BPI-216 LCD display.

---

**Programming Exercise 11.8:** Modify the Hello Device code so that the hexadecimal symbols A through F are also displayed on the seven-segment LED display.

---

---

## 11.6 TROUBLESHOOTING TECHNIQUES

The peripherals and devices in this chapter are suitably advanced. Although they cover a wide variety of topics, there are many other, even more specialized, applications. When you come across a new device and are faced with the challenging task of getting it to work, keep these suggestions in mind:

- Look over the data manual on the new device. If you do not have a data manual, try searching the Web. Intel has a very useful site, offering downloads of manuals (80x86 series plus many others) in PDF format. Many other educational institutions post important information on the Web also, as part of class projects.  
Skim the figures and captions, look at the register and bit assignments, and read the tables. Read about the hardware signals. Study the timing diagrams. Look at any sample interface designs provided by the manufacturer. Be sure you understand why the signals are used the way they are.  
Read about the software architecture of the device. How is it controlled? How do you send data to it, or read data from it? How many different functions does it perform?
- Get the hardware interface working properly. This requires your skill in designing I/O hardware. Some software may be required to fully test the interface.

- If the device has many modes of operation, begin with the simplest. Program the device to operate in this mode to be sure you have control over it. Expand to other modes of operation as you learn more about the device.

Even if all you are doing is modifying someone else's code, written long ago, for a peripheral that is already operational, it is still good to learn as much about the device as possible. This will help you avoid typical problems, such as forgetting to issue the master reset command, even though power was just applied.

---

## SUMMARY

In this chapter, we examined the operation of three peripherals designed to complement the operation of the processor. The first peripheral, the 8259 programmable interrupt controller, exhibited many useful features. A range of interrupt vectors can be programmed and then issued via level- or edge-sensitive inputs that are prioritized. Eight levels of prioritized interrupts are available with a single 8259 operating as a master. The 8259 can be cascaded to provide up to 64 levels of interrupts.

The second peripheral examined was the 8254 programmable interval timer. This device contains three independent 16-bit down counters that can be programmed to count in binary or BCD. Six modes of operation are possible, allowing generation of square and pulse waveforms, programmed time delays, and other time-related functions.

Next, we covered the 80x87 floating-point coprocessor, a device that extends the instruction set of the processor to include operations on a number of different data types, from 32-bit integers to 80-bit real numbers. The 80x87 provides quick execution of complex mathematical operations, eliminating the need for slower—and possibly less accurate—software routines.

We finished with a look at several different ways to interface a hardware device with the personal computer.

---

## STUDY QUESTIONS

1. What are the main differences between polled I/O and interrupt-driven I/O?
2. Redesign the port-address decoder of Figure 11.2 so that the 8259 is mapped to a base port address of 70H. What are all of the port addresses the 8259 will respond to with the new design?
3. What ICWs are needed to program a single 8259 with edge-sensitive inputs and a base interrupt vector of C0H? (ICW4 is not needed.)
4. How many slaves are required if ICW3 in a master 8259 contains 01100101? What devices are connected to each IR input?
5. Draw a schematic of the cascaded 8259 circuit needed in Question 4. Label all interrupt signals in order from lowest to highest priority.
6. A master 8259 has slaves connected to IR<sub>2</sub> and IR<sub>5</sub>. What are the cascade numbers that must be written to each slave?
7. Write the instructions needed to program the 8259 described in Question 2 with the parameters of Question 3.

8. What OCW1 is needed to disable interrupts on IR<sub>3</sub> through IR<sub>6</sub>?
9. A new computer system has five devices that generate interrupts. The devices, in order of highest-to-lowest priority, generate the following signals: DISK, KEYBOARD, TIMER, VIDEO, and IODEV. Show how these signals could be connected to an 8259. What exactly happens if DISK, KEYBOARD, and IODEV interrupts are requested simultaneously?
10. Why does a delay loop become inaccurate if the processor is interrupted? How is this avoided by using the 8254?
11. What instructions are needed to program counter 2 in an 8254 for binary counting in mode 0? The initial count is 3000H. Assume the base port address is B0H.
12. What two ways can be used to load a counter with an initial count of 9F00H?
13. Write a routine that will latch the count of counter 0, store it in register DX, and call TIMEOUT if the count is less than 7.
14. What mode-0 counter value is needed to get a 25-ms time delay? A 1-MHz clock is connected to CLK.
15. What is the longest delay possible with a BCD counter and a 250-kHz clock?
16. Repeat Question 15 for a binary counter.
17. Show how counters 0 and 2 can be cascaded to provide 32-bit counting.
18. Refer to Example 11.13. Name three additional pairs of counts that will produce the 8,000,000 count division.
19. Write a routine to generate a square wave whose frequency, in kilohertz, is specified in BL. BL can take on the values 1 to 99. A 2-MHz clock is available for your use. The 8254 has a base port address of 38H.
20. What are the decimal ranges for each of the FPU's integer formats?
21. Show how the number 1,020.6 is converted into a normalized, short real format.
22. What decimal number is represented by these four assembler-created short real bytes: C3 E5 A9 2B?
23. Show the FPU instructions needed to find the square root of the number stored at LEVEL. Replace LEVEL with the new value.
24. Show how the 80x87 can be used to convert 60 miles/hour into feet/second.
25. Write an 80x87 routine to convert degrees to radians. The input is stored in DEG. Save the result in RAD.
26. Write an 80x87 routine to compute the area of a circle whose diameter is stored in DIAMETER. Save the result in AREA.
27. Name five applications that could make good use of the speed advantage provided by the FPU.
28. Write a routine that will convert a short real number stored at RESULT into an integer part and a fractional part. For example, 80.29 is split into 80 and 0.29. Store the integer part in register AX and the fractional part in BX. Do not use any FPU instructions in your routine.
29. Write the 80x87 instructions needed to evaluate the expression  $X = A^2 - 5 * A * B + B^2$ .
30. What expression is evaluated by the following section of FPU code?

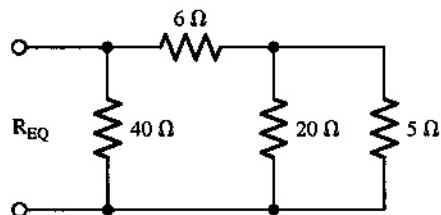
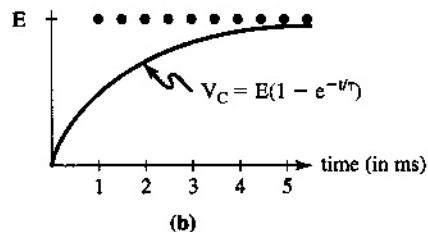
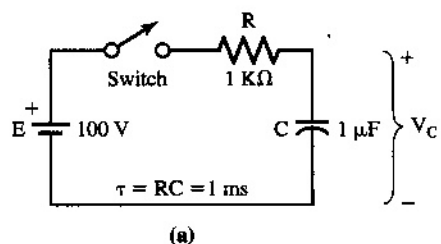
```

FINIT
FLD    W
FDIV   X
FCHS
FMUL   Y
FLD1
FADD
FSQRT
FST    Z

```



31. Write an 80x87 routine that computes the equivalent resistance of the series-parallel circuit shown in Figure 11.36.
32. Write the FPU code needed to implement the charge equation shown in Figure 11.37. If  $E = 100$  V, find  $V_C$  at a time  $t = 2$  ms.

**FIGURE 11.36** For Question 31**FIGURE 11.37** For Question 32

---

# CHAPTER 12

---

## Building a Working 8088 System

---

### OBJECTIVES

In this chapter you will learn about:

- The main parts of a single-board computer
- The design of custom circuitry for the major sections of the microcomputer system
- How to generate and answer the necessary questions for the design or modification of a single-board computer
- The operation of a software monitor program
- How to modify an existing monitor program by writing additional routines

### KEY TERMS

Auxiliary subroutine  
Checksum  
Command recognizer

HEX file  
Null modem

---

## 12.1 INTRODUCTION

This chapter deals exclusively with the design of a custom 8088-based microcomputer system. The system is an ideal project for students wishing to get some hands-on experience and is also a very educational way of using all of the concepts we have studied so far. In addition, even though we are not designing a stand-alone Pentium system, the act of creating a working microprocessor system is still very rewarding. Do not forget that many projects do not require the horsepower of the Pentium. An 8088 will do a fine job controlling a robot arm, for example, whereas a Pentium would be overkill.

Ideally, we wish to design a system that is easy to build, has a minimal cost, and yet gives the most for the money. The very least we expect the system to do is execute programs written in 8088 code. It is therefore necessary to have some kind of software monitor that will provide us with the ability to enter 8088 code into memory, execute programs,

and even aid in debugging. This chapter, then, will consist of two parts. The first part deals with the design of the minimal system, and the second part with the design of a software monitor and the use of its commands.

Pay close attention to the trade-offs that we will be making during the design process. A difficult hardware task can often be performed by cleverly written machine code, and the same goes for the reverse. Do not forget our main goal: to design a *minimal* 8088-based system suitable for custom programming.

Section 12.2 covers the minimum requirements of the system we will design. Section 12.3 describes the design of the system hardware. Section 12.4 contains the parts list for the system. Section 12.5 gives hints on how the system may be constructed. Section 12.6 deals with the design of the software monitor program for the system. Section 12.7 explains a sample session with the single-board computer. Troubleshooting hints are provided in Section 12.8.

---

## 12.2 MINIMAL SYSTEM REQUIREMENTS

The requirements of our minimal system are the same as those of any computer system and consist of four main sections: timing, CPU, memory, and I/O. Because we are the designers building this system for our personal use, it is up to us to answer the following questions:

1. How fast should the CPU clock speed be?
2. How much EPROM memory is needed?
3. How much RAM memory is needed?
4. Should we use static or dynamic RAMs?
5. What kind of I/O should be used—parallel, serial, or both?
6. Do we want interrupt capability? If so, is an interrupt controller necessary?
7. Will future expansion (of memory, I/O, etc.) be required?
8. What kind of software is required?

It should be clear that we have a big task ahead of us. During the design, all of these questions will be answered and the reasons for choosing one answer over another explained. Make sure you understand each step before proceeding to the next one. In this fashion, you should be able to design your *own* computer system, from scratch, and without any outside help.

---

## 12.3 DESIGNING THE HARDWARE

In this section, the four main functional components of the system will be designed. In each case, there will be questions to answer regarding specific choices that must be made. You may want to make a list of all important questions as you go.

### The Timing Section

The timing section has the main responsibility of providing the CPU with a nicely functioning stable clock. Any type of digital oscillator will work in many cases. It is then necessary to decide on a frequency for the oscillator. Many times, this frequency is the operating

frequency of the CPU being used. Microprocessors are commonly available with different clock speeds.

One important factor limiting the clock speed is the speed of the memories being used in the system. A 12-MHz CPU might require RAMs or EPROMs with access times less than 100 ns! In our design, we will use a 10-MHz crystal together with the 8284 clock generator. This is fast enough to provide very quick execution of programs, while at the same time allowing for use of less expensive RAMs with longer access times (200 ns). For fast clocks, we must use a wait-state generator to compensate for slow memories, or use a divider circuit to reduce the clock speed to external components.

The circuit of Figure 12.1 shows a 10-MHz crystal connected to the 8284 clock generator. The output of the 8284 drives two buffers so that any external loading on the CLK signal will not affect its operation. One of the outputs, CPU-CLK, is the master CPU clock signal. Because many other circuits might also require the use of this master clock, we make the CLK signal available too. The CPU therefore gets its own clock signal. It is desirable to separate the clock in this fashion to aid in any digital troubleshooting that may

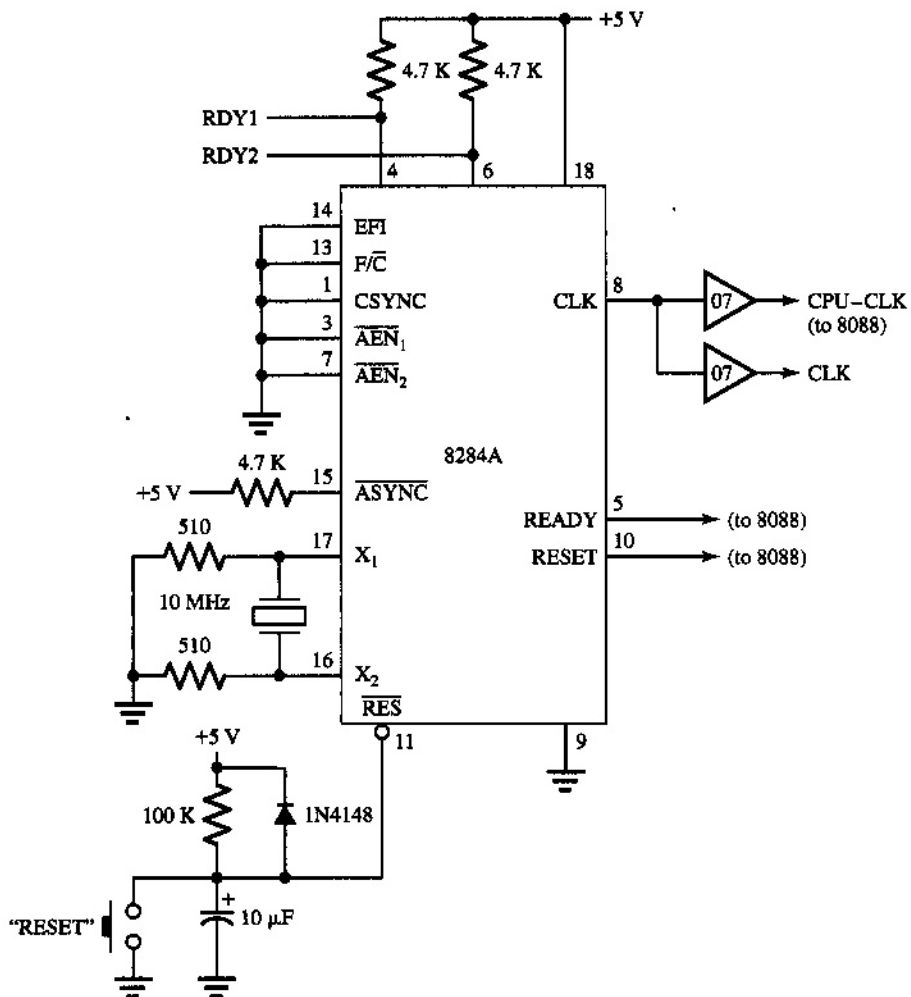


FIGURE 12.1 Clock generator for single-board computer

need to be done. By making multiple clocks available, it is easier to trace the cause of a missing clock, should that problem occur.

In addition to the clock, the CPU must be provided with a reset pulse upon application of power. It is very important to properly reset the CPU at power-up to ensure that it begins executing its main program correctly. The 8284 has built-in reset circuitry that uses an external R-C network to generate the power-on reset pulse. The values shown in Figure 12.1 (100K ohms and 10  $\mu$ F) produce a reset pulse of about 1 ms in duration, long enough to satisfy the hardware reset requirement of the CPU and other system devices.

## The CPU Section

Once we have a working timing section, we must design the CPU portion of our system. During the design of this section, we answer our question about interrupts and pose a few more important questions. For instance, do we need to buffer the address and data lines? Do we want to give bus-granting capability to an external device? Should the system operate in maxmode or minmode? Take a good look at Figure 12.2 before continuing with the reading.

The figure details the connections we must make to the CPU for it to function in our minimal system. On the right side of the CPU, we see the data and address lines. These signals are used in both the memory and I/O sections. The CPU is capable of driving only a few devices by itself (one RAM and one EPROM safely). Because the system we are designing will contain RAM, EPROM, and serial *and* parallel I/O, it is best to buffer the address and data lines. As Figure 12.2 shows, a 74LS244 octal buffer is used to drive address lines  $A_8$  through  $A_{15}$ . Address lines  $A_0$  through  $A_7$  are multiplexed together with the eight data lines, requiring an 8282 octal latch (together with the ALE signal from the 8288) to demultiplex and drive the lower byte of the address bus. These sixteen address lines will allow for 64KB of system memory in our design.

The upper four address lines are not used in this system, because we will not be expanding the system memory requirements past 64KB.

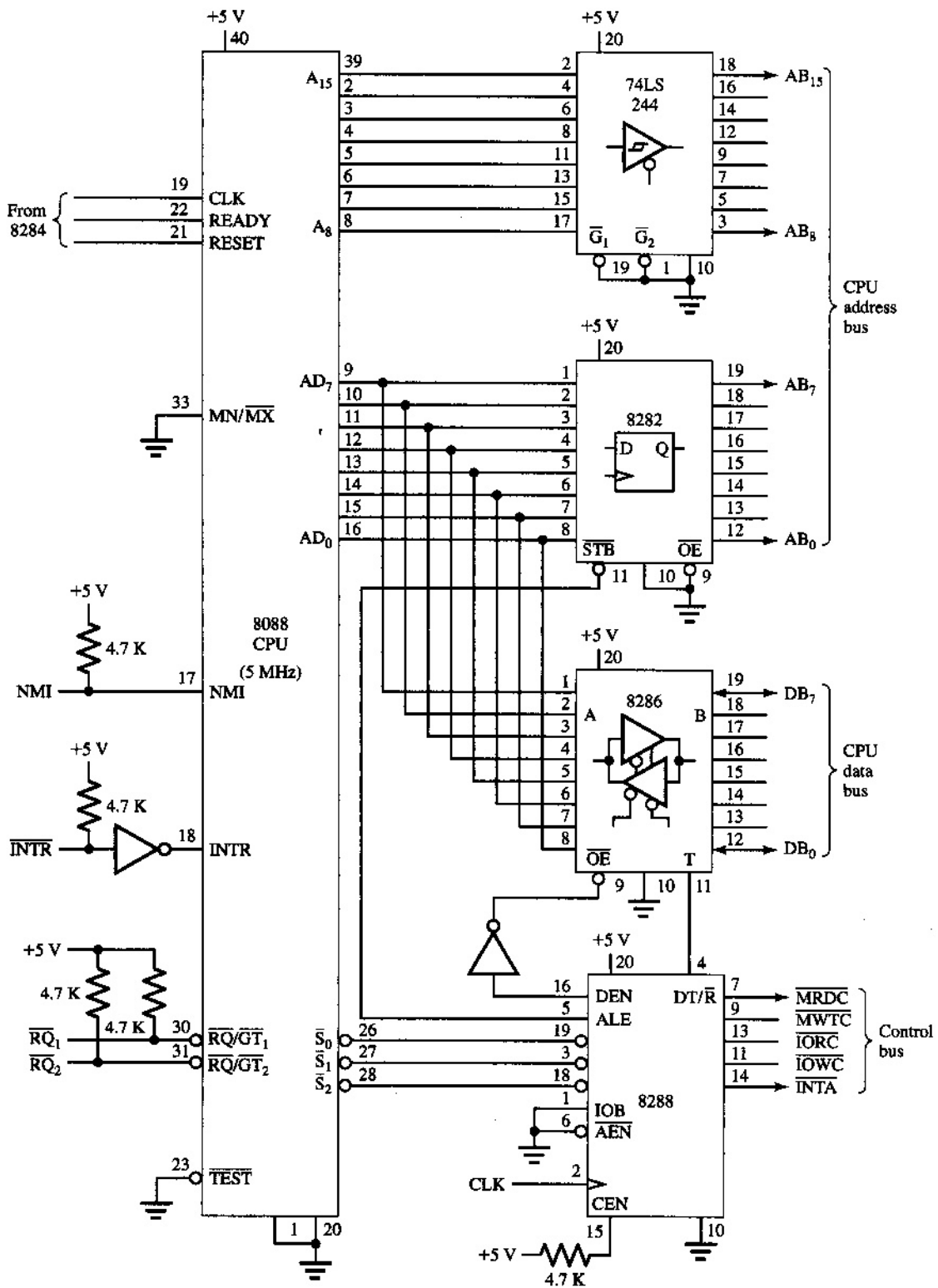
The data bus is buffered in two directions by the 8286 bidirectional line driver/receiver. The direction of data in this device is controlled by the  $DT/\overline{R}$  output of the 8288.

The schematic of the CPU section shows the 8088's  $MN/\overline{MX}$  pin wired to ground. This selects maximum mode operation within the CPU and requires that we use the 8288 bus controller to generate memory and I/O control signals. Do not confuse the description of our system (a *minimal* system) with its mode of operation (*maximum* mode).

The decision to operate the processor in minmode or maxmode depends on a number of factors. If low chip count is necessary, then minmode can be used and the 8288 eliminated. If coprocessor support will be needed in a future expansion of the system, it is best to operate in maxmode from the beginning. The 8288 may also eliminate the need for additional decoding logic in a minimum mode system. Furthermore, bus-granting capability is available only in maxmode. Because no devices in the minimal system use this feature, both  $\overline{RQ}/\overline{GT}$  inputs are pulled high. The pullup resistors will not prevent us from connecting a DMA device to either input at a future time.

Because our goal is to design a system with a minimum of hardware, extensive interrupt support logic will not be necessary. The processor's two external hardware interrupt inputs should serve our needs adequately.

An inverter is used to make the 8088's high-level INTR interrupt respond to a low-level signal. This technique keeps the INTR input in the inactive state if no devices are connected to INTR. NMI is also pulled up to a high level. Remember that NMI is edge-sensitive. When



**FIGURE 12.2** CPU section of single-board computer

no interrupting device is connected, NMI will remain in the high state, and no interrupt will be requested. If we connect a device to NMI in the future, the pullup resistor will not have an adverse effect on any rising-edge NMI signal that is generated.

Technically, although there are five integrated circuits in the CPU section, a *bare-bones* system could get by with only the 8088 running in minmode. But this would most likely require the addition of hardware in the future (to drive the buses and/or possibly switch to maxmode). Any unexpected expansion of hardware is a costly, and sometimes impossible, venture. So, although the minimal system already contains a handful of integrated circuits, choosing maxmode for our project leaves the door open for easy expansion in the future.

## The Memory Section

A number of questions must be answered before we get involved in the design of our memory section. For instance, how much EPROM memory is needed? How much RAM? Should we use static or dynamic RAM? Should we use full or partial address decoding? Will we allow DMA operations?

The answer to each of these questions will help specify the required hardware for the memory section. If we first consider what *applications* we will be using, the previous questions will almost answer themselves. Our application at this time is educational. We desire an 8088-based system that will run short machine-language programs. Keeping this point in mind, we will now proceed to find answers to our design questions.

A programmer, through experience, can estimate the required amount of machine code needed to perform a desired task. The software monitor that we will need to control our system will have to be placed in the EPROMs of our memory section. One standard 2764 EPROM will provide us with 8,192 bytes of programmable memory. This is more than enough EPROM to implement our software monitor. We will still have space left over in the EPROM in case we want to add more functions to the monitor in the future.

The amount of RAM required also depends on our application. Because we will be using our system to test only short, educational programs, we can get by with a few hundred bytes or so. Because dynamic RAMs are generally used in very large memory systems (64K, 256K, and more), we will not use them because most of the memory would go to waste. Other reasons exist for not choosing dynamic RAMs at this time. They require complex timing and refresh logic and will also need to be wired very carefully to prevent messy noise problems from occurring. Even if we use a DRAM controller, we will need some external logic to support the controller, which itself could be a very costly item.

For these reasons, we will use static RAM. Even though a few hundred bytes will cover our needs, we will use one 6264 static RAM, thus making our RAM memory 8,192 bytes long also. The 6264 is a low-power static RAM, with a pinout almost exactly identical to the 2764 we are using for our EPROM memory. So, by adding only two more integrated circuits (plus a few for control), our memory needs are taken care of.

Figure 12.3 shows how we use a 74LS138 three- to eight-line decoder to perform partial-address decoding for us. Because we are not concerned with future expansion on a large scale, partial-address decoding becomes the cheapest way to generate our addressing signals. Address lines  $A_{13}$  through  $A_{15}$  are used because they break up the 8088's memory space into convenient ranges (8KB blocks in this case). We completely ignore the state of the upper four address lines ( $A_{16}$  through  $A_{19}$ ). If future expansion beyond the 64KB range is necessary, the upper four address lines must be used to enable the 74LS138.

With  $A_{13}$  through  $A_{15}$  all low, the 74LS138 decoder will output a 0 on the output connected to the RAM's chip-enable input. With  $A_0$  through  $A_{12}$  selecting individual locations

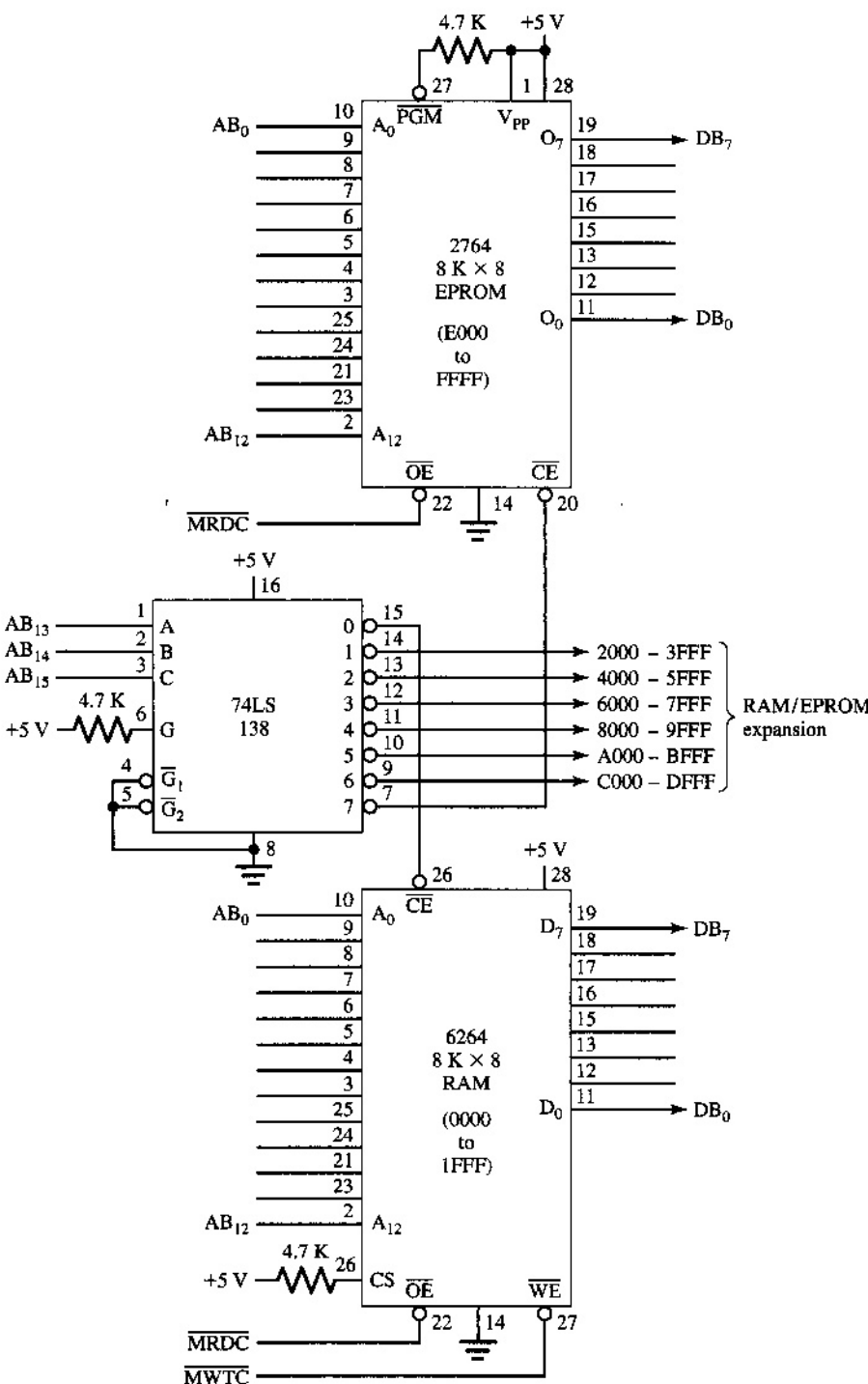


FIGURE 12.3 Memory circuitry for the single-board computer

within the 6264 RAM, we get an address range of 00000H to 01FFFH. Thus, any time the processor accesses memory in that range, the RAM will be enabled. This is a good place for system RAM, because the interrupt vector table must be stored in locations 00000H through 003FFH.



**TABLE 12.1** Partially decoded address ranges in the minimal system

| 74LS138<br>Output | Decoded Address Range | Use        |
|-------------------|-----------------------|------------|
| 0                 | x0000 to x1FFF        | Main RAM   |
| 1                 | x2000 to x3FFF        | Free       |
| 2                 | x4000 to x5FFF        | Free       |
| 3                 | x6000 to x7FFF        | Free       |
| 4                 | x8000 to x9FFF        | Free       |
| 5                 | xA000 to xBFFF        | Free       |
| 6                 | xC000 to xDFFF        | Free       |
| 7                 | xE000 to xFFFF        | Main EPROM |

x = don't care (can be anything from 0 to F)

When  $A_{13}$  through  $A_{15}$  are all high (as they are after a reset causes the initial instruction fetch from FFFF0H), the chip-enable of the 2764 8KB EPROM is pulled low (by the 74LS138). Together with information on the thirteen lower address lines, this maps the EPROM into locations FE000H to FFFFFH. Because partial-address decoding is being used, we can imagine the upper four address lines to be anything we want. This is why we conveniently made them low for the RAM range and high for the EPROM range. Other acceptable RAM ranges are 10000H through 11FFFFH, 50000H through 51FFFFH, and C0000H through C1FFFFH. These are only three more of the sixteen possible RAM address ranges, all of which look identical to the processor. EPROM ranges can be found in a similar manner.

In addition to the RAM and EPROM chip-select signals, the 74LS138 also decodes six additional blocks of addresses. Table 12.1 shows the address range associated with each output of the 74LS138. If additional 8KB RAMs or EPROMs need to be added at a later date, the FREE decode signals can be used to map them into the desired range.

If we were allowing DMA operations, we might not want the 74LS138 to operate in the same way. The enable inputs of the 74LS138 provide us with a way to disable it (all outputs remain high) during a DMA operation, so that an external device may take over the system.

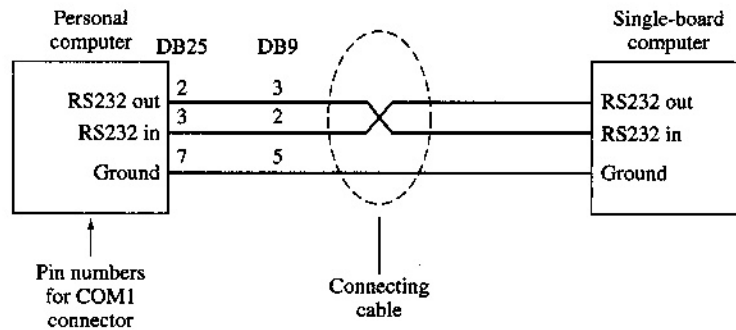
The  $\overline{\text{MRDC}}$  and  $\overline{\text{MWTC}}$  signals generated in the CPU section are used to control the transfer of data between the processor and memory.

## The Serial Section

The serial section of our computer will contain all hardware required to communicate with the outside world (via an EIA-compatible data terminal). One question that must be answered concerns the baud rate at which we will be transmitting and receiving. A very acceptable speed is 2,400 baud. Speeds higher than this will be too fast to read on the screen, and slower speeds will take too long to read.

Figure 12.4 shows the schematic of the serial section, where an 8251 is used to provide serial communications. The chip-enable input of the 8251 is controlled by output 2 of the 74LS138 port-address decoder. Address lines  $A_5$  through  $A_7$  are used by the 74LS138 to decode eight port-address ranges. The 8251 responds to any I/O accesses to ports 40H through 5FH. The remaining seven groups of port addresses are available for expansion.





**FIGURE 12.5** Serial connection to a personal computer

One of these groups will be used to access an 8255 to provide parallel I/O (as shown in the next section).

The MC14411, together with a 1.8432-MHz crystal, generates the required transmitter and receiver clock frequencies for standard baud rates from 300 to 9,600. A DIP switch or jumper can be used to select one of these rates.

The 8251 communicates with the processor via the 8-bit data bus.  $\overline{IORC}$  and  $\overline{IOWC}$ , together with  $A_0$ , control read and write operations in the 8251. CLK is provided to take care of the 8251's internal activities, and RESET is used to initialize the 8251 at power-up.

Because no modem is connected, the 8251's  $\overline{DSR}$  and  $\overline{CTS}$  inputs can be grounded. This ensures that the 8251 is always ready to communicate.

Serial data enters and leaves the 8251 on RxD and TxD. These signals are connected to a MAX232CPE, which converts the 8251's TTL signal levels into RS232-compatible voltage levels, and vice versa. Four 22- $\mu$ F electrolytic capacitors are used to create a  $\pm 10$ V swing on the output of the MAX232. This eliminates the need for an external power supply for these two voltages. The serial-in and serial-out signals from the MAX232 can be wired to a DB25 connector or other suitable connector.

Figure 12.5 shows one way the single-board computer may be connected to a PC. The transmit and receive lines (RS232out and RS232in) on the single-board are connected to the receive and transmit lines on the PC's COM1 connector. The connecting cable is sometimes referred to as a **null modem**, because it cross-couples the transmit and receive lines for full-duplex communication.

Although any port address in the range 40H through 5FH will activate the 8251, the software (via  $A_0$ ) uses only ports 40H and 41H. Port 40H is the 8251's *data* port, which is used to read and write to the receiver and transmitter. Port 41H is the 8251's *status* port, which is used by the software to determine when it is safe to access the receiver or transmitter.

If one serial channel is not sufficient for your needs, a second one can be added by interfacing a second 8251. One of the free port-address ranges should be used to enable the second 8251. Its baud-rate clock will be supplied by the MC14411, and the other half of the MAX232 can be used to drive the second set of serial data lines. A second serial channel is useful for downloading machine code into the minimal system's memory (although this can also be done with a single channel), or for echoing data to a printer.

## The Parallel Section

If parallel I/O is needed, simple latching and buffering circuitry can be used to add a single I/O port using one of the free port-address decoder ranges. If more than one port is needed,

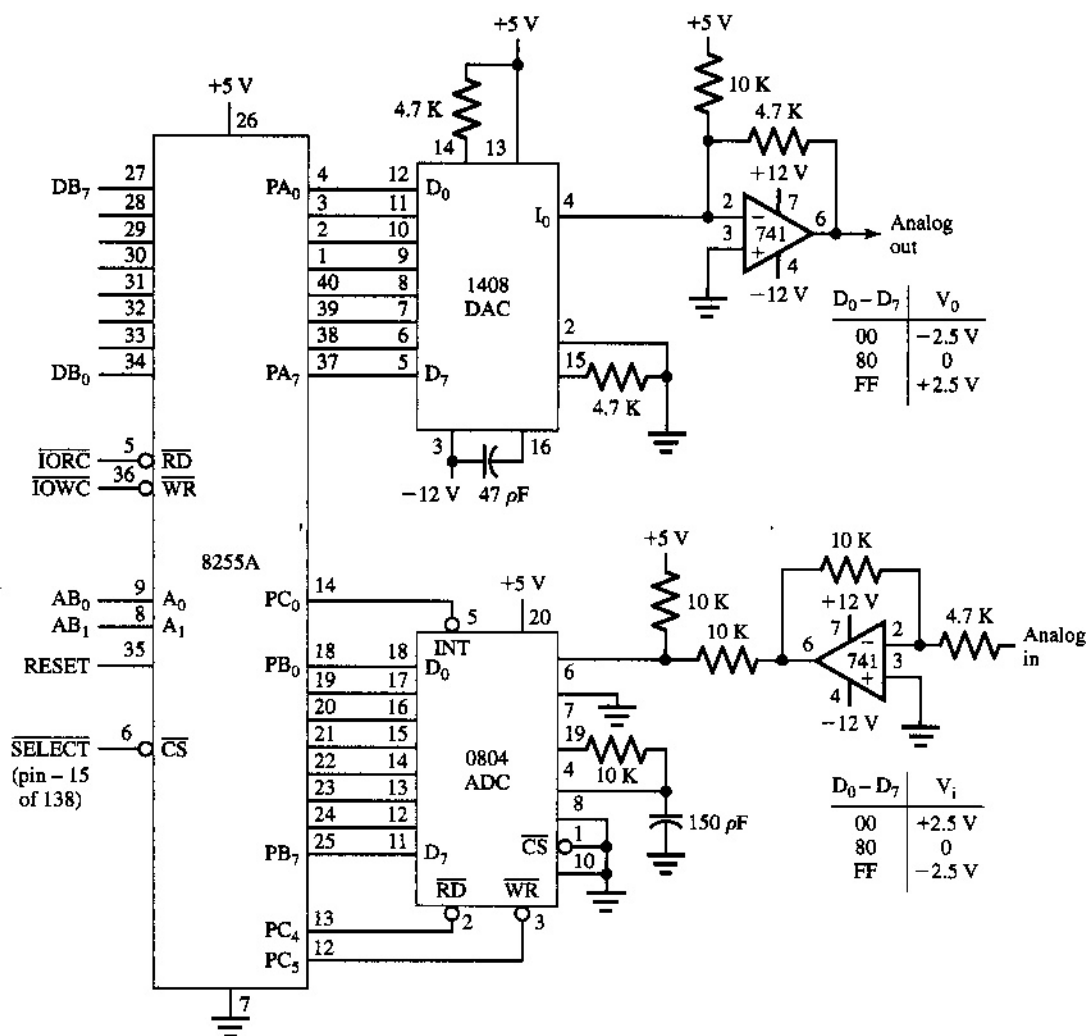


FIGURE 12.6 Parallel I/O circuitry for minimal system

it is best to use a multiport device such as the 8255. The 8255 provides three programmable I/O ports and is easily interfaced with the processor.

Figure 12.6 shows how an 8255 is connected in the minimal system. All of the usual data and I/O signals are connected. Because there are four internal ports in the 8255 (three for data and one for control), two address lines are required to select one of the four internal ports. A<sub>0</sub> and A<sub>1</sub> are used for this purpose. With the chip-enable input of the 8255 wired to the first output of the 74LS138 port-address decoder, port addresses 00H through 03H can be used to select the 8255.

Figure 12.6 also shows how the 8255 is used to provide the minimal system with analog I/O capability. This additional circuitry may not be needed in many applications. In that case, the 8255 merely provides 24 bits of parallel I/O. When analog I/O is a requirement, the circuit of Figure 12.6 provides an acceptable range of analog input and output voltages. A 1408 8-bit digital-to-analog converter is connected to port A of the 8255 (which must be programmed for output operation). The current output of the 1408 is converted into a  $\pm 2.5$ -V swing by a 741 op-amp.

Port B of the 8255 is used to read the output of an 0804 8-bit analog-to-digital converter (which must be programmed for input operation). A second 741 is used to adjust the input voltage range of  $\pm 2.5$  V to the 0- to 5-V swing needed by the 0804. The 0804 is controlled by 2 bits in the 8255's C port. With the 0804 connected in this way, it is possible to digitize over 8,000 analog samples in one second (one sample every 125  $\mu$ s).

Control of the analog circuitry is provided by instructions in the monitor program.

---

## 12.4 THE MINIMAL SYSTEM PARTS LIST

Now that we have finished designing the minimal system, we can look back on all the figures and decide how many ICs we will need to build it. The whole idea behind the design was to build a working maxmode system with a minimum of parts. The following list summarizes all the ICs that are needed (excluding the analog I/O circuitry). Pullup resistors, discrete components, and sockets are not included.

- one 8284A clock generator
- one 74LS07 buffer
- one 74LS04 hex inverter
- one 8088 microprocessor
- one 74LS244 octal buffer
- one 8282 octal latch
- one 8286 bidirectional bus driver
- one 8288 bus controller
- two 74LS138 three- to eight-line decoders
- one 2764 8K by 8 EPROM
- one 6264 8K by 8 RAM
- one 8251 UART
- one MC14411 baud-rate generator
- one MAX232CPE TTL to RS232 converter
- one 10-MHz crystal
- one 1.8432-MHz crystal

In short, only fifteen integrated circuits are needed to build a working 8088 maxmode system.

---

## 12.5 CONSTRUCTION TIPS

The easiest way to build the minimum system is to wire-wrap it. A printed circuit board may be used, but it would be very complex and most likely double-sided.

The minimum system will work the first time power is applied, if the following points are kept in mind:

1. Keep all wires as short as possible. Long wires pick up noise.
2. Connect 0.1  $\mu$ F bypass capacitors across +5 V and GND on all ICs.
3. Trim all excess component leads to avoid short circuits.

4. Connect power and ground to all ICs before wiring anything else.
5. Pull all unused TTL inputs to +5 V with 4.7K-ohm resistors.
6. On a copy of the schematic, mark off connections as they are made.
7. Make sure no ICs are plugged in backward before applying power.
8. Use an ohmmeter to check each connection as it is made.
9. Plug in only the clock ICs first. If the clock does not work, neither will the rest of the system.
10. Check that each IC has proper power before beginning any major troubleshooting.

Experience, of course, is the best teacher, but these hints should be enough to get you started. There is nothing like the feeling of building a circuit that works the first time! If it fails to operate properly, do not get discouraged. With your knowledge of TTL, you should be able to track down the source of the problem in no time. You might be surprised to find that most problems will be due to wrong wiring. Always check your wiring very carefully!

---

## 12.6 WRITING THE SOFTWARE MONITOR

Now that we have the system hardware designed, we must tackle the job of writing the system software. Because our goal is to use the system for testing custom 8088 programs, the monitor program must be capable of performing every step that is needed for us to get the new program into memory, edit it if necessary, display it in hexadecimal format, and execute it. This will require the use of a number of monitor commands. The commands available with the monitor program are:

- B—set breakpoint
- C—clear breakpoint
- D—dump memory contents
- E—enter new register data
- G—go execute a user program
- H—display help message
- I—input data from port
- L—downline load a program
- M—move memory
- O—output data to port
- R—display registers
- S—stop processor
- T—test analog I/O
- X—examine memory

To implement the required monitor commands, we have to write machine language subroutines that perform each function. Many of these routines will perform identical tasks (such as reading an address from the keyboard, converting from hex to ASCII, and outputting data to the display terminal); therefore, we will also need a collection of smaller routines to perform these chores. These routines are called **auxiliary subroutines** and are summarized in Table 12.2. We will study the operation of each auxiliary subroutine first, and then see how they are used within the command routines.

**TABLE 12.2** Auxiliary routines

| <i>Name</i> | <i>Function</i>                                            |
|-------------|------------------------------------------------------------|
| IN_IT       | Initialize I/O devices and system tables                   |
| BLANK       | Send ASCII blank to the display                            |
| CRLF        | Send ASCII CR (carriage return), LF (line feed) to display |
| HTOA        | Convert hex to ASCII                                       |
| A_BIAS      | Add ASCII bias to hex value                                |
| H_OUT       | Output four-digit hex number to display                    |
| C_OUT       | Output ASCII character to display                          |
| CH_CASE     | Convert ASCII to uppercase                                 |
| ERROR       | Display error message                                      |
| C_IN        | Read ASCII character from keyboard                         |
| CHK_SUM     | Check sum during downline load                             |
| GET_BYT     | Read byte value from keyboard                              |
| GET_WRD     | Read word value from keyboard                              |
| CON_V       | Convert ASCII to hex                                       |
| GET_NUM     | Read number from keyboard                                  |
| S_END       | Output ASCII message to display                            |
| ENVIR       | Save system environment                                    |
| D_ENV       | Display system environment                                 |
| D_FLG       | Display processor flag states                              |
| DPR         | Display processor registers                                |

Keep in mind that the monitor program has been written using simple instructions and addressing modes. Many of the routines presented probably can be simplified. You are encouraged to rewrite them once the computer has been built and the basic monitor program is up and running.

### The Auxiliary Routines

Each auxiliary routine is designed to perform a specific function. Enough detail will be provided for you to grasp the overall operation of each routine. Pay attention to the methods used to pass information to/from each routine. It will also be useful for you to make a table showing (for each routine) what registers are used for input, which ones are used for output, and which registers are simply used during computations.

**IN\_IT.** This routine initializes the 8251 serial I/O device so that it will be capable of generating waveforms containing 8 data bits, no parity, and 2 stop bits. An X16 clock is also selected. IN\_IT also programs the 8255 for port A out and port B in, with port C used for handshaking with the ADC. The DAC is initialized to output 0 V. The monitor's breakpoint flag is reset. The receiver of the 8251 is read to clear any stray character that may be present at power-on. The 8251 is accessed through data port 40H and control/status port 41H. The 8255 is accessed through ports 0 through 3. Equate statements are included in the

monitor program to associate these port addresses with labels. For example, the data port of the 8251 is defined like this:

```
S_DATA EQU 40H
```

The label is used in an instruction to aid in understanding what port is being accessed. The routine looks like this:

```
IN_IT PROC NEAR
    MOV AL,0CEH
    OUT S_CTRL,AL ;output 8251 mode word
    MOV AL,5
    OUT S_CTRL,AL ;output 8251 command word
    MOV AL,83H
    OUT AD_CTRL,AL ;init 8255
    MOV AL,80H
    OUT D_AC,AL ;zero DAC
    MOV AL,10H
    OUT AD_STAT,AL
    MOV AL,30H
    OUT AD_STAT,AL ;reset ADC
    MOV ES:[BR_STAT+RTOP],0 ;clear breakpoint flag
    IN AL,S_DATA ;clear 8251 receiver
    RET
IN_IT ENDP
```

**BLANK.** This routine outputs an ASCII blank character to the display. It may first appear that a routine dedicated to this simple function is a waste of time (and code). But consider that there may be many routines that need to output blanks during their execution. It is much more convenient for the programmer to simply CALL BLANK than to duplicate the instructions each time.

```
BLANK PROC NEAR
    MOV AL,20H ;code for ASCII blank
    CALL C_OUT
    RET
BLANK ENDP
```

**CRLF.** This routine is used to make the display scroll up one line. The codes for carriage return and line feed are output to the display. This routine, like BLANK, is needed often.

```
CRLF PROC NEAR
    MOV AL,13 ;ASCII CR
    CALL C_OUT
    MOV AL,10 ;ASCII LF
    CALL C_OUT
    RET
CRLF ENDP
```

**HTOA.** This routine performs hex-to-ASCII conversion. The data byte contained in register AL is converted into a two-character sequence of ASCII characters and output to the display. For example, if AL contains 3FH, an ASCII "3" and an ASCII "F" are output to the display.

```
HTOA PROC NEAR
    PUSH AX
    SHR AL,1 ;get upper nibble
```



```

        SHR     AL,1
        SHR     AL,1
        SHR     AL,1
        CALL    A_BIAS      ;convert to ASCII and output
        POP     AX          ;get lower nibble
        CALL    A_BIAS      ;convert and output again
        RET
HTOA    ENDP

```

**A\_BIAS.** This routine converts the lower 4 bits of register AL into a printable ASCII equivalent character. For example, ----0011 becomes 33H, which is "3." ----1011 becomes 42H, which is "B." The character is then output to the display.

```

A_BIAS  PROC    NEAR
        AND     AL,0FH      ;clear upper 4 bits
        ADD     AL,30H      ;add ASCII bias
        CMP     AL,3AH      ;is it A through F?
        JC      NO_7        ;no
        ADD     AL,7        ;yes, correct to alphabetic
NO_7:    CALL    C_OUT      ;output to display
        RET
A_BIAS  ENDP

```

**H\_OUT.** This routine outputs the four-character ASCII equivalent of the number stored in register DX. For example, if DX contains 3E7CH, the ASCII characters "3," "E," "7," and "C" are output to the display. Notice how HTOA is used to simplify this routine.

```

H_OUT   PROC    NEAR
        MOV     AL,DH      ;do upper byte first
        CALL    HTOA
        MOV     AL,DL      ;then lower byte
        CALL    HTOA
        RET
H_OUT   ENDP

```

**C\_OUT.** This routine examines the level of the 8251's transmitter-ready flag, and, when it is ready, outputs the character in AL to the transmitter (for viewing on the display).

```

C_OUT   PROC    NEAR
        PUSH    AX
        MOV     AH,AL
CO_S:    IN      AL,S_CTRL   ;get 8251 status
        AND     AL,01H      ;test TRDY
        JZ      CO_S        ;loop until not busy
        MOV     AL,AH
        OUT     S_DATA,AL    ;output character
        POP     AX
        RET
C_OUT   ENDP

```

**CH\_CASE.** This routine examines the ASCII character in register AL. If the character is lowercase ("a" through "z"), it is converted into uppercase ("A" through "Z").

```

CH_CASE PROC    NEAR
        CMP     AL,'a'      ;test for 'a'...'z' range
        JC      UN_ALPH
        CMP     AL,'z'+1

```

```

                JNC     UN_ALPH
                AND     AL,0DFH      ;convert into uppercase
UN_ALPH:       RET
CH_CASE       ENDP

```

**ERROR.** This routine gives an audible beep when an error is detected. A “?” is also displayed on the screen to indicate an error.

```

ERROR  PROC     NEAR
        MOV     AL,'?'      ;output '?'
        CALL    C_OUT
        MOV     AL,7        ;beep terminal (control-G)
        CALL    C_OUT
        RET
ERROR  ENDP

```

**C\_IN.** This routine examines the level of the 8251's receiver-ready flag, and, when ready, reads a character from the receiver. The MSB of the byte returned in register AL is always cleared.

```

C_IN   PROC     NEAR
CI_S:  IN       AL,S_CTRL    ;read 8251 status
        AND     AL,02H      ;test RRDY
        JZ      CI_S        ;loop until ready
        IN      AL,S_DATA    ;read new character
        AND     AL,7FH      ;clear MSB
        RET
C_IN   ENDP

```

**CHK\_SUM.** This routine checks the C\_SUM location in the monitor's data table and returns an error message if it is not zero. A nonzero C\_SUM means that a program was downline-loaded incorrectly.

```

CHK_SUM  PROC     NEAR
        CMP     ES:[C_SUM+RTOP],0    ;test C_SUM
        JZ      GD_LD               ;OK if zero
        LEA     SI,CSE              ;load address of error message
        CALL    S_END               ;output message
        JMP     GET_COM              ;get a new command
GD_LD:   RET
CHK_SUM  ENDP

```

The error message is stored in the monitor's data area, like this:

```
CSE  DB  '<-Checksum error->$'
```

with the “\$” serving as the end-of-string character.

**GET\_BYT.** This routine is used to read two successive characters from the serial port and convert them into their 8-bit equivalent. For example, if “5” and “C” are received, register AL will contain 5CH on return.

```

GET_BYT  PROC     NEAR
        CALL    C_IN           ;get a character
        CALL    C_OUT          ;echo it to display
        CALL    CON_V          ;convert into binary
        SHL     AL,1           ;move result into upper 4 bits
        SHL     AL,1
        SHL     AL,1

```

```

        SHL     AL,1
        MOV     BL,AL      ;save upper half of result
        CALL    C_IN       ;get second character
        CALL    C_OUT      ;echo it
        CALL    CON_V      ;convert to binary
        OR      AL,BL      ;combine with upper half
        ADD     ES:[C_SUM+RTOP],AL ;add new value to C_SUM
        RET
GET_BYT  ENDP

```

Notice that the received byte is added to C\_SUM. This allows GET\_BYT to be used in the downline-loading command.

**GET\_WRD.** This routine calls GET\_BYT twice to read a four-character number from the serial port. The number is returned in register AX.

```

GET_WRD  PROC    NEAR
        CALL    GET_BYT    ;get first byte
        MOV     BH,AL       ;save it
        CALL    GET_BYT    ;get second byte
        MOV     AH,BH       ;return result in AX
        RET
GET_WRD  ENDP

```

**CON\_V.** This routine converts an ASCII character in the range "0" to "9" or "A" to "F" into its corresponding 4-bit binary equivalent. The result is returned in the lower half of register AL.

```

CON_V    PROC    NEAR
        SUB     AL,'0'      ;remove ASCII bias
        CMP     AL,10       ;test 'A' to 'F' range
        JC      NO_SUB7     ;not a letter
        SUB     AL,7         ;remove alpha bias
NO_SUB7:  RET
CON_V    ENDP

```

**GET\_NUM.** This routine accepts a multidigit hexadecimal number from the keyboard and stores it in register DX. If more than four characters are entered, only the last four will be used in the conversion. A CR or blank will terminate the input. If no characters are entered prior to the CR or blank, register BH will contain "0," otherwise it will contain "1." Any illegal character causes an exit to ERROR.

This short table gives a few examples of GET\_NUM at work:

| Inputs                   | Outputs        |
|--------------------------|----------------|
| '3','A','7' <cr>         | DX=03A7 BH='1' |
| '1','2','3','4','5' <cr> | DX=2345 BH='1' |
| <cr>                     | DX=0000 BH='0' |

This is the actual routine:

```

GET_NUM  PROC    NEAR
TOP_N:   MOV     BH,'0'      ;init BH
        MOV     DX,0        ;clear DX
GT_NM:   CALL    C_IN       ;get a character
        CMP     AL,13       ;is it CR?
        JZ      GT_BYE     ;yes
        CALL    C_OUT      ;no, echo it

```

```

        CMP     AL,20H      ;is it a blank?
        JZ      GT_BYE     ;yes
        MOV     BH,'1'     ;no, adjust BH
        CALL    CH_CASE    ;convert to uppercase
        CMP     AL,'0'     ;test '0' to '9' range
        JC      BAD_NM
        CMP     AL,'9'+1
        JC      OK_SUB
        CMP     AL,'A'     ;test 'A' to 'F' range
        JC      BAD_NM
        CMP     AL,'F'+1
        JNC     BAD_NM
        SUB     AL,7        ;remove alpha bias
OK_SUB:  SUB     AL,30H     ;remove ASCII bias
        SHL     DX,1       ;make room for new nibble
        SHL     DX,1
        SHL     DX,1
        SHL     DX,1
        ADD     DL,AL       ;adjust result
        JMP     GT_NM      ;repeat as necessary
GT_BYE:  CLC
        RET
BAD_NM:  CALL    ERROR
        JMP     TOP_N
GET_NUM  ENDP

```

**S\_END.** This routine reads ASCII characters from memory and outputs them to the display. The characters are pointed to by register SI. All character strings must terminate with "\$." Upon entry, SI must point to the first character in the string.

```

S_END    PROC     NEAR
S_NEXT:  MOV     AL,[SI]   ;get a character
        CMP     AL,'$'    ;end of string?
        JZ      S_QWT     ;yes
        CALL    C_OUT     ;no, output to display
        INC     SI        ;point to next character
        JMP     S_NEXT    ;repeat
S_QWT:   RET
S_END    ENDP

```

**ENVIR.** This routine saves the monitor's environment (all registers and flags) when the monitor is reentered from an external source. The usual reentry technique requires an INT 95H instruction at the end of the user routine (see Section 12.7). You may think of ENVIR as the interrupt service routine for INT 95H. ENVIR also reestablishes the monitor's segment registers, which may have been altered by the external software. All registers are stored in memory as 2-byte words (low byte first) beginning at location R\_DATA[RTOP] (location 1F80 in the system RAM). The registers are stored in the following order: AX, BX, CX, DX, BP, SI, DI, SP, DS, SS, ES. The flags are stored in locations 1FA1 and 1FA2.

When ENVIR completes execution it *falls into* D\_ENV to display the system environment.

```

ENVIR    PROC     NEAR
        PUSHF                ;save flags on stack
        PUSH     AX          ;save AX on stack
        MOV     AX,DS

```

```

PUSH    AX           ;save DS on stack
MOV     AX,DATA
ADD     AX,0E00H
MOV     DS,AX        ;reload monitor DS
MOV     AX,ES
PUSH    AX           ;save ES on stack
MOV     AX,0
MOV     ES,AX        ;reload monitor ES
POP     AX           ;pop and save old ES
MOV     ES:R_DATA[RTOP + 20],AX
POP     AX           ;pop and save old DS
MOV     ES:R_DATA[RTOP + 16],AX
POP     AX           ;pop and save old AX
MOV     ES:R_DATA[RTOP+0],AX
MOV     ES:R_DATA[RTOP+2],BX    ;save all other registers
MOV     ES:R_DATA[RTOP+4],CX
MOV     ES:R_DATA[RTOP+6],DX
MOV     ES:R_DATA[RTOP+8],BP
MOV     ES:R_DATA[RTOP+10],SI
MOV     ES:R_DATA[RTOP+12],DI
POP     AX
POP     BX
MOV     ES:R_DATA[RTOP+14],SP    ;save SP
PUSH    BX
PUSH    AX
MOV     ES:R_DATA[RTOP+18],SS    ;save SS
POP     AX
MOV     ES:[F_LAGS+RTOP],AX      ;save flags

```

This routine physically appears just before the code for D\_ENV. The instruction following the last MOV in ENVIR is the first instruction of D\_ENV. The data table containing the stored register values is defined like this:

```
R_DATA    DW    11 DUP(?)
```

where memory space for eleven words (one for each register saved) is reserved. When INT 95H is used at the end of a user routine, it causes ENVIR to execute, which, in turn, saves the final state of each register at the completion of the user routine.

**D\_ENV.** This routine displays the contents of all CPU registers stored in the R\_DATA table by ENVIR. The display format (with sample register values) is as follows:

```

AX:1111  BX:2222  CX:3333  DX:4444
BP:5555  SI:6666  DI:7777  SP:8888
DS:9999  SS:AAAA  ES:BBBB

```

After outputting the value of ES, the routine falls into D\_FLG to display the state of the flags.

```

D_ENV:  MOV     SI,0           ;init pointer to register data
        MOV     CX,11         ;init loop counter
        CALL    CRLF
T_DR:   MOV     AL,R_LETS[SI]   ;output register name
        CALL    C_OUT
        MOV     AL,R_LETS[SI+1]
        CALL    C_OUT
        MOV     AL,':'         ;and a ':'
        CALL    C_OUT

```

```

        MOV     DX,ES:R_DATA[RTOP+SI]    ;get register data
        CALL    H_OUT                    ;and output it
        CALL    BLANK                    ;2 blanks for spacing
        CALL    BLANK
        ADD     SI,2                      ;point to next register
        MOV     AX,SI                    ;do display formatting
        AND     AL,7
        JNZ     ADJUST
        CALL    CRLF
ADJUST:  LOOP    T_DR                    ;repeat for all registers

```

**D\_ENV** requires a predefined data table of register names. The table looks like this:

```
R_LETS  DB  'AXBXCXDXBPSIDISPDSSES'
```

**D\_FLG.** This routine displays the state of each of the five arithmetic flags: sign, zero, auxiliary carry, parity, and carry, in the following format:

```
Flags:  S=1  Z=0  A=0  P=1  C=1
```

Together with the register display of **D\_ENV**, **D\_FLG** provides a method for determining exactly what a user routine has done to the registers and flags during execution.

```

D_FLG:  LEA     SI,FL_MSG                ;output flag message
        CALL    S_END
        MOV     SI,0                    ;init pointer to tables
        MOV     CX,5                    ;init loop counter
N_FLG:  MOV     AL,F_SYM[SI]             ;get a flag name
        CALL    C_OUT                  ;display it
        MOV     AL,'='
        CALL    C_OUT                  ;display '='
        MOV     AX,ES:[F_LAGS+RTOP]    ;get flag byte
        AND     AL,F_MASK[SI]          ;mask out specific flag
        MOV     AL,'0'                 ;adjust AL according to flag state
        JZ      NOT_1
        INC     AL
NOT_1:  CALL    C_OUT                  ;display flag state
        CALL    BLANK                  ;and output spacing blanks
        CALL    BLANK
        INC     SI                     ;point to next flag
        LOOP    N_FLG                 ;repeat
        RET
ENVIR   ENDP

```

**D\_FLG** requires three predefined data tables, which are:

```

FL_MSG  DB  13,10,'Flags: $'
F_SYM   DB  'SZAPC'
F_MASK  DB  80H,40H,10H,4,1

```

Note the use of the **ENDP** statement in the routine. **ENVIR**, **D\_ENV**, and **D\_FLG** are all contained within the same procedure block.

**DPR.** This final auxiliary routine is used to enter **D\_ENV** and display the flags, *assuming they have been previously saved*.

```

DPR     PROC    NEAR
        JMP     D_ENV
DPR     ENDP

```

This code is used to display the environment without having to encounter an INT 95H in the user code.

As you study the command routines in the next section, watch how the auxiliary routines are used to simplify the code required to execute a monitor command.

## The Monitor Commands

The monitor commands are really the heart of the software monitor. Through the use of the monitor commands, the job of creating new and useful software becomes much easier. We will now examine just how these commands are implemented. Study the methods used to perform I/O with the user, and how decisions are made within the routines. You should be able to gain a very good understanding of the structure of a command routine and be able to use that knowledge to write *your own* command routine to perform a job that the basic monitor cannot.

## The Command Recognizer

To use a command routine, we must be able to get to its starting address in memory (to fetch the first instruction of the command routine). It is much easier and more convenient to enter a single letter command, such as D, G, or R, than to enter a multidigit hexadecimal starting address. The purpose of the **command recognizer** is to determine which of the monitor commands has been entered by the user, and jump to the command routine for execution. The command recognizer accepts uppercase and lowercase command letters.

Once the command is recognized, the address of the selected command routine is read from a data table, and the routine is jumped to. For this reason, the command routines must jump back to the beginning of the monitor program (GET\_COM) and not return as a subroutine would.

The command recognizer is written so that new commands may be easily added with a minimum of change in its code.

```

GET_COM:  MOV     SP,2000H      ;init stack pointer
          CALL    CRLF         ;newline
          MOV     AL,'>'      ;output command prompt
          CALL    C_OUT
          CALL    C_IN         ;get command letter
          CALL    C_OUT        ;echo it
          CALL    CH_CASE      ;convert to uppercase
          CMP     AL,13        ;if CR, start again
          JZ      GET_COM
          MOV     CX,NUM_COM    ;init loop counter
          MOV     SI,0         ;init table pointer
C_TEST:   CMP     AL,COMS[SI]   ;compare user command with table item
          JZ      CHK_SN       ;match
          ADD     SI,2         ;no match, point to next item
          LOOP    C_TEST       ;repeat test
          CALL    ERROR        ;no match at all
          JMP     GET_COM
CHK_SN:   CALL    C_IN         ;command must be followed by CR
          CMP     AL,13        ;or blank
          JZ      DO_JMP
          CALL    C_OUT
          CMP     AL,20H
          JZ      DO_JMP

```

```

CALL    ERROR
JMP     GET_COM
DO_JMP: JMP     J_UMPS[SI]    ;fetch command routine address and jump

```

GET\_COM uses the COMS data table during the search for a matching command letter. The individual command letters are all followed by a blank, so that when SI is incremented by 2 it will always access the next command letter. This lets GET\_COM use SI to point to the correct location within the command routine address table J\_UMPS as well.

The data tables used by GET\_COM are:

```

NUM_COM  DW    14
COMS     DB    'B C D E G H I L M O R S T X'
J_UMPS   DW    B_RKP, C_BRP, D_UMP, I_NIT
          DW    E_XEC, H_ELP, P_IN, L_OAD
          DW    M_OVE, P_OUT, D_REG, S_TOP
          DW    T_EST, E_XAM

```

## The Command Routines

The command routines are designed to provide the features necessary to load a new program into memory, debug it (find/fix errors), and execute it. Some routines perform operations on the system itself, rather than on the user program. Examples of these types of commands are M (move memory), S (stop processor), and I (input data from port). Study the command routines carefully. Look for ways to improve them once your personal system is up and running.

**The B Command: Set Breakpoint.** This command is used to specify the address where the user program should break away from its execution and return to the monitor. All registers are saved and displayed upon return.

Breakpoints are very useful when debugging code. To use a breakpoint, select an instruction (and its associated address in memory). The instruction located at the breakpoint address is saved and then replaced by an INT 3 instruction (opcode byte CCH). When the processor gets to the INT 3 instruction (during execution of the user program), a special monitor reentry procedure will be executed, which saves and displays the registers and flags. Then the original instruction is restored and the breakpoint cleared.

If a breakpoint is already set, additional B commands will produce an error message and the breakpoint will not be changed.

The format of the command is B <address>. An example of the B command is:

```
B 110C
```

which will cause a breakpoint at address 110C of the user program.

```

B_RKP:  CALL    GET_NUM                ;get breakpoint address
        MOV     AL,ES:[BR_STAT+RTOP]   ;check breakpoint status
        CMP     AL,0
        JZ      DO_BP                  ;no breakpoint saved yet
        LEA     SI,BP_AS                ;breakpoint already saved
        CALL    S_END
        JMP     GET_COM
DO_BP:  MOV     ES:[B_MMA+RTOP],DX      ;save breakpoint address
        MOV     SI,DX
        MOV     AL,ES:[SI]              ;fetch byte at breakpoint
        MOV     ES:[OP_KODE+RTOP],AL    ;and save it
        MOV     AL,0CCH                 ;insert breakpoint code

```



```

MOV     ES:[SI],AL
MOV     ES:[BR_STAT+RTOP],1      ;adjust breakpoint status
LEA     SI,BP_SA                 ;inform user about breakpoint
CALL    S_END
JMP     GET_COM

```

**B\_RKP** uses two messages to inform the user of what it has done:

```

BP_AS    DB    'Breakpoint already saved...$'
BP_SA    DB    'Breakpoint saved.$'

```

**B\_RKP** determines which message to send based on the user's actions.

**The C Command: Clear Breakpoint.** This command clears the saved breakpoint address. The breakpoint status is adjusted and the user instruction restored. This command must be used before attempting to set a new breakpoint.

The C command has no parameters.

```

C_BRP:   MOV     AL,ES:[BR_STAT+RTOP]    ;check breakpoint status
          CMP     AL,1
          JZ      OP_LD                  ;breakpoint exists
          LEA     SI,BR_ALC              ;no breakpoint, inform user
          CALL    S_END
          JMP     GET_COM
OP_LD:   LEA     SI,BR_CLR                ;tell user breakpoint cleared
          CALL    S_END
          MOV     SI,ES:[B_MMA+RTOP]     ;load breakpoint address
          MOV     AL,ES:[OP_KODE+RTOP]   ;load user instruction byte
          MOV     ES:[SI],AL             ;restore user byte
          MOV     ES:[BR_STAT+RTOP],0    ;clear breakpoint status
          JMP     GET_COM

```

**C\_BRP** uses two predefined messages to inform the user about what it has done:

```

BR_ALC    DB    'Breakpoint already cleared.$'
BR_CLR    DB    'Breakpoint cleared.$'

```

**The D Command: Display Memory Contents.** This command displays the contents of memory in hexadecimal format. Each line of the display contains the starting address, followed by 16 bytes of code. A sample line of the display looks like this:

```
1000    F3 1A 29 B3 02 CA 33 F7 88 34 CD 02 10 2A BB C9
```

The first byte (F3H) was read out of memory location 1000, the second byte (1AH) from location 1001, and so on. The last byte on the line (C9H) is read out of location 100F.

The format of the command is: **D** <starting address> <ending address>. An example of the D command is:

```
D 1000 100F
```

which results in the sample output previously shown.

The routine takes this form:

```

D_UMP:   CALL    GET_NUM                ;get starting address
          PUSH    DX                    ;save it on stack
          CALL    BLANK
          CALL    GET_NUM                ;get ending address
          POP     SI                     ;retrieve starting address
          SUB     DX,SI                  ;compute length of block

```

```

                MOV     CX,DX           ;init loop counter
                INC     CX
                MOV     DX,SI           ;go start dump
                JMP     DO_DMP
CHK_ADR:        MOV     DX,SI           ;check for address wrap-around
                MOV     AL,DL
                AND     AL,0FH          ;is least-significant digit zero?
                JNZ     NO_ADR          ;no
DO_DMP:         CALL    CRLF           ;yes output newline and address
                CALL    H_OUT
                CALL    BLANK
NO_ADR:         CALL    BLANK
                MOV     AL,ES:[SI]      ;get a byte from memory
                CALL    HTOA           ;display it
                INC     SI              ;point to next location
                LOOP    CHK_ADR        ;repeat
                JMP     GET_COM

```

**The E Command: Enter New Register Data.** This command allows the stored register data (in R\_DATA) to be changed. All processor registers may be altered. The routine displays the name of each register, its current value, and then gives the user the option of entering a new value. If no new value is entered, the current value is not changed. For example:

AX - 1A23?200

represents the first line of output from the E command. Register AX currently contains 1A23, but the user has changed this value to 0200. To skip over a register and not change its current value, simply hit the Enter key.

The E command has no parameters.

```

I_NIT:         MOV     SI,0             ;init pointer
                MOV     CX,11           ;init loop counter
I_NEX:         CALL    CRLF           ;output register name
                MOV     AL,R_LETS[SI]
                CALL    C_OUT
                MOV     AL,R_LETS[SI+1]
                CALL    C_OUT
                CALL    BLANK
                MOV     AL,'-'          ;and a dash
                CALL    C_OUT
                CALL    BLANK
                MOV     DX,ES:R_DATA[RTOP+SI] ;and the saved value
                CALL    H_OUT
                MOV     AL,'?'          ;ask for a new value
                CALL    C_OUT
                CALL    GET_NUM         ;read new value
                CMP     BH,'0'          ;change?
                JZ      NUN             ;no
                MOV     ES:R_DATA[RTOP+SI],DX ;yes, save new value
NUN:           ADD     SI,2             ;point to next register
                LOOP    I_NEX          ;repeat
                JMP     GET_COM

```

**The G Command: Go Execute a User Program.** This command loads registers AX, BX, CX, DX, BP, SI, and DI from the monitor's register storage area (R\_DATA) and then jumps to

the user-supplied address to execute the user routine. Because system RAM is from 00000H to 01FFFH, the user program must be located within the first 8KB of memory. It is suggested that user programs have an ORG of at least 400H so that they do not interfere with the interrupt vector table located in locations 00000H through 003FFH. Also, the monitor reserves a block of system RAM at the high end of free memory (for its internal stack and storage tables). This block begins at address 1F80H. User programs should not utilize RAM above this address either.

The jump to the user program is actually accomplished by pushing the starting address onto the stack and "RETurning" to it.

The format of the command is: G <execution address>. An example of the G command is:

```
G 1000
```

which causes the user program beginning at address 1000H to be executed.

```
E_XEC:  CALL  GET_NUM          ;get starting address
        SUB   AX,AX            ;clear AX
        PUSH  AX              ;let CS=0000 on stack
        PUSH  DX              ;let IP=DX on stack
        CALL  CRLF
        MOV   AX,ES:R_DATA[RTOP+0] ;load data registers
        MOV   BX,ES:R_DATA[RTOP+2]
        MOV   CX,ES:R_DATA[RTOP+4]
        MOV   DX,ES:R_DATA[RTOP+6]
        MOV   BP,ES:R_DATA[RTOP+8]
        MOV   SI,ES:R_DATA[RTOP+10]
        MOV   DI,ES:R_DATA[RTOP+12]
        RET                  ;pop stack to execute user code
```

*The H Command: Display Help Message.* This command displays a help message showing the syntax of each monitor command. The help message is stored as a long text string (terminated by "\$") beginning at H\_MSG.

The H command has no parameters. Its routine is:

```
H_HELP:  LEA    SI,H_MSG      ;init pointer to help message
        CALL   S_END         ;display message
        JMP    GET_COM
```

*The I Command: Input Data from Port.* This command is used to read and display the byte present at a specific input port. The byte read from the specified input port is displayed in square brackets, as in [34].

The format of the command is: I <port address>. An example of the I command is:

```
I 2F
```

which displays the byte seen at input port 2FH.

```
P_IN:    CALL   GET_NUM      ;get input port address
        CALL   CRLF
        MOV    AL,'['       ;display left bracket
        CALL   C_OUT
        IN     AL,DX         ;read input port
        CALL   HTOA         ;display byte
        MOV    AL,']'       ;display right bracket
        CALL   C_OUT
        JMP    GET_COM
```

**The L Command: Downline Load a User Program.** This command is used to downline load a standard Intel-format **HEX file** into system RAM. The HEX file is created by the assembler and linker. A sample line from the HEX file might look like this:

```
:11307A0062792074686520415353454D424C4552212A
```

All lines begin with a ":". Embedded within the line of text is a length byte (11), a load address (307A), a record type (00), data bytes (6279 . . .), and a final byte called a **checksum** byte (2A). A downline loader routine must retrieve the information present in the HEX file and use it to load the embedded information into memory. The L command always loads a program into memory beginning at address 00400H and supports four record types:

- 00—Data record
- 01—End-of-file record
- 02—Extended-address record
- 03—Starting-address record

The HEX file is received by the serial input of the single-board (as if someone were quickly entering it through the keyboard). The routine checks the sum of each line as it is received and gives an error message if the checksum is incorrect.

The L command has no parameters.

```
L_LOAD:      CALL      CRLF
             MOV       BP,400H ;program always loads at 400H
NXT_REC:     MOV       ES:[C_SUM + RTOP],0 ;clear checksum
KOLON:       CALL      C_IN    ;wait for ':'
             CALL      C_OUT
             CMP       AL,':'
             JNZ       KOLON
             CALL      GET_BYT ;get record length
             MOV       CL,AL   ;init loop counter
             MOV       CH,0
             CALL      GET_WRD ;get load address
             MOV       DI,AX
             CALL      GET_BYT ;get record type
             CMP       AL,0
             JZ        D_REC   ;record 00
             CMP       AL,1
             JZ        EOF_REC ;record 01
             CMP       AL,2
             JZ        EA_REC  ;record 02
             CMP       AL,3
             JZ        SA_REC  ;record 03
             LEA       SI,URT  ;record type not valid
             CALL      S_END
             JMP       GET_COM
D_REC:       CALL      GET_BYT ;now get all data bytes
             MOV       ES:[BP][DI],AL ;one by one and store them
             INC       DI ;in memory
             LOOP      D_REC
             CALL      GET_BYT ;read checksum
             CALL      CHK_SUM ;check for correct sum
             JMP       NXT_REC
EOF_REC:     CALL      GET_BYT ;read checksum
             CALL      CHK_SUM ;check for correct sum
             JMP       GET_COM
```

```

EA_REC:  CALL    GET_WRD    ;get new segment address
         SHL     AX,1       ;shift left 4 bits
         SHL     AX,1
         SHL     AX,1
         SHL     AX,1
         MOV     BP,AX       ;load memory pointer
         CALL    GET_BYT    ;read and check sum
         CALL    CHK_SUM
         JMP     NXT_REC
SA_REC:  CALL    GET_WRD    ;load starting address
         CALL    GET_WRD    ;even though it is ignored
         CALL    GET_BYT
         CALL    CHK_SUM
         JMP     NXT_REC

```

L\_OAD requires a predefined error message for acknowledging invalid record types:

```
URT DB '<-Unidentified record type->$'
```

**The M Command: Move Memory.** This command routine is useful for moving blocks of memory data from one location to another. The data included in the input range is not actually moved anywhere, it is *copied* instead. The starting and ending addresses of the block to be moved (copied) must be specified, along with the starting address of the destination. This command can be used after a downline load to move the user program to a different location in system RAM (or to copy monitor code from EPROM into RAM).

The format of the command is: M <starting address> <ending address> <destination address>. An example of the M command is:

```
M 1200 12FF 1600
```

which copies 256 bytes of RAM from addresses 1200H through 12FFH to memory beginning at address 1600H.

```

M_OVE:  CALL    GET_NUM    ;get starting address
         MOV     SI,DX
         CALL    GET_NUM    ;get ending address
         SUB     DX,SI       ;compute length of block
         MOV     CX,DX       ;init loop counter
         INC     CX
         CALL    GET_NUM    ;get destination address
         MOV     DI,DX
MVIT:   MOV     AL,ES:[SI]   ;read a byte
         MOV     ES:[DI],AL  ;write it at new location
         INC     SI          ;advance pointers
         INC     DI
         LOOP    MVIT        ;repeat
         JMP     GET_COM

```

**The O Command: Output Data to Port.** This routine is used to output a data byte to a specific port. The byte and port addresses are specified in the command. This command is useful for testing output-port hardware (such as the serial and parallel I/O sections).

The format of the command is O <output port address> <data>. An example of the O command is:

```
O 7 33
```

which outputs the data byte 33H to port 07H.

```

P_OUT:  CALL    GET_NUM    ;get port address
        PUSH    DX
        MOV     AL, ' '
        CALL    C_OUT
        CALL    GET_NUM    ;get output data
        MOV     AL, DL
        POP     DX
        OUT     DX, AL      ;output data to port
        JMP     GET_COM

```

**The R Command: Display Registers.** This command is used to display the contents of all processor registers (and flags) saved in the R\_DATA storage area. It uses code already contained in the auxiliary routine DPR.

The R command has no parameters.

```

D_REG:  CALL    DPR
        JMP     GET_COM

```

**The S Command: Stop the Processor.** This command halts the processor. It may be useful to examine the level of the 8088's signals when it is halted (or to escape from the halt state with an interrupt).

```

S_TOP:  LEA     SI, S_MSG
        CALL    S_END
        HLT

```

S\_TOP uses a predefined message to indicate that the machine has halted.

```

S_MSG  DB      'Placing 8088 in HALT state.$'

```

**The T Command: Test Analog I/O Ports.** This command is used to test the operation of the analog I/O circuitry (as depicted in Figure 12.6). The user is provided with a choice of two operations. The first test option exercises the digital-to-analog converter by outputting values from a data table that generates a sine wave at the analog output. The system must be RESET to get out of the sine wave routine.

The second test option reads the analog-to-digital converter and echoes the data to the digital-to-analog converter. The echo test is also terminated by RESET.

The T command has no parameters.

```

T_EST:  LEA     SI, TST_MSG    ;output choice message
        CALL    S_END
GET_RP:  CALL    C_IN          ;get user choice
        CALL    C_OUT         ;echo it
        CMP     AL, '1'       ;must be '1' or '2'
        JZ      WAVER
        CMP     AL, '2'
        JZ      EK_0
        CALL    ERROR         ;give an error beep
        JMP     GET_RP
WAVER:  MOV     CX, 256        ;init loop counter
        MOV     SI, 0         ;init pointer to data
P_IE:   MOV     AL, SINE[SI]   ;get a sine wave sample
        OUT     D_AC, AL      ;output to D/A
        ADD     SI, 1         ;point to next data sample
        LOOP    P_IE          ;repeat
        JMP     WAVER         ;go generate another cycle

```

```

E_K0:    IN      AL,AD_STAT      ;check for end-of-conversion signal
         AND     AL,01H
         JNZ     EK_0
         MOV     AL,AD_RD        ;do handshaking for A/D read
         OUT     AD_STAT,AL
         IN      AL,A_DC         ;read A/D
         NOT     AL              ;maintain phase relationship
         OUT     D_AC,AL         ;echo data to D/A
         MOV     AL,AD_WR        ;start a new conversion
         OUT     AD_STAT,AL
         MOV     AL,AD_NOM
         OUT     AD_STAT,AL
         JMP     EK_0            ;repeat

```

WAVR makes use of a 256-byte data table containing the digitized samples of a one-cycle sine wave. The table is as follows:

```

SINE DB 82H,85H,88H,8BH,8EH,91H,94H,97H
      DB 9BH,9EH,0A1H,0A4H,0A7H,0AAH,0ADH,0AFH
      DB 0B2H,0B5H,0B8H,0BBH,0BEH,0C0H,0C3H,0C6H
      DB 0C8H,0CBH,0CDH,0D0H,0D2H,0D4H,0D7H,0D9H
      DB 0DBH,0DDH,0DFH,0E1H,0E3H,0E5H,0E7H,0E9H
      DB 0EBH,0ECH,0EEH,0EFH,0F1H,0F2H,0F4H,0F5H
      DB 0F6H,0F7H,0F8H,0F9H,0FAH,0FBH,0FBH,0FCH
      DB 0FDH,0FDH,0FEH,0FEH,0FEH,0FEH,0FEH,0FFH
      DB 0FEH,0FEH,0FEH,0FEH,0FEH,0FDH,0FDH,0FCH
      DB 0FBH,0FBH,0FAH,0F9H,0F8H,0F7H,0F6H,0F5H
      DB 0F4H,0F2H,0F1H,0EFH,0EEH,0ECH,0EBH,0F9H
      DB 0E7H,0E5H,0E3H,0E1H,0DFH,0DDH,0DBH,0D9H
      DB 0D7H,0D4H,0D2H,0D0H,0CDH,0CBH,0C8H,0C6H
      DB 0C3H,0C0H,0BEH,0BBH,0B8H,0B5H,0B2H,0AFH
      DB 0ADH,0AAH,0A7H,0A4H,0A1H,9EH,9BH,97H
      DB 94H,91H,8EH,8BH,88H,85H,82H,7EH
      DB 7BH,78H,75H,72H,6FH,6CH,69H,66H
      DB 62H,5FH,5CH,59H,56H,53H,50H,4EH
      DB 4BH,48H,45H,42H,3FH,3DH,3AH,37H
      DB 35H,32H,30H,2DH,2BH,29H,26H,24H
      DB 22H,20H,1EH,1CH,1AH,18H,16H,14H
      DB 12H,11H,0FH,0EH,0CH,0BH,9,8
      DB 7,6,5,4,3,2,2,1,0,0,0,0,0,0,0,0
      DB 0,0,0,0,0,0,0,1,2,2,3,4,5,6,7,8
      DB 9,0BH,0CH,0EH,0FH,11H,12H,14H
      DB 16H,18H,1AH,1CH,1EH,20H,22H,24H
      DB 26H,29H,2BH,2DH,30H,32H,35H,37H
      DB 3AH,3DH,3FH,42H,45H,48H,4BH,4EH
      DB 50H,53H,56H,59H,5CH,5FH,62H,66H
      DB 69H,6CH,6FH,72H,75H,78H,7BH,7FH

```

A predefined message is also required to provide the test choices:

```

TST_MSG DB 13,10,'1) Send sinewave to D/A converter, or'
         DB 13,10,'2) Echo A/D to D/A'
         DB 13,10,'Choice ?$'

```

**The X Command: Examine Memory.** This command is used to manually enter a program by hand, or to examine the contents of selected memory locations. The user supplies the starting address in memory. The X command routine displays the address and the contents of

each memory location. The user is given the chance to enter a new value or to leave the data unchanged. The format of the X command is: X <starting address>. An example of the X command is:

```
X 1000
```

which generates the following string of addresses:

```
1000  3A?<sp>
1001  29?07<cr>
1002  B6?<cr>
```

The 3A at address 1000 remained unchanged, because a space was entered at the “?” prompt. The 29 at address 1001 was changed to 07 and the <cr> on the third line terminated the X command.

```
E_XAM:  CALL    GET_NUM      ;get starting address
        MOV     SI,DX
DO_LYN:  CALL    CRLF        ;display starting address
        MOV     DX,SI
        CALL    H_OUT
        CALL    BLANK
        CALL    BLANK
        MOV     AL,ES:[SI]   ;read memory
        CALL    HTOA        ;display byte
        MOV     AL,'?'      ;ask for new data
        CALL    C_OUT
        CALL    GET_NUM
        CMP     BH,'0'      ;change?
        JZ      E_MIP       ;no (must have been CR or SP)
        MOV     ES:[SI],DL   ;yes, replace data
EX_GNA:  INC     SI          ;point to next location
        JMP     DO_LYN      ;repeat
E_MIP:   CMP     AL,20H      ;skip over data?
        JZ      EX_GNA      ;yes
        JMP     GET_COM     ;no, exit
```

## The Body of the Monitor

At this point we have covered the creation of all routines (command and auxiliary) that we need inside our monitor. The last step we need to perform is to collect all the routines together and organize them into a source file. The source file must provide an interrupt vector for the INT 3 instruction used by the breakpoint routine. It must also contain code to initialize the stack pointer, and handle breakpoint and reentry (INT 95H) procedures.

The following source code must reside at the beginning of the source file. It is followed by the code of the command and auxiliary routines.

```
.CODE
;segment usage is as follows
;   CS for machine code
;   DS for rom-based data tables
;   ES for ram-stack operations

ORG    100H

;this is the starting address of the
;reserved system-ram area.
```



```
RTOP EQU 1F80H
```

```
;This is where the re-start (int 95h) and
;breakpoint (int 03h) return vectors are
;generated.
```

```

    LEA    BX,RE_STRT
    MOV    CX,0E00H
    MOV    AX,0
    MOV    SI,0
    MOV    DS,AX
    MOV    [SI+254H],BX    ;init restart address (INT 95H)
    MOV    [SI+256H],CX
    LEA    BX,BR_ENTR      ;init breakpoint address (INT 3)
    MOV    [SI+0CH],BX
    MOV    [SI+0EH],CX
    MOV    AX,CS           ;init all segment registers
    ADD    AX,DATA
    MOV    DS,AX
    SUB    AX,AX
    MOV    ES,AX
    MOV    SS,AX
    MOV    SP,2000H        ;init stack pointer
    CALL   IN_IT           ;init I/O devices
    LEA    SI,HELLO        ;send monitor greeting
    CALL   S_END
    JMP    GET_COM         ;go get first command
;entry point during a breakpoint (via INT 3)
BR_ENTR:
    CALL   ENVIR           ;save/display environment
    LEA    SI,BR_BAK       ;display breakpoint message
    CALL   S_END
    POP    AX              ;get breakpoint address from stack
    MOV    ES:[BR_IP+RTOP],AX
    POP    AX
    MOV    ES:[BR_CS+RTOP],AX
    POP    AX
    MOV    DX,ES:[BR_CS+RTOP] ;display address of breakpoint
    CALL   H_OUT           ;in CS:IP format
    MOV    AL,':'
    CALL   C_OUT
    MOV    DX,ES:[BR_IP+RTOP]
    DEC    DX
    CALL   H_OUT
    MOV    AL,']'
    CALL   C_OUT
    MOV    SI,ES:[B_MMA+RTOP] ;restore user code
    MOV    AL,ES:[OP_KODE+RTOP]
    MOV    ES:[SI],AL
    MOV    ES:[BR_STAT+RTOP],0 ;clear breakpoint status
    JMP    GET_COM
;normal entry point (via INT 95H)
;this routine falls into GET_COM
RE_STRT:
    CALL   ENVIR           ;save/display environment
    LEA    SI,RENTER       ;display reentry message
```

```

CALL    S_END
POP     AX                               ;get exit address from stack
MOV     ES:[BR_IP+RTOP],AX
POP     AX
MOV     ES:[BR_CS+RTOP],AX
POP     AX
MOV     DX,ES:[BR_CS+RTOP]             ;display exit address
CALL    H_OUT
MOV     AL,':'
CALL    C_OUT
MOV     DX,ES:[BR_IP+RTOP]
SUB     DX,2
CALL    H_OUT
MOV     AL,']'
CALL    C_OUT
GET_COM: <now include command recognizer code>

```

Two data segment areas are also used by the monitor software. The first data segment area contains all of the text messages used by the system, other types of data tables (including the sine wave table), and the system equates (for definition of I/O ports).

The second data segment area contains all of the reserved storage locations for the monitor.

```

        .DATA
HELLO    DB      13,10,'8088 Monitor, Ver:3.0$'
WR_BY    DB      13,10,'C1997 James L. Antonakos$'
RENTER   DB      13,10,'Re-entry by external program at [$'
BR_BAK   DB      13,10,'Breakpoint encountered at [$'

;messages from auxiliary and command routines go here also

D_AC     EQU      0
A_DC     EQU      1
AD_STAT  EQU      2
AD_CTRL  EQU      3
AD_RD    EQU      20H
AD_WR    EQU      10H
AD_NOM   EQU      30H
C8255    EQU      83H    ;Aout, Bin, CLin, CHout
S_DATA   EQU      40H
S_STAT   EQU      41H
S_CTRL   EQU      41H
M8251    EQU      0CEH    ;2 stop, no parity, 8 data, *16 clock
C8251    EQU      05H    ;R and T enable only
TRDY     EQU      01H
RRDY     EQU      02H

R_DATA   DW      11 DUP(?)
GO_ADR   DW      ?
BR_STAT  DB      ?
B_MMA    DW      ?    ;breakpoint main-memory address
OP_KODE  DB      ?    ;opcode byte
BR_CS    DW      ?    ;breakpoint CS
BR_IP    DW      ?    ;breakpoint IP
F_LAGS   DW      ?    ;system flags
C_SUM    DB      ?    ;checksum storage

```

The entire source file must be assembled and the resulting machine code burned into an EPROM. You should have a copy of the entire source file (SBC-MON.ASM) for your use.

To get the monitor program to start up correctly, a far jump instruction is placed into the EPROM at the location corresponding to address FFFF0H. The code for the jump instruction is EA 00 01 00 0E, which jumps to the first instruction of the monitor program located at CS:IP = 0E00:0100.

You may already have ideas for improving the design of the monitor program (by rewriting existing commands or adding new ones). *But if you plan to build the minimal system, make sure the basic monitor works before you begin changing it!*

## 12.7 A SAMPLE SESSION WITH THE SINGLE-BOARD COMPUTER

Once you have constructed the minimal system and burned the monitor program into EPROM, you will be ready to test the system. If you are planning on using a PC as the communication console for the single-board computer, it will be necessary to wire both computers together (as previously shown in Figure 12.5), and then execute a *terminal emulation* program on the PC. If no terminal emulator is available, the SBCIO program shown here will do all that is necessary to provide serial I/O between both machines.

```
;Program SBCIO.ASM: Communicate with Single-Board Computer via COM1.
;
.MODEL SMALL
.DATA
XMSG DB 'SBCIO: Press Control-X to exit...',0DH, 0AH, 0DH, 0AH, '$'
.CODE
.STARTUP
    LEA DX,XMSG ;set up pointer to exit message
    MOV AH,9 ;display string function
    INT 21H ;DOS call
COM1: MOV DX,3FDH ;set up status port address
MORE: IN AL,DX ;read UART status
    TEST AL,1 ;has a character been received?
    JNZ READ ;yes, go get it
AKEY: MOV AH,0BH ;check keyboard status
    INT 21H
    CMP AL,0FFH ;has any key been pressed?
    JNZ COM1
    MOV AH,7 ;read keyboard function, no echo
    INT 21H ;DOS call
    CMP AL,18H ;is it Control-X?
    JZ BYE
    PUSH AX ;save character
UWAIT: IN AL,DX ;read UART status
    TEST AL,20H ;check transmitter ready bit
    JZ UWAIT
    MOV DX,3F8H ;set up data port address
    POP AX ;get character back
    OUT DX,AL ;output it
    JMP AKEY
```

```

READ:  MOV     DX,3F8H      ;set up data port address
        IN      AL,DX       ;read UART receiver
        MOV     DL,AL       ;load character for output
        MOV     AH,2        ;display character function
        INT     21H         ;DOS call
        JMP     AKEY
BYE:    .EXIT
        END

```

SBCIO continuously checks COM1's UART status, as well as the status of the PC's keyboard. Whenever a key is pressed on the keyboard, it is transmitted to the single-board computer as soon as the transmitter in COM1 is ready. Likewise, if a character is received by COM1's UART, it is immediately output to the display screen (via the DOS interrupt INT 21H). So, full duplex communication is implemented through the SBCIO program. Please note that COM1's baud rate and other parameters must be set up before executing SBCIO. The following sample session illustrates many of the 8088 single-board computer's features. You may wish to try to duplicate the session on your computer. Keep in mind that all user inputs are in **boldface**:

1. Turn computer on. A greeting should appear:

```
8088 Monitor, Ver3.0
```

2. Consider the following test program:

```

1000  03 D8  ADD     BX,AX
1002  03 D1  ADD     DX,CX
1004  CD 95  INT     95H

```

3. To load the test program into memory, use the X command:

```

>X 1000
1000  C7?03
1001  06?D8
1002  74?03
1003  50?D1
1004  CD?<sp>
1005  00?95
1006  B2?<cr>
>

```

*Note:* The <sp> skips over the data in location 1004 (which is already correct) and the <cr> terminates the command.

4. Display the program now stored in memory:

```

>D 1000 1005
1000  03 D8 03 D1 CD 95
>

```

5. Load AX through DX with initial data to be passed to the program:

```

>E
AX - FF20?1111
BX - 7E36?2222
CX - 3352?3333
DX - A5E8?4444
etc.

```

## 6. Check the register contents:

```
>R
AX:1111 BX:2222 CX:3333 DX:4444
      etc.
```

## 7. Execute the program:

```
>G 1000
AX:1111 BX:3333 CX:3333 DX:7777
      etc.
Reentry by external program at [0000:1004]
```

*Note:* BX now equals BX + AX and DX now equals DX + CX. Also, the INT 95H instruction caused the registers to be saved and displayed upon reentry to the monitor.

## 8. Place a breakpoint at the beginning of the second instruction:

```
>B 1002
Breakpoint saved.
```

## 9. Check the user code again:

```
>D 1000 1005
1000 03 D8 CC D1 CD 95
```

*Note:* The "CC" opcode is the breakpoint interrupt. The user code "03" is now saved in system RAM, to be replaced after the breakpoint is encountered during execution.

## 10. Execute the program again:

```
>G 1000
AX:1111 BX:4444 CX:3333 DX:7777
      etc.
Breakpoint encountered at [0000:1002]
```

*Note:* By examining the register contents, we see that BX has been changed (by another addition with AX), but CX and DX remain unchanged. This was the intent of placing the breakpoint at address 1002.

## 11. Check the user code again:

```
>D 1000 1005
1000 03 D8 03 D1 CD 95
```

*Note:* The breakpoint service routine has restored the original instruction byte at address 1002.

## 12.8 TROUBLESHOOTING TECHNIQUES

Here are some things we can do if the single-board computer does not work when we turn it on:

- Feel around the board for hot components. A chip that is incorrectly wired or placed backwards in its socket can get very hot. You may even smell the hot component.
- Make sure all the ICs have power by measuring with a DMM or oscilloscope. Put the probe right on the pin of the IC, not on the socket lead.
- Use an oscilloscope to examine the CLK output of the 8284. Push the RESET button to verify that the RESET signal is being generated properly.

- Look at the address, data, and control lines with an oscilloscope or logic analyzer. Activity is a good sign; there may be something as simple as a missing address or data line, or crossed lines. No activity means the processor is not receiving the right information. By examining the logic analyzer traces, you should be able to determine if the memory and I/O address decoders are working properly, as well as the address and data bus drivers.
- Verify that the EPROM was burned correctly and is in the right socket and not switched with the RAM. You should be able to connect a logic analyzer to verify that the processor fetches the first instruction from address FFFF0H. You should also be able to see the first instruction byte come out of the EPROM as well.
- Examine the TxD output of the 8251. Activity at power-on or RESET is a good sign, since the monitor program is designed to output a short greeting to let us know it is alive. If TxD wiggles around, but the serial output of the MAX232 does not, there could be a wiring problem there. Putting the capacitors in backward is bad for the MAX chip.
- Check every connection again from a fresh schematic. Many times, a missing connection is found, even when every attempt was made to be careful during construction.
- Change all the chips, one by one. Look for bent or missing pins when you remove them.
- When all else fails, tell someone else everything you've done and see if they can suggest anything else.
- You could also set the project aside for a while to get your mind off it. The problem may present itself to you when you least expect it. You may suddenly remember that you commented out an important I/O routine in the monitor because you were having trouble with the assembler. Silly things like that are really fun to find.

---

## SUMMARY

This chapter dealt with the hardware and software design of a minimal 8088 maxmode single-board computer. The system uses 8KB of EPROM, 8KB of RAM, and has a 2,400-baud serial I/O section. In addition, a parallel device is included to provide analog I/O capability.

Many design ideas were suggested, and reasons for choosing one idea over another were given.

The software monitor program is composed of many auxiliary routines, each of which performs one task. The monitor command routines selectively call the auxiliary routines, resulting in shorter, more logical code.

The software monitor is by no means complete and may be improved by adding commands or features after the basic system is up and running.

---

## STUDY QUESTIONS

1. Modify the circuitry of Figure 12.2 so that the entire address bus can be tri-stated by a 0 placed on a control signal called UNBUS.
2. What changes must be made to the memory section (see Figure 12.3) to allow the use of 16KB 27128 EPROMs? Assume that a total of four 27128s will be used.

3. What then will be the address range for each memory in Question 2?
4. How can the 74LS138 memory-address decoder be disabled?
5. Design a selector circuit that will allow a single switch to select either 300- or 2,400-baud operation in the serial device.
6. Redesign the port-address decoder in Figure 12.4 so that the 74LS138 is enabled only when  $\overline{IORC}$  or  $\overline{IOWC}$  is low.
7. Redesign the port-address decoder to decode port addresses in the range 00?? only (i.e., the upper byte of the address must be 0).
8. Design a parallel circuit (using an 8255) that has LEDs on port B, switches on port A, and a common-cathode seven-segment display on port C. Each LED must turn on when its associated bit is high. An open switch indicates a logic 1 level. The LSB of port C drives segment A of the display. Each successive bit drives the next segment. The base port address is E0.
9. Write the software necessary to initialize the parallel circuit of Question 8. All LEDs should be off and the seven-segment display should be showing a U (for "up and running").
10. Write an auxiliary routine that will display the decimal equivalent (0 to 255) of the contents of register AL.
11. Repeat Question 10 for the value contained in DX. The decimal range is now 0-65,535.
12. Modify C\_OUT so that it automatically calls CRLF if register AL contains 0DH on entry.
13. Using the auxiliary routines, complete the following table:

| <i>Command</i> | <i>Inputs</i> | <i>Outputs</i> | <i>Registers Used</i> |
|----------------|---------------|----------------|-----------------------|
| GET_WRD        | -             | AX             | AX, BH, [GET_BYT]     |

The routine name GET\_BYT in square brackets means that it is called by GET\_WRD. Show a table entry for each auxiliary routine.

14. Write an auxiliary routine to accept a multidigit decimal number from the keyboard and place its binary equivalent into register AX. Give an error message if the number is greater than 65,535.
15. What changes must be made to allow the command recognizer to recognize entire names for commands? For example, instead of D, use DUMP or DISP. For G, use EXEC or GO.
16. Show the changes needed to add the command V to the monitor program. The V command needs no inputs. The command routine associated with V is V\_VERIFY.
17. Write a command routine called V\_VERIFY that will test all available RAM locations. It will display the address, data written, and data read for any locations that fail the test.
18. Write a command routine that will calculate the checksum of RAM and EPROM and display the results in hexadecimal. The checksum is obtained by adding up all bytes in the desired memory range, ignoring carries.
19. Write a command routine to add and subtract two supplied hexadecimal numbers.
20. How can the breakpoint routine be modified to allow for more than one breakpoint at a time?
21. What other formats for displaying memory contents might be useful?
22. Modify the DUMP routine so that the display is paused if Control-S is entered from the keyboard. A paused display is restarted by pressing any key.
23. Rewrite the INIT routine so that a single register may be changed by naming it in the command. For example, E CX will only allow register CX to be updated.

24. What instructions must be added to the E\_XEC command routine so that any address outside the range 400H to 1F00H causes an error message?
25. Modify the port-input routine so that the data at the input port is continuously updated until <cr> is entered.
26. Write an interrupt service routine for INT 80H. Modify the monitor's start-up code to initialize the INT 80H vector table entry to point to the D\_ISPAT routine. The D\_ISPAT interrupt service routine uses the number in register AL to call a specific auxiliary command, according to the following table:

| <i>AL Value</i> | <i>Routine Called</i> |
|-----------------|-----------------------|
| 00              | C_IN                  |
| 01              | C_OUT                 |
| 02              | S_END                 |
| 10              | BLANK                 |
| 11              | CRLF                  |
| 20              | GET_BYT               |
| 28              | GET_WRD               |
| 2F              | GET_NUM               |
| 40              | CH_CASE               |
| 80              | DPR                   |

27. Compute the approximate frequency of the sine wave output during the analog test procedure.
28. Assemble this code fragment using DEBUG. Execute it with the monitor commands. What are the final results?

```

MOV  AX,1234H
MOV  BX,5678H
ADD  AL,BL
XOR  AH,BH

```

29. What monitor commands would you deem nonessential? Explain your reasons for doing so.
30. What monitor commands have not been discussed at all (or mentioned in the study questions)? What are the requirements of any additional commands you can think of? Is there enough room in the 2764 EPROM to include them?
31. How does the SBCIO program poll the UART on COM1?



# PART 5

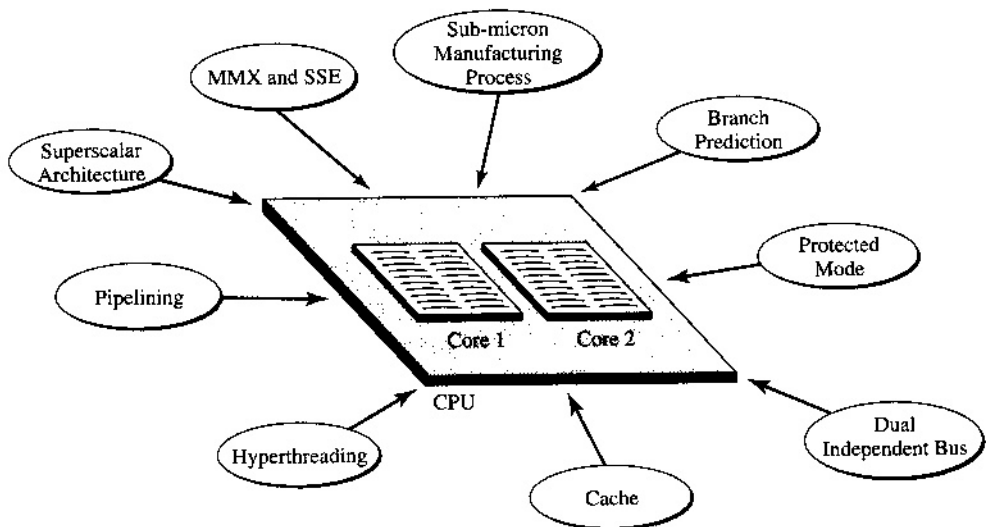
## Advanced Topics

---

**13** Hardware Details of the Pentium

**14** Protected-Mode Operation

**15** The Pentium II and Beyond



Many advanced techniques combine to provide modern microprocessors with high performance

---

# CHAPTER 13

---

## Hardware Details of the Pentium

---

### OBJECTIVES

In this chapter you will learn about:

- The operation of the Pentium's hardware signals
- The philosophy behind RISC designs
- The various types of bus cycles supported by the Pentium
- The Pentium's pipelined superscalar architecture
- Branch prediction
- The operation of the Pentium's code and data cache
- The Pentium's floating-point unit

### KEY TERMS

|                         |                         |                       |
|-------------------------|-------------------------|-----------------------|
| Atomic operation        | History bits            | Second-level cache    |
| Branch prediction       | Hit ratio               | Semaphore             |
| Bus snooping            | Intel debug port        | Set-associative cache |
| Cache                   | Least recently used     | Superscalar machine   |
| Cache coherency         | algorithm               | Target address        |
| Complex instruction set | Line of data            | Translation lookaside |
| computer (CISC)         | MESI protocol           | buffer                |
| Debug register          | Pipelining              | Triple ported         |
| Floating-point register | Probe mode              | Writeback             |
| Flush acknowledge       | Reduced instruction set | Writethrough          |
| cycle                   | computer (RISC)         |                       |

---

## 13.1 INTRODUCTION

To fully understand the operation of the Pentium, it is necessary to examine the operation of its hardware architecture. Our experience with the hardware architecture of the 8088 and 8086 microprocessors in Chapters 8 through 12 sets the stage for the material presented here. In this chapter, we will see how the Pentium operates from a hardware perspective, a dramatic contrast to the 8088 and 8086.

Learning to design a Pentium-based motherboard for a computer system is *not* the goal of this chapter. Instead, the concentration will be on how the Pentium does what it does. The operation of the U- and V-pipelines, the code and data cache, branch prediction—all of the tricks the designers used to squeeze more performance out of the Pentium—will be explored. The goal is to get a feel for the overall Pentium package and how its various internal components work together.

Section 13.2 presents a detailed discussion of the processor signals. This is followed by an introduction to the techniques used in designing a RISC processor in Section 13.3. The various types of bus cycles supported by the Pentium are covered in Section 13.4. Superscalar architecture and pipelining are examined next, in Sections 13.5 and 13.6, respectively. Section 13.7 explains how branch prediction is used to keep the instruction pipelines flowing smoothly. A detailed look at the processor's code and data cache is found in Section 13.8. This is followed by a discussion of the redesigned floating-point unit in Section 13.9. The chapter concludes with a set of troubleshooting tips in Section 13.10.

## 13.2 CPU PIN DESCRIPTIONS

Figure 13.1 shows a top view of the 60-MHz and 66-MHz Pentium packages. The package is a 273-pin PGA (pin grid array). The 60-MHz and 66-MHz Pentiums were the first ones introduced to the computing market.

Newer Pentiums with faster clock speeds and other enhancements use a different package, the 296-pin PGA shown in Figure 13.2. These Pentiums allow hardware control over the ratio of bus speed to processor speed, to support motherboard designs that may not be capable of operating at the clock speed of the processor. In addition, the power supply voltage is 3.3 V, compared to 5 V on the earlier 60- and 66-MHz versions. This helps reduce the power used by the processor.

### Pentium Hardware Signals

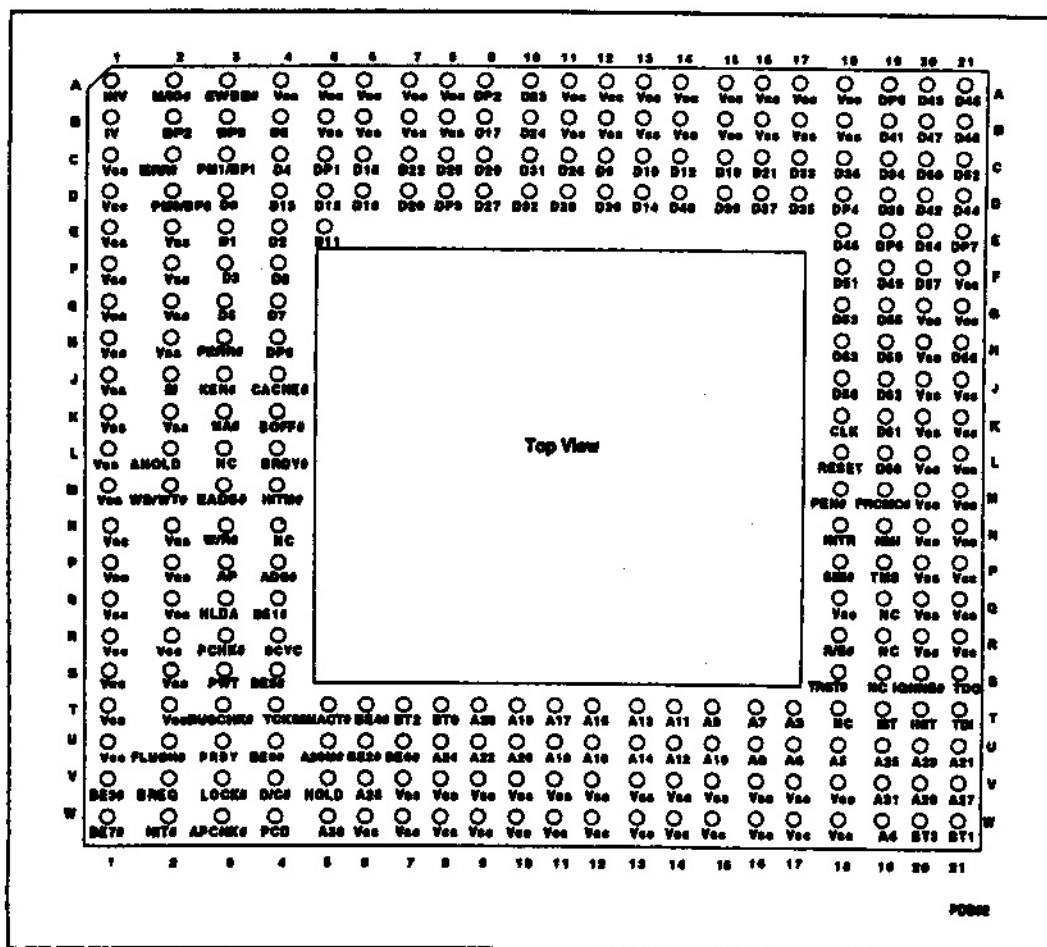
**$\overline{A20M}$  (Address 20 Mask).** This input is used to force the Pentium to limit addressable memory to 1MB, to emulate the memory space of the 8086.  $\overline{A20M}$  may only be active in real mode.

**$A_3$  Through  $A_{31}$  (Address Lines).** These twenty-nine address lines, together with the byte enable outputs, form the Pentium's 32-bit address bus. A memory space of 4,096MB (4 gigabytes) is possible, along with 65,536 I/O ports.

The address lines are used as inputs during an inquire cycle to read an address into the Pentium, for examination by the internal cache.

**$\overline{ADS}$  (Address Strobe).** The  $\overline{ADS}$  output, when low, indicates the beginning of a new bus cycle. Signals that define the new bus cycle are valid when  $\overline{ADS}$  is active. These signals include the address bus and byte enables, AP,  $\overline{CACHE}$ , LOCK, M/ $\overline{IO}$ , W/ $\overline{R}$ , D/ $\overline{C}$ , SCYC, PWT, and PCD.

**AHOLD (Address Hold).** This input is used to place the Pentium's address bus into a high impedance state so that an inquire cycle can be run. The address used during the inquire cycle is read in by the Pentium when AHOLD is active.



**FIGURE 13.1** Pentium® processor (510/60, 567/66) pinout (top view). Reprinted by permission of Intel Corporation. Copyright© 1995 Intel Corporation

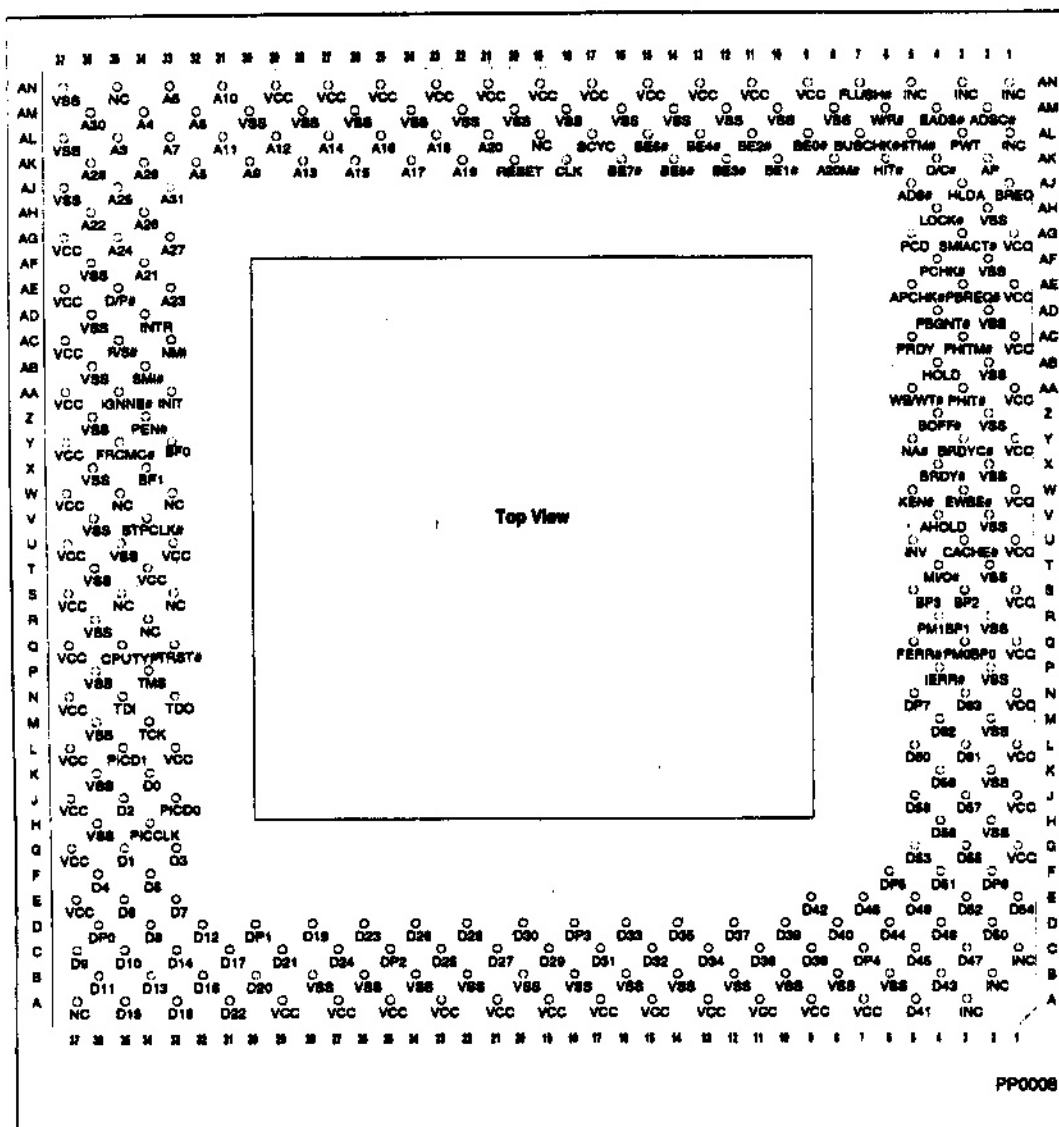
**AP (Address Parity).** This signal is bidirectional and is used to indicate the even parity of address lines  $A_5$  through  $A_{31}$ . AP is used as an output when the Pentium is outputting an address, and as an input during an inquire cycle.

**APCHK (Address Parity Check).** This output goes low if the Pentium discovers a parity error on the address lines. External circuitry is responsible for taking the appropriate action if a parity error is encountered.

**APICEN (Advanced Programmable Interrupt Controller Enable).** This input is used to enable or disable the Pentium's internal APIC interrupt controller circuitry. APICEN is sampled when the processor is reset.

**$\overline{BE}_0$  Through  $\overline{BE}_7$ .** These 8-byte enable outputs, together with  $A_3$  through  $A_{31}$ , make up the 32-bit address output by the Pentium. Each byte enable is used to control a different 8-bit portion of the processor's 64-bit data bus. Table 13.1 shows the purpose of each byte enable.

**$BF_0$ ,  $BF_1$  (BUS Frequency).** These inputs, sampled during reset, control the ratio of bus frequency to CPU core frequency. The bus/core ratio is 2/3 when  $BF_0$  is high or floating. When low, the ratio becomes 1/2.



**FIGURE 13.2** Pentium® processor (610\75, 735\90, 815\100, 1000\120, 1110\133) pinout (top view). Reprinted by permission of Intel Corporation. Copyright© 1995 Intel Corporation

**TABLE 13.1** Byte enable operation

| Output            | Data Bus Byte Enabled            |
|-------------------|----------------------------------|
| $\overline{BE}_0$ | D <sub>0</sub> –D <sub>7</sub>   |
| $\overline{BE}_1$ | D <sub>8</sub> –D <sub>15</sub>  |
| $\overline{BE}_2$ | D <sub>16</sub> –D <sub>23</sub> |
| $\overline{BE}_3$ | D <sub>24</sub> –D <sub>31</sub> |
| $\overline{BE}_4$ | D <sub>32</sub> –D <sub>39</sub> |
| $\overline{BE}_5$ | D <sub>40</sub> –D <sub>47</sub> |
| $\overline{BE}_6$ | D <sub>48</sub> –D <sub>55</sub> |
| $\overline{BE}_7$ | D <sub>56</sub> –D <sub>63</sub> |

**$\overline{BOFF}$  (Backoff).** This input causes the processor to terminate any bus cycle currently in progress and tri-state its buses. Execution of the interrupted bus cycle is restarted when  $\overline{BOFF}$  goes high.

**$PM/BP_0$ ,  $PM/BP_1$ ,  $BP_2$  and  $BP_3$ .** The BP (breakpoint) outputs are associated with a set of internal registers called **debug registers**. There are eight debug registers. The first four (DR0 through DR3) are used to store the memory or I/O address of a program *breakpoint*. A breakpoint is a predefined address that the programmer chooses to help determine how a program executes. For example, a breakpoint may be set to address 1000H to see if the program ever reads or writes to that address. The breakpoint outputs go high when the breakpoint address in its respective debug register matches a program-generated address. Two PM (performance monitoring) outputs are multiplexed with the lower two BP outputs. These signals are active after a reset and indicate the status of two internal performance monitoring counters. Setting the DE (debug extensions) bit in CR4 (control register 4) causes these two outputs to change to  $BP_0$  and  $BP_1$ . We will examine the operation of the debug and control registers in Chapter 14.

**$\overline{BRDY}$  (Burst Ready).** During a read cycle, this input indicates that data is available on the data bus. For write cycles,  $\overline{BRDY}$  informs the processor that the output data has been stored.  $\overline{BRDY}$  is used for both memory and I/O operations. If  $\overline{BRDY}$  is not low when sampled, the Pentium inserts extra clock cycles into the current cycle (*wait states*) to provide extra time for the data transfer.

**$BREQ$  (Bus Request).** This output goes low when the Pentium has a pending bus cycle ready to begin. In a multiprocessor system,  $BREQ$  may be used to select a processor competing for the system bus.

**$BT_0$  through  $BT_3$  (Branch Trace).**  $BT_0$  through  $BT_2$  output the lower 3 bits ( $A_0$  through  $A_2$ ) of the target address of the currently executing branch instruction.  $BT_3$  indicates the operand size of the current instruction (0 for 16-bit, 1 for 32-bit).

**$\overline{BUSCHK}$  (BUS Check).** This input, when low, indicates to the Pentium that there was a problem with the last bus cycle. The processor may perform a machine check exception to recover.

**$\overline{CACHE}$  (Cacheability).** This output indicates whether the data associated with the current bus cycle is being read from or written to the internal cache.

**$CLK$  (Clock).** This is the clock input to the processor.  $CLK$  must be stable (running at the desired frequency) within 150 ms of power-on.

**$CPUTYP$  (CPU Type).** This input is used to specify the processor type in a dual-processor system. When low,  $CPUTYP$  specifies the primary processor. The dual processor is indicated when  $CPUTYP$  is high.

**$D/\overline{C}$  (Data/Code).** This output indicates that the current bus cycle is accessing code ( $D/\overline{C}$  is low) or data ( $D/\overline{C}$  is high).

**$D/\overline{P}$  (Dual/Primary).** This output indicates the processor type in a dual-processing system. When low, the processor is the primary processor. The dual processor is indicated when high.

**$D_0$  through  $D_{63}$  (Data Bus).** These signals make up the Pentium's 64-bit bidirectional data bus. The actual data lines in use during a particular bus cycle are indicated by the byte enable ( $\overline{BE}$ ) outputs.

**$DP_0$  through  $DP_7$  (Data Parity).** These bidirectional signals are used to indicate the even parity of each data byte on the data bus.  $DP_0$  represents the parity of the lower byte ( $D_0$  through  $D_7$ ).

**$\overline{DPEN}$  (Dual Processing Enable).** This signal is an output on the dual processor and an input on the primary processor in a dual-processing system. During reset,  $\overline{DPEN}$  is used to initiate dual processing.

**$\overline{EADS}$  (External Address Strobe).**  $\overline{EADS}$  is used to indicate that an external address may be read by the address bus during an inquire cycle.

**$\overline{EWBE}$  (External Write Buffer Empty).** This input, when low, indicates that the Pentium may proceed with the next cache writethrough operation.

**$\overline{FERR}$  (Floating-Point Error).** This output indicates that the processor's FPU generated an error.  $\overline{FERR}$  is included to maintain compatibility with the error-handling mechanism of MS-DOS.

**$\overline{FLUSH}$  (Flush Cache).** This input, when low, causes the Pentium to writeback all modified data lines in its internal code and data cache.

**$\overline{FRMC}$  (Functional Redundancy Checking Master/Checker).** This input is sampled during a RESET operation to determine whether the Pentium is operating as a master (when high) or a checker (when low). Two Pentiums may be connected in such a way that one is the master and the other is the checker. The checker checks every operation performed by the master to guarantee correct execution. A master/checker pair provides a measure of reliability to critical systems, such as flight control computers.

**$\overline{HIT}$  (Inquire Cycle Cache Hit/Miss).** This output indicates a cache hit (when low) as the result of an inquire cycle.

**$\overline{HITM}$  (Hit/Miss to a Modified Cache Line).** This output indicates that a modified line in the cache was hit as the result of an inquire cycle.

**$HLDA$  (Bus Hold Acknowledge).** The  $HLDA$  output goes high (as a result of a HOLD request) to indicate that the Pentium has been placed in a hold state. If the code and data cache contain current instructions and operands, execution continues with no bus activity, only cache accesses.

**$HOLD$  (HOLD Bus).** If  $HOLD$  is high when sampled, the Pentium tri-states its bus signals and activates  $HLDA$ .  $HOLD$  may be used by another processor that wishes to become the bus master.

**$IBT$  (Instruction Branch Taken).** This output goes high for one clock cycle whenever the processor takes a branch (a JNZ that jumps, for example).

**$\overline{IERR}$  (Internal Error).** This output, when low, indicates that a parity or redundancy check error has occurred. Parity errors may cause the Pentium to enter shutdown mode (see Section 13.4).

**$\overline{IGNNE}$  (Ignore Numeric Exception).** A zero on this input allows the processor to continue executing floating-point instructions, even if an error is generated.

**$INIT$  (Initialization).**  $INIT$  is a rising edge-sensitive input that causes the processor to be initialized in the same way as a RESET, except that the internal registers and cache are left unchanged.

**INTR (Interrupt Request).** This input, when high, causes the Pentium to initiate interrupt processing. An 8-bit vector number is read on the lower byte of the data bus to select an interrupt service routine. INTR is ignored if the interrupt enable bit in the flag's register is clear.

**INV (Invalidation Request).** During an inquire cycle, INV is used to determine what happens to a cache line during a hit. If INV is high, the cache line is invalidated. If INV is low, the line is marked shared. See Section 13.8 for details on shared, and other, cache line states.

**IU (U-Pipeline Instruction Complete).** This output goes high for one clock cycle each time an instruction completes in the U pipeline.

**IV (V-Pipeline Instruction Complete).** This output goes high for one clock cycle when an instruction completes in the V pipeline.

**$\overline{KEN}$  (Cache Enable).** A zero on this input enables caching of data being read into the processor. If  $\overline{KEN}$  is high when sampled, no caching will occur.

**$\overline{LOCK}$  (Bus Lock).** This output goes low to indicate that the current bus cycle is locked and may not be interrupted by another bus master.

**$\overline{M/\overline{IO}}$  (Memory/Input-Output).** This output indicates the type of bus cycle currently starting. If  $\overline{M/\overline{IO}}$  is high, a memory cycle is beginning; otherwise an IO operation is performed.

**$\overline{NA}$  (Next Address).** This input, when low, indicates that the external memory system is capable of performing a pipelined access (two bus cycles in progress at the same time).

**NMI (Nonmaskable Interrupt).** The Pentium responds to this rising edge input by issuing interrupt vector 2. No external interrupt acknowledge cycles are generated.

**$\overline{PBGNT}$  (Private Bus Grant).** This signal is used in a dual-processing system to indicate when private bus arbitration is allowed.

**$\overline{PBREQ}$  (Private Bus Request).** This signal is used to request a private bus operation in a dual-processing system.

**PCD (Page Cacheability Disable).** This output indicates the state of CR3's PCD (page cache disable) bit. It is used to control cacheability on a page-by-page basis. More details on PCD will be covered in Chapter 14.

**$\overline{PCHK}$  (Data Parity Check).** This output goes low if the Pentium detects a parity error on the data bus. External hardware must take the appropriate action to handle the error.

**$\overline{PEN}$  (Parity Enable).** If this input is low during the same cycle a parity error is detected, the Pentium will save a copy of the address and control signals in an internal machine check register.

**$\overline{PHIT}$  (Private Hit).** When operating in a dual-process or system,  $\overline{PHIT}$  is used to help maintain local cache coherency.

**$\overline{PHITM}$  (Private Modified Hit).** Used in conjunction with  $\overline{PHIT}$  to maintain local cache coherency in a dual-processing system.

**PICCLK (Programmable Interrupt Controller Clock).** This input controls the serial data rate in the internal APIC interrupt controller.

**PICD<sub>0</sub>, PICD<sub>1</sub> (Programmable Interrupt Controller Data).** These two signals are used to exchange data with the internal APIC interrupt controller.



**PRDY (Probe Ready).** This output is used for debugging purposes. It indicates that the processor has stopped normal execution and is entering a debugging mode called **probe mode**, where it may execute debugging instructions. PRDY goes high in response to activity on  $R/\overline{S}$ .

**PWT (Page Writethrough).** This output indicates the status of the cache's writethrough paging bit in CR3.

**$R/\overline{S}$ .** This negative edge-sensitive input places the Pentium in a wait state and possibly executes instructions in probe mode. It is only used for debugging with the **Intel debug port**, a specific hardware debugging interface.

**RESET (Processor Reset).** The RESET input causes the Pentium to initialize its registers to known states, invalidate the code and data cache, and fetch its first instruction from address FFFFFFF0H. RESET must be active for at least 1 ms after power-on to ensure proper operation.

**SCYC (Split Cycle Indication).** This output goes high to indicate that two or more locked bus cycles are performing a misaligned transfer. A misaligned transfer involves a 16- or 32-bit operand that is stored at a starting address that is not a multiple of four, or a 64-bit operand that does not begin at an address that is a multiple of eight.

**$\overline{SMI}$  (System Management Interrupt).** This negative edge-sensitive input is used to generate a system management interrupt. System management is used to perform special functions, such as power management.

**$\overline{SMI}ACT$  (System Management Interrupt Active).** The  $\overline{SMI}ACT$  output goes low in response to an  $\overline{SMI}$  request and remains low until the processor leaves system management mode.

**$\overline{STPCLK}$  (Stop Clock).** When low, this input causes the Pentium to stop its internal clock (thus reducing the power consumed).

The next five inputs are all associated with the Intel debug port.

**TCK (Test Clock).** This input is used to clock data into and out of the Pentium when performing a specific test procedure called a *boundary scan* (IEEE Standard 1149.1).

**TDI (Test Data Input).** This input allows serial test data to be input into the Pentium. Data is clocked in on the rising edge of TCK.

**TDO (Test Data Output).** Serial test information is clocked out of TDO on the falling edge of TCK.

**TMS (Test Mode Select).** This input is used to control the sequencing of boundary scan tests.

**$\overline{TRST}$  (Test Reset).** This input, when low, resets the test controller logic.

**$W/\overline{R}$  (Write/Read).** This output is used to indicate if the current bus cycle is a read operation ( $W/\overline{R}$  is low) or a write operation ( $W/\overline{R}$  is high).

**$WB/\overline{WT}$  (Writeback/Writethrough).** This input defines the current cache line update protocol as writeback or writethrough. Cache lines may be defined on a line-by-line basis with  $WB/\overline{WT}$ .

Clearly, the hardware operation of the Pentium is complex, as would be any system designed around it. Only experienced computer engineers, who know all the tricks of the trade, can successfully design a high-speed motherboard based on the Pentium.

## 13.3 RISC CONCEPTS

The last decade has delivered a great deal of change in the area of computer architecture, due to advances in microelectronic manufacturing technology. As more and more logic was crammed into the small space of the silicon wafer, computer architects were able to implement processors with increasingly complex instructions and addressing modes. In general, these processors fall into a category called **CISC**, for **complex instruction set computer**. Unfortunately, cramming so much logic into a single package leads to a performance bottleneck, because many CISC instructions require multiple clock cycles to execute.

One solution to the performance bottleneck was the emergence of a new design philosophy called **RISC**, for **reduced instruction set computer**. RISC designers chose to make the instruction sets *smaller*, using fewer instructions and simpler addressing modes. This reduced set of operations was easier to implement on silicon, resulting in faster performance. Coupled with other architectural differences and improvements, RISC gained popularity, acceptance, and commercial interest. Video games and laser printers now boast of their internal RISC processors.

In general, a RISC machine is designed with the following goals in mind:

- Reduce accesses to main memory.
- Keep instructions and addressing modes simple.
- Make good use of registers.
- Pipeline everything.
- Utilize the compiler extensively.

Let us briefly examine each goal, with the idea that we will compare the Pentium processor against each goal as we go.

### Reduce Accesses to Main Memory

Even with the improvements made in memory technology, processors are much faster than memories. For example, a processor clocked at 100 MHz would like to access memory in 10 ns, the period of its 100-MHz clock. Unfortunately, the memory interfaced to the processor might require 60 ns for an access. So, the processor ends up waiting during each memory access, wasting execution cycles.

To reduce the number of accesses to main memory, RISC designers added instruction and data cache to the design. A **cache** is a special type of high-speed RAM where data and the *address* of the data are stored. Whenever the processor tries to read data from main memory, the cache is examined first. If one of the addresses stored in the cache matches the address being used for the memory read (called a *hit*), the cache will supply the data instead. Cache is commonly ten times faster than main memory, so you can see the advantage of getting data in 10 ns instead of 60. Only when we *miss* (i.e., do not find the required data in the cache), does it take the full access time of 60 ns. But this can only happen once, because a copy of the new data is written into the cache after a miss. The data will be there the next time we need it.

Instruction cache is used to store frequently used instructions, such as those in a short loop. Data cache is used to store frequently used data. Each cache is initially empty and fills as a program executes. The Pentium employs an 8KB instruction cache and an 8KB data cache. We will examine their operation in detail in Section 13.8

## Keep Instructions and Addressing Modes Simple

Computer scientists learned, after studying the operation of many different types of programs, that programmers use only a small subset of the instructions available on the processor they are using. The same is true for the processor's addressing modes.

Implementing fewer instructions and addressing modes on silicon reduces the complexity of the instruction decoder, the addressing logic, and the execution unit. This allows the machine to be clocked at a faster speed, because less work needs to be done each clock period.

Unfortunately, the Pentium processor cannot meet this design goal. In order to remain compatible with the installed software of the entire 80x86 family, the Pentium designers had to keep each and every instruction and addressing mode of the previous machine, the 80486. From this point of view, the Pentium looks like a CISC machine.

## Make Good Use of Registers

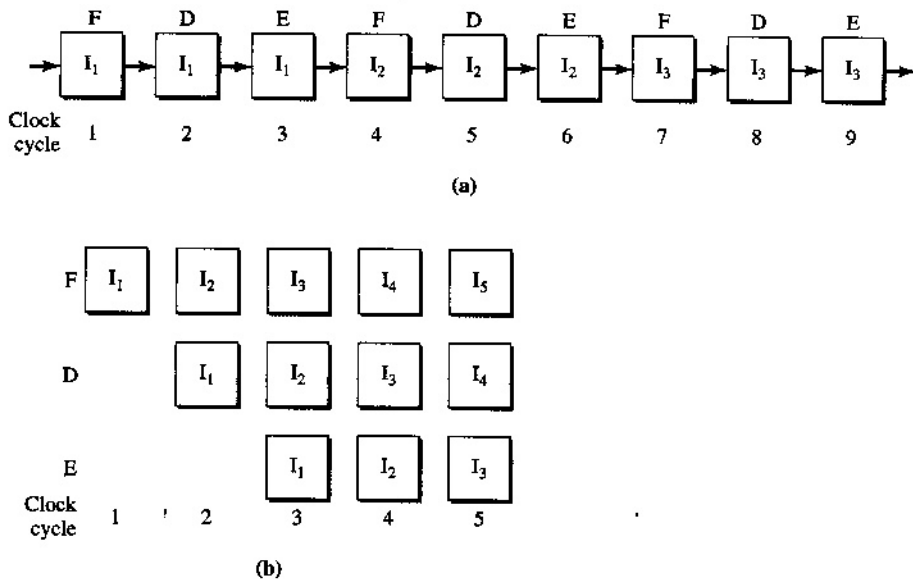
RISC machines typically have large sets of registers. The number of registers available in a processor can affect performance the same way a memory access does. A complex calculation may require the use of several data values. If the data values all reside in memory during the calculations, many memory accesses must be used to utilize them. If the data values are stored in the internal registers of the processor instead, their access during calculations will be much faster. It is good, then, to have lots of internal registers.

The 8-bit processors of the late 1970s had few registers (two on the 6800 and seven on the 8085) for general purpose use. These registers were also only 8 bits wide. The Pentium has seven general purpose registers (EAX, EBX, etc.), all of which are 32 bits wide. This is four times the internal register storage space of the 8085. The Pentium also contains six 16-bit segment registers, a 32-bit stack pointer, and eight **floating-point registers**, each *80 bits* wide, which are used by the floating-point unit (covered in Section 13.9). So the Pentium has a fairly large set of registers to work with.

## Pipeline Everything

**Pipelining** is a technique used to enable one instruction to complete with each clock cycle. Compare the two three-instruction sequences in Figure 13.3. On a nonpipelined machine, nine clock cycles are needed for the individual fetch, decode, and execute cycles. On a pipelined machine, where fetch, decode, and execute operations are performed *in parallel*, only five cycles are needed to execute the same three instructions. The first instruction requires three cycles to complete. Additional instructions then complete at a rate of one per cycle. As Figure 13.3(b) indicates, during clock cycle 5 we have  $I_3$  completing,  $I_4$  being decoded, and  $I_5$  being fetched. A long sequence of instructions, say 1,000 of them, might require 3,000 clock cycles on a nonpipelined machine, and only 1,002 clock cycles when pipelined (three cycles for the first instruction, and then one cycle each for the remaining 999). So, pipelining results in a tremendous performance gain.

The Pentium employs two types of pipelines. These are the instruction pipelines (U and V pipelines, covered in Section 13.6), and a pipeline that performs special types of bus cycles. The instruction pipelines are five-stage pipelines and are capable of independent operation. The U pipeline also forms the first five stages of the eight-stage floating-point pipeline. Certain types of bus cycles, such as back-to-back burst reads and writes, are possible through pipelined addressing logic.



**FIGURE 13.3** Execution of three instructions: (a) nonpipelined; (b) pipelined

Furthermore, the Pentium employs a technique called **branch prediction** that helps identify possible interruptions to the normal flow of instructions through the U and V pipelines. By predicting which instructions might branch and change program flow, it is possible to keep a steady stream of instructions flowing into the pipelines. Once again, an attempt is made to keep instructions completing at a rate of one per clock cycle. The Pentium is very like a RISC machine in this respect.

### Utilize the Compiler Extensively

When a high-level language program (such as a C program) is compiled, the individual statements within the program's source file are converted into assembly language instructions or groups of instructions. A Pentium compiler, if written properly, can perform many optimizations on the assembly language code to take advantage of the Pentium's architectural advances. For instance, the compiler may reorder certain pairs of instructions to allow them to execute in parallel in the floating-point unit or dual-integer pipelines. Instructions may also be rearranged to take advantage of the Pentium's branch prediction strategy. Other optimizations may involve use of the instruction/data cache, or algorithms to allocate the minimum number of processor registers during parsing of an arithmetic statement. Sometimes it is possible to substitute an instruction (such as `MOV EAX,0`) with an equivalent instruction (such as `SUB EAX,EAX`) to reduce the number of clock cycles or the number of bytes of machine code.

Overall, the compiler plays an important role in helping a CISC or RISC machine achieve high performance. As a matter of fact, programs written for earlier 80x86 machines, even the 80486, can be improved by recompiling their sources with a Pentium compiler.

### RISC Summary

What we have seen is that the Pentium contains both CISC and RISC characteristics. This is due in part to Intel's commitment to support for all 80x86 users. Through pipelining,

branch prediction, instruction and data cache, and clever compilation, the Pentium delivers impressive speed. In fact, the Pentium is almost twice as fast as the 80486DX when clocked at identical speeds.

## 13.4 BUS OPERATIONS

The Pentium processor performs a number of different operations over its address and data buses. Data transfers (both single-cycle and burst transfers), interrupt acknowledge cycles, inquire cycles for examining the internal code and data cache, and I/O operations are just some of the operations possible. In this section we will examine the basic operation and purpose of each type of bus cycle.

### Decoding a Bus Cycle

The Pentium bus logic indicates the type of bus cycle currently starting with the use of its cycle definition signals. These are the  $\overline{M/\overline{IO}}$ ,  $\overline{D/\overline{C}}$ ,  $\overline{W/\overline{R}}$ ,  $\overline{CACHE}$ , and  $\overline{KEN}$  signals. Table 13.2 shows how these signals define the current bus cycle.

*Special cycle* bus cycles require additional decoding and use the byte enable outputs for selection. Table 13.3 defines these types of bus cycles.

The memory system must be designed to respond to each type of bus cycle.

**TABLE 13.2** Bus cycle encodings

| $\overline{M/\overline{IO}}$ | $\overline{D/\overline{C}}$ | $\overline{W/\overline{R}}$ | $\overline{CACHE}$ | $\overline{KEN}$ | Cycle Description                     |
|------------------------------|-----------------------------|-----------------------------|--------------------|------------------|---------------------------------------|
| 0                            | 0                           | 0                           | 1                  | X                | Interrupt acknowledge                 |
| 0                            | 0                           | 1                           | 1                  | X                | Special cycle                         |
| 0                            | 1                           | 0                           | 1                  | X                | I/O read non-cached                   |
| 0                            | 1                           | 1                           | 1                  | X                | I/O write non-cached                  |
| 1                            | 0                           | 0                           | 1                  | X                | Code read 8 bytes non-cached          |
| 1                            | 0                           | 0                           | X                  | 1                | Code read 8 bytes non-cached          |
| 1                            | 0                           | 0                           | 0                  | 0                | Code read 32 bytes burst cached       |
| 1                            | 1                           | 0                           | 1                  | X                | Memory read up to 8 bytes non-cached  |
| 1                            | 1                           | 0                           | X                  | 1                | Memory read up to 8 bytes non-cached  |
| 1                            | 1                           | 0                           | 0                  | 0                | Memory read 32 bytes burst cached     |
| 1                            | 1                           | 1                           | 1                  | X                | Memory write up to 8 bytes non-cached |
| 1                            | 1                           | 1                           | 0                  | X                | 32 byte cache writeback burst         |

Note: X = don't care

**TABLE 13.3** Special bus cycles

| $\overline{BE}_7$ | $\overline{BE}_6$ | $\overline{BE}_5$ | $\overline{BE}_4$ | $\overline{BE}_3$ | $\overline{BE}_2$ | $\overline{BE}_1$ | $\overline{BE}_0$ | Special Bus Cycle    |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|----------------------|
| 1                 | 1                 | 1                 | 1                 | 1                 | 1                 | 1                 | 0                 | Shutdown             |
| 1                 | 1                 | 1                 | 1                 | 1                 | 1                 | 0                 | 1                 | Flush cache          |
| 1                 | 1                 | 1                 | 1                 | 1                 | 0                 | 1                 | 1                 | Halt                 |
| 1                 | 1                 | 1                 | 1                 | 0                 | 1                 | 1                 | 1                 | Writeback            |
| 1                 | 1                 | 1                 | 0                 | 1                 | 1                 | 1                 | 1                 | Flush acknowledge    |
| 1                 | 1                 | 0                 | 1                 | 1                 | 1                 | 1                 | 1                 | Branch trace message |

## Bus Cycle States

There are six possible states the Pentium bus may be in, depending on what type of cycle is being processed. These states are Ti, T1, T2, T12, T2P, and TD. Ti is the idle state and indicates that no bus cycle is currently running. The bus begins in this state after a hardware reset. T1 and T2 are the first and second states of a bus cycle. During T1, a valid address is output on the address lines and  $\overline{ADS}$  is made active (low). Data is read or written during T2, and the  $\overline{BDY}$  input is examined.

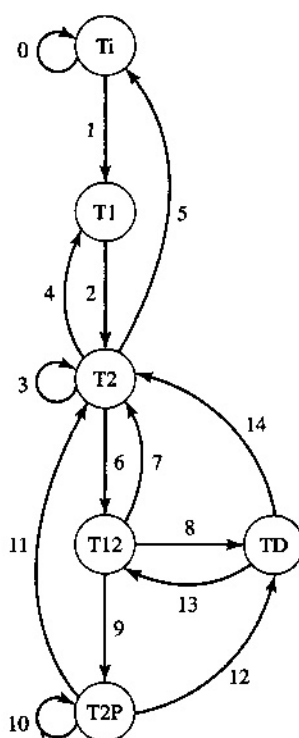
When a second bus cycle is started before the first one completes, the bus enters the T12 state. Data for the first bus cycle is transferred, and a new address is output on the address lines. State T2P continues the bus cycle started in T12. TD is used to insert a *dead* state between two consecutive cycles (read followed by write or vice versa), in order to give the system bus time to change states.

The bus state controller follows a predefined set of transitions to switch from state to state. Figure 13.4 shows these transitions in the form of a state diagram.

The transitions between states are defined as follows:

- 0: No bus cycle requested.
- 1: New bus cycle started.  $\overline{ADS}$  is taken low.
- 2: Second clock cycle of current bus cycle.
- 3: Stay in T2 until  $\overline{BDY}$  is active or new bus cycle is requested.
- 4: Go back to T1 if a new request is pending.
- 5: Bus cycle complete; go back to idle state.
- 6: Begin second bus cycle.
- 7: Current cycle is finished and no dead clock is needed.

FIGURE 13.4 Bus state transitions



- 8: A dead clock is needed after the current cycle is finished.
- 9: Go to T2P to transfer data.
- 10: Wait in T2P until data is transferred.
- 11: Current cycle is finished and no dead clock is needed.
- 12: A dead clock is needed after the current cycle is finished.
- 13: Begin a pipelined bus cycle if NA is active.
- 14: No new bus cycle is pending.

Clearly, a lot of thought went into the design of the Pentium's bus controller.

### Single-Transfer Cycle

This cycle is used to transfer up to 8 bytes of non-cacheable data between the processor and memory. The byte enable outputs indicate how many bytes will be transferred. The timing for single-transfer read and write operations is shown in Figure 13.5.

The cycle begins during clock cycle T1, when  $\overline{ADS}$  goes low.  $\overline{CACHE}$  is taken high to indicate to external circuitry that the data is not going to, or coming from, the internal cache. If  $\overline{BDRY}$  goes low during the T2 clock cycle, the data will be transferred and the operation completes during clock cycle Ti.

If  $\overline{BDRY}$  is not low during T2, additional T2 clock cycles are generated. These extra clock cycles are called *wait states* and are used to give the memory and I/O systems additional time to complete the read or write request.

### Burst Cycles

The Pentium supports burst reads and writes of 32 bytes. The cache uses burst cycles for line loads and writebacks. During a burst operation, a new 8-byte chunk can be transferred every clock cycle. The processor supplies the starting address of the first group of 8 bytes at the beginning of the cycle. The next three groups of 8 bytes are transferred according to the burst order shown in Table 13.4.

The external memory system must generate the remaining three addresses itself and supply the data in the correct order. Figure 13.6 shows the timing for a cacheable burst read cycle. Note that an 8-byte data chunk is transferred during every T2 clock cycle.

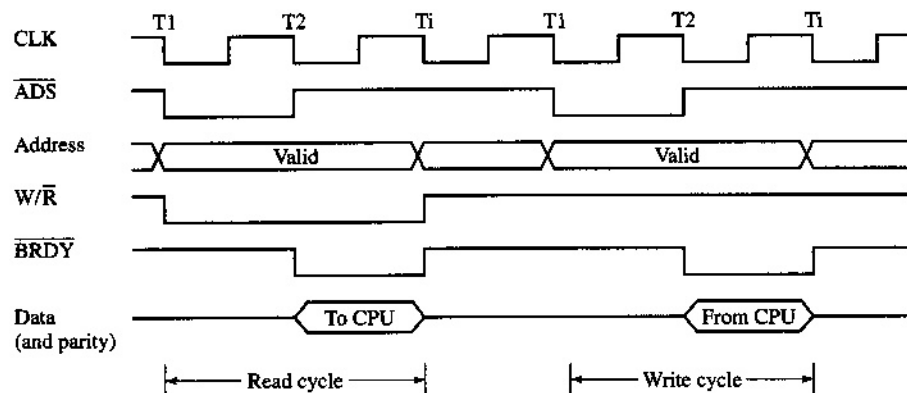
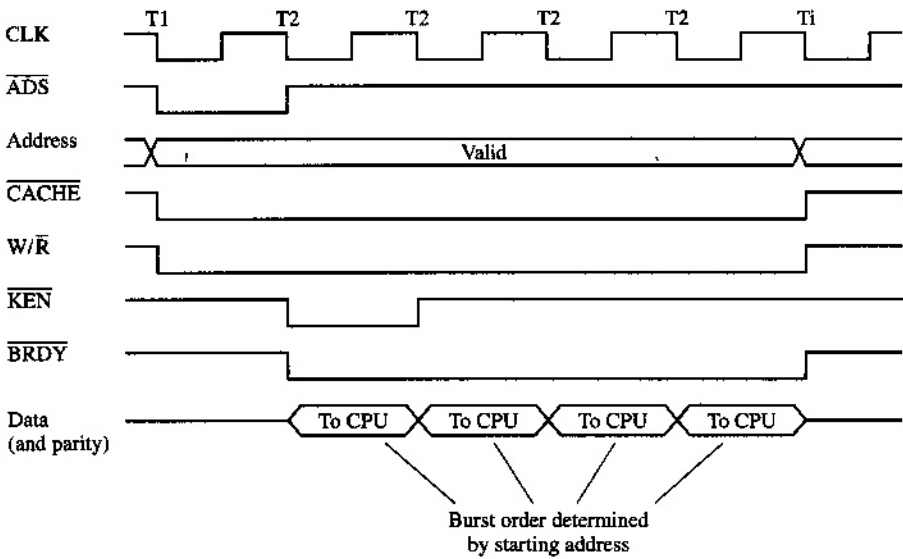


FIGURE 13.5 Single-transfer read/write cycles

**TABLE 13.4** Burst transfer order

| 1st Address | 2nd Address | 3rd Address | 4th Address |
|-------------|-------------|-------------|-------------|
| 0           | 8           | 10          | 18          |
| 8           | 0           | 18          | 10          |
| 10          | 18          | 0           | 8           |
| 18          | 10          | 8           | 0           |



**FIGURE 13.6** Burst read cycle

### Locked Operations

Many operating system processes depend on what is termed *atomic access* to data stored in memory. An **atomic operation** cannot be broken down into smaller suboperations. The data accessed during the atomic operation often comes in the form of a **semaphore**—a special type of counter variable that must be read, updated, and stored in one single, uninterruptible operation. This requires a read cycle followed by a write cycle. No other devices may take over control of the processor buses while the atomic operation is taking place. This is what it means to “lock” the bus. Some instructions, like XCHG, automatically lock the bus when one of their operands is a memory operand.

### BOFF

The **BOFF** input provides a way for other processors in a multiprocessor system to instantly take over the Pentium’s buses. The Pentium samples **BOFF** every clock cycle and, if low, puts its buses into a high-impedance state, beginning with the next clock cycle. Execution resumes with the interrupted cycle when **BOFF** is returned high.



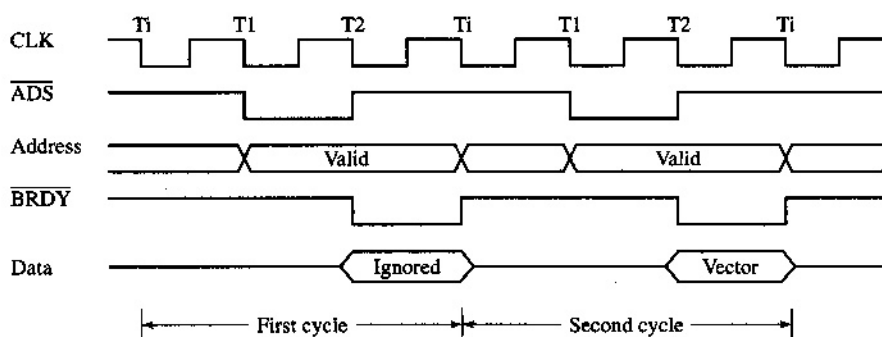


FIGURE 13.7 Interrupt acknowledge cycles

### Bus Hold

The HOLD input provides a second way for a different bus master to take control of the Pentium's buses. Unlike  $\overline{BOFF}$ , HOLD completes the current bus cycle and then tri-states its buses. The HLDA output indicates when the Pentium is in the HOLD state.

### Interrupt Acknowledge

The processor runs two interrupt acknowledge cycles in response to an INTR request. Both cycles are locked. External circuitry (such as the 8259A Programmable Interrupt Controller used in the PC) is responsible for supplying an 8-bit interrupt vector number on D<sub>0</sub> through D<sub>7</sub>. To maintain hardware compatibility with earlier 80x86 machines, the data on D<sub>0</sub> through D<sub>7</sub> is ignored by the processor during the first interrupt acknowledge cycle and accepted during the second. Figure 13.7 shows the associated timing.

The byte enable outputs are used to indicate which of the two cycles the processor is running. During the first interrupt acknowledge cycle,  $\overline{BE}_4$  is low and the other 7 byte enables are high. During the second cycle, only  $\overline{BE}_0$  is low.

### Cache Flush

In response to a zero supplied to the  $\overline{FLUSH}$  input, the Pentium flushes its internal code and data cache by performing writebacks for any lines that have been changed. The lines are then invalidated. When all writebacks complete, the processor runs a **flush acknowledge** cycle to inform external circuitry (such as a second-level cache) that its caches are flushed.

### Shutdown

If the Pentium detects an internal parity error, a shutdown cycle is run. Execution is suspended while in shutdown, until the processor receives an NMI, an INIT, or a RESET request. Internal cache data remains unchanged unless the processor is snooped with an inquire cycle or the cache is flushed via FLUSH.

### HALT

This cycle is similar to shutdown, except that the INTR signal may also be used to resume execution (if interrupts are enabled). This cycle is run when the processor encounters the HLT instruction.

### Pipelined Cycles

The Pentium is capable of starting a second bus cycle before the current one is completed. It does so through pipelined read and write logic, in response to a request on the  $\overline{NA}$  input. As always, pipelining improves performance by performing different operations in parallel. Read cycles may be pipelined into write cycles, and vice versa. Writeback operations and locked bus cycles are not pipelined.

### Inquire Cycles

Inquire cycles are used to maintain cache coherency in a multiprocessor system. The Pentium processor is able to watch the system bus (address, data, and control signals) in a multiprocessor system. This is called **bus snooping**. If the Pentium detects a memory read/write operation being performed by another CPU, it runs an internal inquire cycle to determine whether the address on the bus is stored in one of its internal caches. If so, the cache may need to be updated. In order to implement bus snooping, it is necessary for the address bus to be bidirectional.

### Bus Cycle Summary

It is necessary to design memory systems capable of supporting all of the different kinds of bus cycles that are possible with the Pentium. Good use must be made of the bus cycle definition signals and the byte enable outputs. Not taking advantage of this important part of the Pentium's architecture will result in lower performance.

---

## 13.5 THE PENTIUM'S SUPERSCALAR ARCHITECTURE

The Pentium is capable, under special circumstances, of executing two integer or two floating-point instructions simultaneously. This parallel execution is made possible through the Pentium's twin U and V pipelines. Processors capable of parallel instruction execution of multiple instructions are known as **superscalar machines**.

There are four restrictions placed on a pair of integer instructions attempting parallel execution:

- Both must be simple instructions.
- No data dependencies may exist between them.
- Neither instruction may contain both immediate data and a displacement value (such as `MOV TABLE[SI],7`).
- Prefixed instructions (such as `MOV ES:[DI],AL`) may only execute in the U pipeline.

A simple instruction does not require microcode control to execute and generally takes one clock cycle to complete. Some examples are register-to-register MOVs, INC, DEC, and near conditional jumps (JZ, JNZ, etc.). A conditional jump must be the second instruction in the pair.

Some simple instructions may take two or three clock cycles. These are arithmetic and logical instructions (ALU instructions) that use both register and memory operands.

A data dependency between two instructions exists if the second instruction reads an operand written to it by the first instruction (read-after-write dependency), or if both instructions write to the same operand. For example, the instructions

```
ADD AX, BX
ADD AX, CX
```

cannot be paired, because both read and write the AX register.

For floating-point instructions, the first instruction of the pair must be one of the following:

```
FLD <single/double>      FLD ST(i)
FADD      FSUB      FMUL      FDIV
FCOM      FUCOM      FTST      FABS
FCHS
```

The second instruction must be FXCH.

As mentioned earlier, the compiler plays an important role in the ordering of instructions during code generation. By looking for the allowable combinations of integer and floating-point instructions, the compiler is able to help keep the Pentium's superscalar architecture running at full speed.

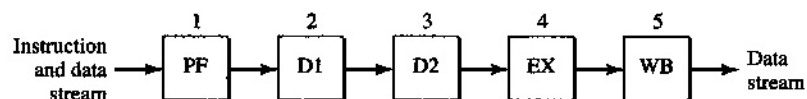
## 13.6 PIPELINING

As mentioned in Section 13.3, **pipelining** is a valid technique for improving instruction execution rate. Let us examine the operation of the Pentium's U and V pipelines.

Figure 13.8 shows the names and order of the five-stage U and V instruction pipelines. Specifically, we have:

|    |                                |
|----|--------------------------------|
| PF | Prefetch                       |
| D1 | Instruction Decode             |
| D2 | Address Generate               |
| EX | Execute, Cache, and ALU Access |
| WB | Writeback                      |

Note that each pipeline has its own PF stage, its own D1 stage, and so on. The U pipeline can execute any processor instruction (including the initial stages of the floating-point instructions), whereas the V pipeline only executes *simple* instructions (as defined in Section 13.5). Under ideal conditions, two integer instructions may complete execution every clock cycle, as indicated by Figure 13.9. Recall that a pair of instructions that execute in parallel must both be simple instructions and cannot have any data dependencies between them. Examine the state of the pipelines during the fifth clock cycle. The first pair of instructions complete execution in this clock cycle. Four more cycles are needed to complete the other eight instructions that are in various stages of execution.



**FIGURE 13.8** U and V instruction pipeline stages

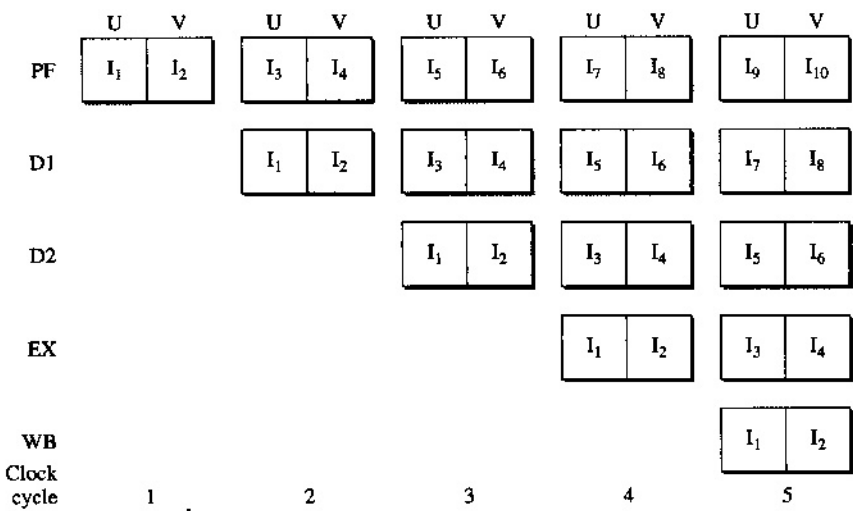


FIGURE 13.9 Pipelined Instruction execution

Instructions are fed into the PF stage after being prefetched from the instruction cache or memory. Decoders in each D1 stage (U and V pipelines) determine if the current pair of instructions can execute together. Instructions that contain a prefix byte require an additional clock cycle in the D1 stage and may only execute in the U pipeline. They may not be paired with any other instruction.

Addresses for operands that reside in memory are calculated in stage D2. In the EX stage, operands are read from the data cache (or memory), ALU operations are performed, and branch predictions for instructions (except conditional branches in the V pipeline) are verified. See Section 13.7 for details on branch prediction.

The final stage, WB, is used to write the results of the completed instruction and verify conditional branch instruction predictions.

When paired instructions reach the EX stage, it is possible that one or the other will stall and require additional cycles to execute. A pipeline stall lowers performance, because no work is done during the stall. Instructions stall for various reasons, most notably when their operands are not found in the data cache. If the instruction in the U pipeline stalls, so does the instruction in the V pipeline. If the V pipeline instruction stalls, the instruction in the U pipeline may continue executing. Both instructions must progress to the WB stage before another pair (or the next single instruction) may enter the EX stage.

13.7 BRANCH PREDICTION

The performance gain realized through pipelining can be reduced by the presence of program transfer instructions in the instruction stream. Instructions such as JMP, CALL, RET, and the conditional jumps change the flow of execution in a program. This is a problem for the instruction pipeline, which is always filled with a group of instructions that occupy sequential locations in memory. Program transfer instructions change the sequence, causing all instructions that entered the pipeline after the program transfer instruction to

become invalid. Refer back to Figure 13.3(b). Suppose that instruction  $I_3$  is a conditional jump to instruction  $I_{50}$  at some other address in the program (the **target address**). This means that the instructions that entered the pipeline after  $I_3$  ( $I_4$  and  $I_5$ ) are invalid, because they occur in the program *after* the conditional jump. These instructions must be discarded, or flushed, from the pipeline and the new sequence of instructions, beginning with  $I_{50}$ , loaded in. This causes bubbles in the pipeline, where no work is done as the pipeline stages are reloaded.

To avoid this problem, the Pentium uses a scheme called dynamic branch prediction. In this scheme, a prediction is made concerning the branch instruction currently in the pipeline. The prediction will be either taken or not taken. If the prediction turns out to be true, the pipeline will not be flushed, and no clock cycles will be lost. If the prediction turns out to be false, the pipeline is flushed and started over with the correct instruction. It is best if the predictions are true most of the time.

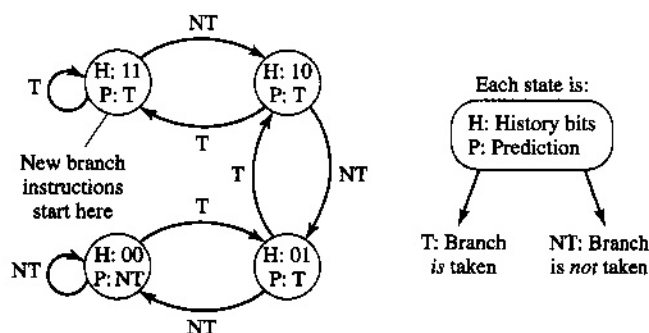
The Pentium accomplishes branch prediction through the use of a branch target buffer (BTB), a special cache that stores the instruction and target addresses of any branch instructions that have been encountered in the instruction stream. Along with the addresses for each instruction, the BTB also stores two history bits that indicate the execution history of the last two branch instructions. The **history bits** are initially set to 11 when a new target address is placed into the BTB. Whenever the corresponding branch instruction is encountered, the history bits are updated, as indicated in Figure 13.10. Note that repeated failures to take a branch cause the history bits to become 00 and the prediction to become not taken.

The BTB uses the history bits to predict whether the branch is taken or not taken. As long as they are not both zero, the prediction will be taken. If a new branch instruction is encountered (no target address in the BTB), the prediction is not taken.

Two 32-byte prefetch buffers work with the BTB and the D1 stage of the U and V pipelines to keep a steady stream of instructions flowing into the pipelines. One buffer prefetches instructions from the current program address. The other buffer, activated when the BTB predicts "taken," will prefetch instructions from the target address. The BTB is accessed with the address of the branch instruction during the D1 stage. If found and the BTB's prediction is "taken," the second prefetch buffer becomes active and no cycles are lost. Incorrect predictions, or correct predictions with the wrong target address, cause the pipelines to be flushed. This wastes three clock cycles in the U pipeline and four in the V pipeline.

In accordance with good programming practice, conditional jumps are used to perform loops. Because we generally require multiple passes through a loop, we would want to begin predicting "taken" right away. The Pentium sets the history bits to 11 for a new

**FIGURE 13.10** Dynamic branch prediction operation



entry, so the number of lost clock cycles will be minimized by the dynamic branch prediction mechanism.

## 13.8 THE INSTRUCTION AND DATA CACHES

The Pentium processor employs two separate internal cache memories, one for instructions and the other for data. In this section, we will examine the benefits of using caches, their internal organization and operation, and methods of maintaining valid data in both cache and main memory.

### Cache Background

Which is faster: 10-ns memory or 70-ns memory? The answer is obvious and at the heart of the performance goal for a system using cache memory. Cache is a special type of high-speed RAM (access time of 10 ns or less) that is used to help speed up accesses to memory and reduce traffic on the processor's buses.

As indicated by Figure 13.11, an on-chip cache is used to feed instructions and data to the CPU's pipeline. When an instruction or data operand is required from main memory, the on-chip cache will be searched first. If the instruction or data is found in the cache (a *hit*), a copy is sent to the pipeline very quickly, usually within one clock cycle.

When the required instruction or data is not found in the internal cache (a *miss*), the processor is forced to go to external memory. An external cache (also called a **second-level cache**) is examined next. If there is another miss, or there is no external cache, the main memory is accessed. A copy of the instruction or data from main memory is written to the cache so that it will be there if needed again.

How does all this activity speed things up? Consider a system that has an internal cache access time of 10 ns and a main memory access time of 70 ns. There is no external cache. The time for a hit is 10 ns. The time for a miss is 80 ns, because both cache and main memory are accessed during a miss. So, the average access time will be between 10 ns and 80 ns, depending on how many hits there are. A **hit ratio** specifies the percentage of hits to total cache accesses. For example, a hit ratio of 0.9 means that nine times out of ten the cache contains the requested information. Thus, the hit ratio affects the average access time. To predict the average access time, use the following equation:

$$T_{acc} = \text{HitRatio} \times T_{cache} + (1 - \text{HitRatio}) \times (T_{cache} + T_{ram})$$

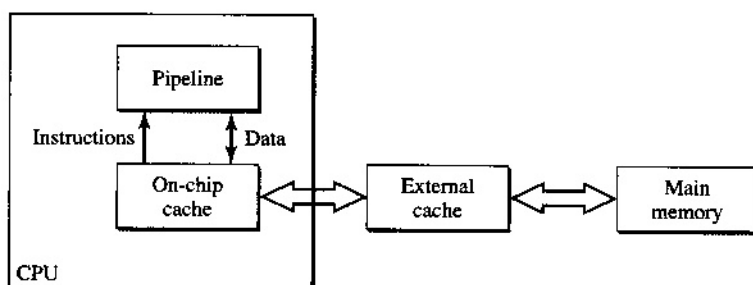


FIGURE 13.11 Using cache in a microcomputer system

Continuing with our example, a hit ratio of 0.9 results in an access time of 17 ns (9 ns for hits, 8 ns for misses). The hit ratio is governed by many factors, including the size of the program, the type and amount of data used by the program, and the addressing activity during execution.

Two characteristics of a running program pave the way for a performance improvement when cache is used. First, when we access a memory location, there is a good chance we will access it again in the future. Second, when we access one location, there is a good possibility that we will access the *next* location also. In general, these accesses maintain a *locality of reference* in a small range of memory locations. Consider the following loop of instructions:

```

                                MOV    CX,1000
                                SUB    AX,AX
NEXT:                          ADD    AX,[SI]
                                MOV    [SI],AX
                                INC    SI
                                LOOP   NEXT

```

The code for the four instructions that make up the body of the loop will be executed 1,000 times. If the cache is initially empty, each instruction fetch will generate a miss in the cache, causing the instruction to be read from main memory. The first pass through the loop fills the cache with the code for the entire loop. The next 999 passes will all generate hits for each instruction fetch. This will speed things up tremendously.

When a miss occurs, the cache reads a copy of a *group* of locations from main memory. This group is called a **line of data**. This prepares the cache for hits in case there are more localized references made (to data within the line). Our loop example shows the benefit of this process. The first instruction, `MOV CX,1000`, causes a miss. When its code is copied from main memory, so are the bytes for the next few instructions (as many as will fit within the line). So, after fetching the first instruction, the rest of the loop is already in the cache, before we have even finished the first pass!

In addition to the instruction fetches, the loop example also contains accesses to data operands stored in memory via `ADD AX,[SI]` and `MOV [SI],AX`. Once again, during reads, a miss will cause a line of data to be read from main memory. So each miss reads in data for the next few passes through the loop, guaranteeing fast data access.

But what about the data writes caused by the `MOV [SI],AX` instruction? Should they be written just to the cache (10 ns) or to the cache and main memory (70 ns)? This depends on the cache policy used by a particular system. Writing results only to the cache is called **writeback**; writing to the cache and to main memory is called **writethrough**. The writeback policy results in fast writes at the expense of out-of-date main memory data. Writethrough maintains valid data in main memory, but requires long write times, which slows down execution.

Out-of-date main memory data in the writeback policy is eventually updated with the correct data from the cache. This happens when the cache is full and a line must be replaced with a new line of data coming in. If the line being replaced was written to, it must be copied back to main memory. So during a miss, it may be necessary to access main memory twice: once to store the updated line being replaced, and a second time to read the new line of data to satisfy the miss.

When the cache is full and a line must be replaced, the victim line may be chosen any number of ways. One algorithm used to pick a victim is called **LRU (least recently used)**. LRU is based on the idea that the cache entry least recently used is not likely to be used in

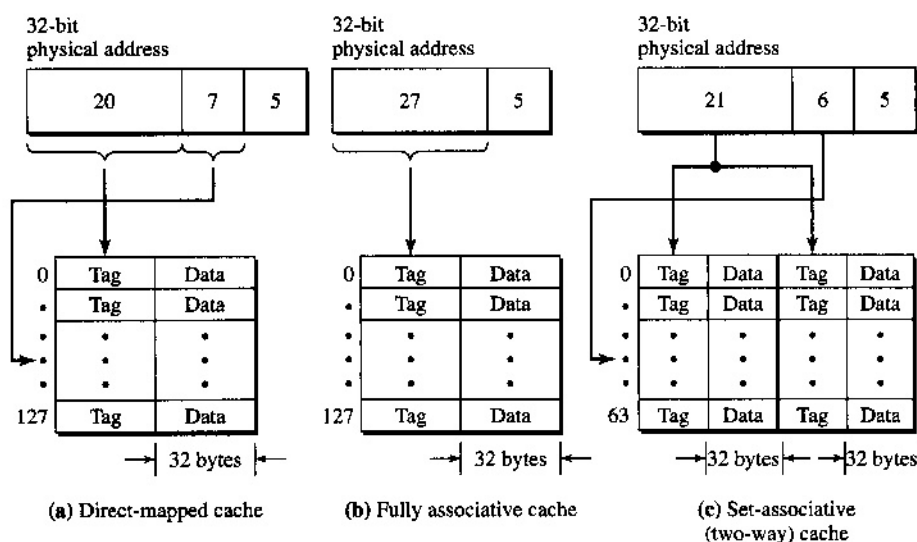
the future, so it may be replaced. One or more bits are added to the cache entry to support the LRU algorithm. These bits are updated during hits and examined when a victim must be chosen.

## Cache Organization

How is it possible that a cache that contains numerous entries can search them so quickly and report a hit if a match is found? This has to do with the organization of the cache. The address and data that make up each cache entry may be organized in different hardware configurations. There are three main designs: direct-mapped, fully associative, and set-associative. A direct-mapped cache uses a portion of the incoming physical address to select an entry. A *tag* stored with the entry is compared with the remaining address bits. A match represents a hit. This cache is illustrated in Figure 13.12(a). Note that there are 128 entries in the cache, which has a line size of 32 bytes. The lower 5 bits of the physical address are not needed, because all the bytes accessed by them are stored in a single line. The next 7 address bits (called *index bits*) are used to select one of 128 entries in the cache. The tag at the selected entry is compared with the remaining 20 upper address bits.

Fully associative cache, shown in Figure 13.12(b), uses larger tags and does not select an entry based on index bits, as direct-mapped cache does. Instead, the upper address bits of the incoming physical address are compared with *every* tag in the cache. This requires more extensive hardware than the direct-mapped cache. However, the fully associative cache is capable of storing data in more flexible ways than a direct-mapped cache. For instance, in the direct-mapped cache there can be only one data entry with an index value of 7 at any time. In the fully associative cache, multiple tags may contain addresses that correspond to an index of 7 (although index bits are not used). This helps to increase the hit ratio over that of the direct-mapped cache when the same data is accessed frequently.

The compromise between the direct-mapped and fully associative designs is the **set-associative cache**. The entries are divided into sets containing two, four, eight, or more



**FIGURE 13.12** Cache organization: (a) direct-mapped; (b) fully associative; (c) set-associative (two-way)



entries. Two entries per set is referred to as *two-way set-associative cache*. Each entry in a set has its own tag. A particular set is selected through the use of index bits. The remaining upper address bits are compared with each tag in the set at the same time. The tag comparators are smaller than those used in the fully associative cache, because they do not have to compare as many bits. Figure 13.12(c) shows how a two-way set-associative cache is organized.

The set-associative cache combines the simple tag comparator hardware of the direct-mapped cache with the flexible data caching capability of the fully associative cache. This is the type of cache used by the Pentium.

## The Data and Instruction Cache

The Pentium's data and instruction caches are both organized as two-way set-associative caches with 128 sets. This gives 256 entries per cache. There are 32 bytes in a line (64 bytes per set), resulting in 8KB of storage per cache. The data and instruction caches may be accessed simultaneously. An LRU algorithm is used to select victims in each cache.

Each entry in a set has its own tag. Figure 13.13 shows the internal structure of each cache. The tags in the data cache are **triple ported**, meaning that they can be accessed from three different places at the same time. Two of these are the U and V pipelines, which access the data cache to read/write instruction operands. The third port is used for a special operation called **bus snooping**. Bus snooping is used to help maintain consistent data in a multiprocessor system where each processor has a separate cache.

In addition, each entry in the data cache can be configured for writethrough or write-back.

The instruction cache is write-protected to prevent self-modifying code from changing the executing program. Tags in the instruction cache are also triple ported, with two ports used for split-line accesses (reading the upper half of one line and the lower half of the next line at the same time). The third port is for bus snooping.

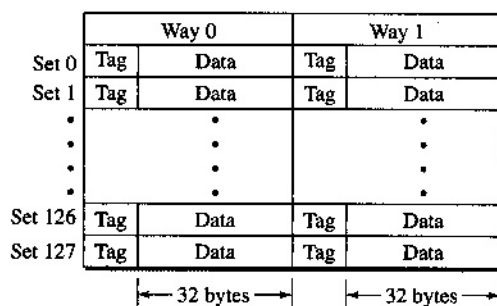
Parity bits are used in each cache to help maintain data integrity. Each tag has its own parity bit, as does every byte in the data cache. There is one parity bit for every 8 bytes of data (a quarter of a line) in the instruction cache.

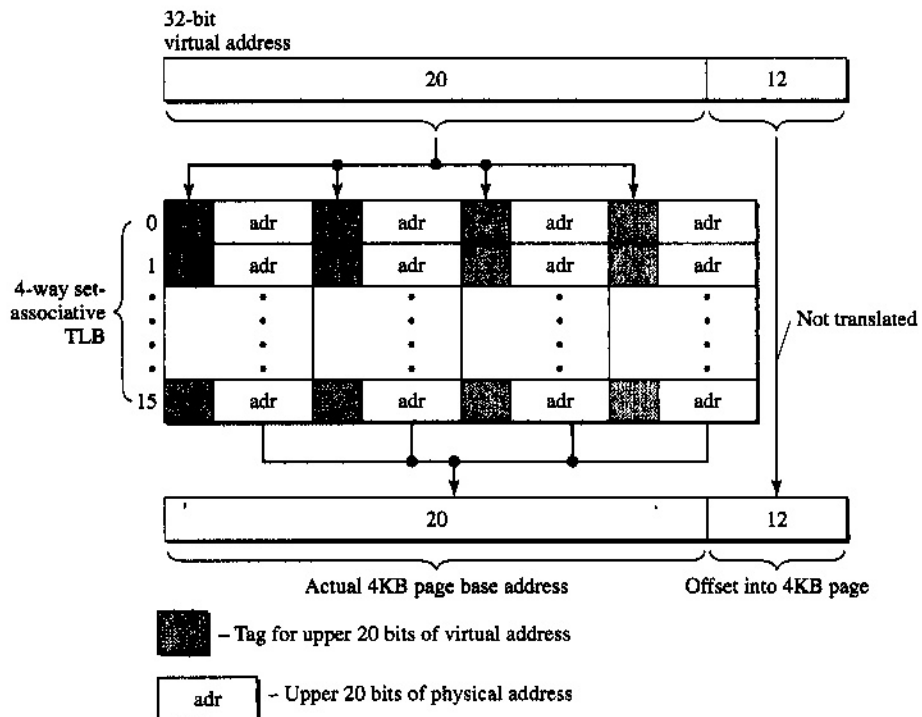
## The Translation Lookaside Buffers

The instruction and data caches contain **TLBs (translation lookaside buffers)** that translate *virtual* (also called *logical* or *linear*) *addresses* into physical addresses. Physical addresses are used to access the cache because the same address is used to access main memory.

The TLBs are caches themselves. The data cache contains two TLBs. The first is four-way set-associative with 64 entries. This TLB translates addresses for 4KB pages of

**FIGURE 13.13** Structure of 8KB instruction and data cache





**FIGURE 13.14** Translating virtual addresses into physical addresses with a TLB

main memory. Figure 13.14 shows how this TLB is used to translate a virtual address into a physical address.

The lower 12 bits of the physical address are the same as the lower 12 bits of the virtual address. The upper 20 bits of the virtual address are checked against four tags and translated into the upper 20 physical address bits during a hit. Because this translation must be done very quickly, the TLB is kept small (only 64 entries).

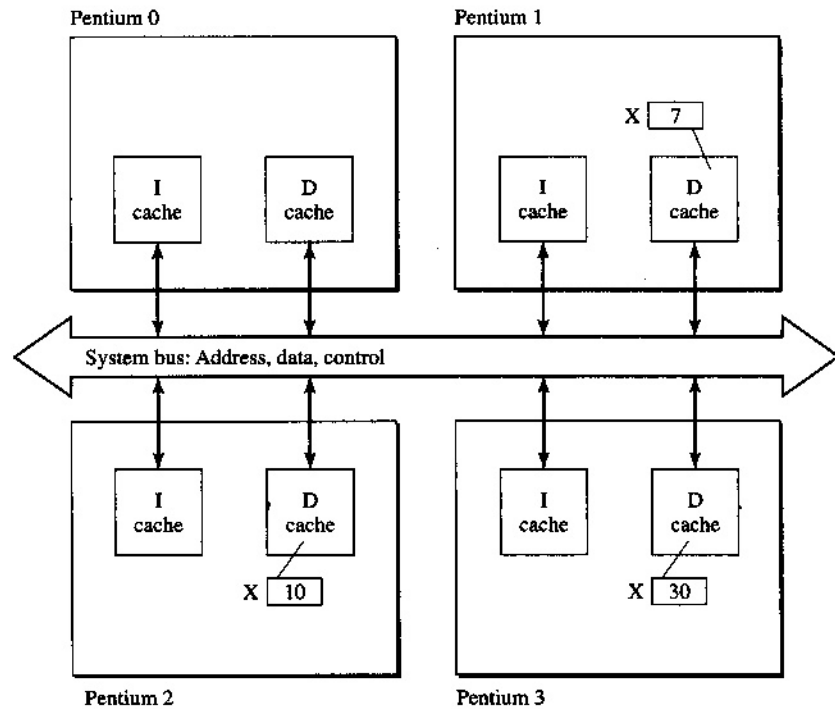
The second TLB used by the data cache is four-way set-associative with eight entries and is used to handle 4MB pages. Both TLBs are parity protected and dual ported.

The instruction cache uses a single four-way set-associative TLB with 32 entries. Both 4KB and 4MB page addresses are supported, with the 4MB pages cached in 4KB chunks. Once again, parity bits are used on the tags and data to maintain data integrity.

Entries are replaced in all three TLBs through use of a 3-bit LRU counter stored with each set.

### Cache Coherency in a Multiprocessor System

When multiple processors are used in a single system, there needs to be a mechanism whereby all processors agree on the contents of shared cache information. For example, two or more processors may utilize data from the same main memory location, X. Each processor will maintain a copy of the data for location X in its own data cache. So after each processor has changed the value of the data item, it is possible to have different (incoherent) values of X's data in each cache, as shown in Figure 13.15. When this happens, which value of X's data should actually be written back into main memory?



**FIGURE 13.15** A multiprocessor system with incoherent cache data

The Pentium's mechanism for maintaining **cache coherency** in its data cache is called **MESI (modified/exclusive/shared/invalid)**. MESI is a cache-consistency protocol that uses 2 bits stored with each line of data to keep track of the state of the cache line. The four states are modified, exclusive, shared, and invalid. Specifically, each state is defined as follows:

**Modified:** The current line has been modified (does not match main memory) and is only available in a single cache.

**Exclusive:** The current line has not been modified (matches main memory) and is only available in a single cache. Writing to this line changes its state to modified.

**Shared:** Copies of the current line may exist in more than one cache. A write to this line causes a writethrough to main memory and may invalidate the copies in the other cache.

**Invalid:** The current line is empty. A read from this line will generate a miss. A write will cause a writethrough to main memory.

Only the shared and invalid states are used in the code cache.

The MESI protocol requires the Pentium to monitor all accesses to main memory in a multiprocessor system (**bus snooping**). Referring back to Figure 13.15, if Processor 3 writes its local copy of X (30) back to memory, the memory write cycle will be detected by the other three processors. Each processor will then run an internal *inquire* cycle to determine whether its data cache contains the address of X. Processors 1 and 2 will update their cache based on their individual MESI states.

Inquire cycles examine the code cache as well. Recall that the code and data cache tags are triple ported, with one port dedicated to bus snooping. The Pentium's address lines are used as inputs during an inquire cycle to accomplish bus snooping.

## Cache Instructions

Three instructions are provided to allow the programmer some control over the operation of the cache. These instructions are **INVD** (Invalidate cache), **INVLPG** (Invalidate TLB entry), and **WBINVD** (Writeback and invalidate cache). **INVD** effectively erases all information in the data cache (by marking it all invalid). Any values not previously written back will be lost when **INVD** executes. This problem can be avoided by using **WBINVD**, which first writes back any updated cache entries, and then invalidates them. **INVLPG** invalidates the TLB entry associated with a supplied memory operand. This may be necessary when implementing a paged memory system.

Clearly these instructions are meant to be used by system programmers. Indiscriminately wiping out the cache during execution generally leads to trouble.

## Cache Summary

In this section, we examined how data and code cache may be used to enhance processing speed. The operation of the Pentium's 8KB two-way set-associative code and data caches was discussed. We also examined the operation of a translation lookaside buffer, the different cache policies (such as writethrough and writeback), and the MESI protocol for maintaining cache consistency.

In conclusion, it is important to note that all of these cache operations are performed automatically by the Pentium. No programming code is needed to get the cache to work.

## 13.9 THE FLOATING-POINT UNIT

Like the 80486 processor, the Pentium contains an on-chip floating-point unit. Prior to the 80486, an external math coprocessor was used to perform floating-point operations for the 8086, 80286, and 80386. These 80x87 coprocessors, illustrated in Table 13.5, shared address and data signals, as well as control lines, with the processor. The simple fact that these coprocessors were *outside* the CPU increased the time required to perform a floating-point operation, due to synchronization issues. Moving the coprocessor onto the same chip as the processor, like the 80486 and Pentium, allows faster communication and quicker execution.

The Pentium's FPU is totally redesigned over that used in the 80486. Many floating-point instructions now require fewer clock cycles than previous 80x87 units, and new algorithms provide additional speed increases. Table 13.6 shows the improvement in the floating-point multiply instruction, **FMUL**, in each generation. Clearly, the Pentium's FPU leaves little room for improvement.

**TABLE 13.5** Coprocessor family

| Processor | Coprocessor  |
|-----------|--------------|
| 8086/88   | 8087         |
| 80286     | 80287        |
| 80386     | 80387        |
| 80486     | Internal FPU |
| Pentium   | Internal FPU |

**TABLE 13.6** FMUL instruction performance

| Coprocessor | Minimum Clock Cycles Required |
|-------------|-------------------------------|
| 8087        | 130                           |
| 80287       | 130                           |
| 80387       | 29                            |
| 80486 FPU   | 16                            |
| Pentium FPU | 1                             |

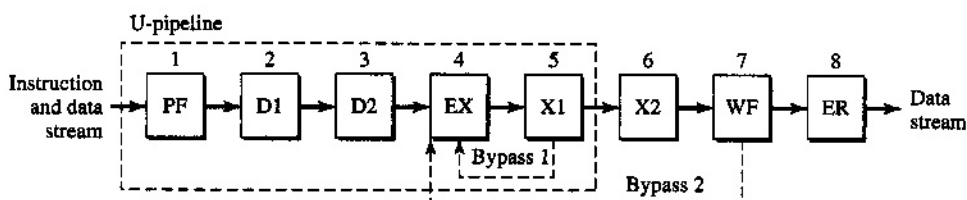


FIGURE 13.16 FPU pipeline stages

The FPU achieves its quick speed through a pipeline containing eight stages. The first five stages make up the U pipeline, which processes integer instructions. The operation of the fifth stage is different, however, and feeds the remaining three stages that complete the floating-point pipeline. Recall that the fifth stage of the U pipeline was WB (writeback). When configured for a floating-point operation, the fifth stage becomes X1, the first floating-point execution stage. Figure 13.16 shows the eight-stage FPU pipeline. The stages, and their functions, are as follows:

|    |                                                                                                                                                                           |
|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PF | Prefetch                                                                                                                                                                  |
| D1 | Instruction decode                                                                                                                                                        |
| D2 | Address generation                                                                                                                                                        |
| EX | Memory and register read, floating-point data converted into memory format, memory write                                                                                  |
| X1 | Floating-point execute, stage one. Memory data converted into floating-point format, write operand to floating-point register file, bypass 1 (send data back to EX stage) |
| X2 | Floating-point execute stage two                                                                                                                                          |
| WF | Round floating-point result and write to floating-point register file, bypass 2 (send data back to EX stage)                                                              |
| ER | Error reporting, update status word                                                                                                                                       |

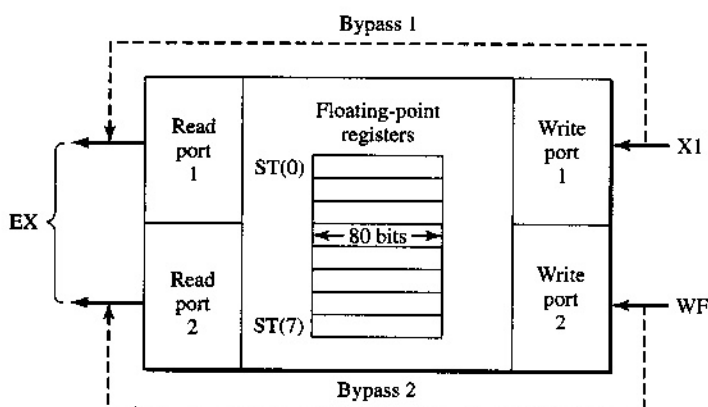
Note the use of two different bypass connections. Bypass 1 connects the output of the X1 stage to the input of the EX stage. This is done to allow a floating-point register write operation in the X1 stage to bypass the floating-point register file and send the results to the instruction performing a floating-point register read in the EX stage. For example, the two instructions

```
FLD    ST
FMUL   ST
```

both use the ST register as an operand. The result produced by FLD must be available before the FMUL instruction can proceed. Using bypass 1, the data from FLD (in stage X1) is made available as an input operand to FMUL (in stage EX) before it is written into the floating-point register file. This prevents the pipeline from stalling while waiting for the register file to provide a copy of ST. Overall, we get a decrease in the number of clock cycles required to perform many different combinations of instructions. This technique is also called *forwarding*.

The second type of bypass takes place between the WF and EX stages, where the result of an arithmetic instruction (in stage WF) is made available to the next instruction fetching operands in the EX stage.

The floating-point register file contains the eight 80-bit floating-point registers, ST(0) through ST(7). The read and write sections are dual ported, to allow two reads or two



**FIGURE 13.17** FPU register file

writes to take place simultaneously. As indicated by Figure 13.17, the two read ports send data into the EX stage (which reads register operands). The two write ports receive data from the X1 and WF stages of the pipeline. Both bypasses are shown to illustrate how results are fed back from X1 and WF to EX.

The Pentium is capable of executing two floating-point instructions at the same time under special circumstances. To do so, the first instruction of the pair must be one of the following:

|                     |           |      |      |
|---------------------|-----------|------|------|
| FLD <single/double> | FLD ST(i) |      |      |
| FADD                | FSUB      | FMUL | FDIV |
| FCOM                | FUCOM     | FTST | FABS |
| FCNS                |           |      |      |

The second instruction must be FXCH. Because the U pipeline makes up the first five stages of the FPU pipeline, the FXCH instruction executes in the V pipeline. Because both instructions execute in parallel, the FXCH instruction literally requires *zero* clock cycles! Normally FXCH requires one clock cycle when issued by itself.

Because many of the floating-point instructions use the operand on top of the floating-point stack, there is a good possibility of a stall when two instructions require the top-of-stack operand. Using FXCH as the second instruction of a legal pair helps avoid this problem.

If the second floating-point instruction in a pair is not FXCH, both floating-point instructions are executed one at a time. If the second instruction is not a floating-point instruction, it cannot be paired with the first instruction because the FPU uses both pipelines (U and V) when fetching a 64-bit operand from memory.

Overall, the Pentium's FPU has many features that contribute to fast floating-point execution.

## 13.10 TROUBLESHOOTING TECHNIQUES

The advanced nature of the Pentium microprocessor requires us to think differently about the nature of computing. As we have already seen, exotic techniques such as branch prediction, pipelining, and superscalar processing have paved the way for improved performance. Let us take a quick look at some other improvements from Intel:

- Intel has added MMX Technology to its line of Pentium processors (Pentium, Pentium Pro, and Pentium II). A total of fifty-seven new instructions enhance the

processor's ability to manipulate audio, graphic, and video data. Intel accomplished this major architectural addition by *reusing* the 80-bit floating-point registers in the FPU. Using a method called SIMD (single instruction multiple data), one MMX instruction is capable of operating on 64 bits of data stored in an FPU register.

- The Pentium Pro processor (and the newer Pentium II) employs a technique called *speculative execution*. In this technique, multiple instructions are fetched and executed, possibly out of order, in order to keep the pipeline busy. The results of each instruction are speculative until the processor determines that they are needed (based on the result of branch instructions and other program variables). Overall, a high level of parallelism is maintained.
- First used in the Pentium Pro, a new bus technology called Dual Independent Bus architecture utilizes two data buses to transfer data between the processor and main memory (including the level-2 cache). One bus is for main memory, the second for the level-2 cache. The buses may be used independently or in parallel, significantly improving the bus performance over that of a single-bus machine.
- The five-stage Pentium pipeline was redesigned for the Pentium Pro into a *superpipelined* fourteen-stage pipeline. By adding more stages, less logic can be used in each stage, which allows the pipeline to be clocked at a higher speed. This is easily verified by the 200- and 300-MHz processors that came after the first Pentium. Although there are drawbacks to superpipelining, such as bigger branch penalties during an incorrect prediction, its benefits are well worth the price.

Every aspect of computing must be studied in order to fully understand how to develop improved methods, software, and hardware for increased performance. Spend some time trying to think of an improvement of your own, as if you are designing a new microprocessor. Look through recent issues of computer architecture journals, or search the Web for information. You will find that a lot of people are thinking about improvements, too.

---

## SUMMARY

In this chapter we examined the architectural organization and operation of the Pentium microprocessor. The Pentium's 64-bit data bus, internal code and data caches, twin integer pipelines, on-chip FPU, and branch prediction hardware combine to produce impressive execution speed. In addition, the MESI cache coherency protocol allows the Pentium to operate in parallel with other Pentiums in a multiprocessing system. In short, the Pentium utilizes a new architectural direction while maintaining software compatibility with the rest of the 80x86 family.

---

## STUDY QUESTIONS

1. How is a 32-bit address output by the Pentium?
2. Why are the Pentium's address lines bidirectional?
3. What type of error checking is performed on the address and data lines?
4. How is instruction execution possible when the processor buses are in a HOLD state?
5. What processor signals are used to define the type of bus cycle currently running?
6. What signals are associated with the internal cache?
7. What are the Pentium's hardware interrupt signals?

8. Explain the differences between a CISC processor and a RISC processor.
9. How are accesses to main memory reduced when a cache is used?
10. How does an instruction pipeline speed up execution?
11. What is branch prediction?
12. How does a compiler affect the performance of the machine code?
13. How does the Pentium indicate the type of bus cycle it is running?
14. What bus cycle states are directly reachable from state T2?
15. How many bytes are read/written during a single-transfer bus cycle? How many for a burst transfer?
16. What are burst transfers used for?
17. What are locked bus cycles used for? How does the Pentium indicate that the current bus cycle is locked?
18. Is it possible for two bus cycles to be in progress at the same time?
19. What is the main difference between the shutdown and halt bus cycles with regard to the internal state of the processor?
20. Why is the Pentium a superscalar processor?
21. Is `ADD DATA[BP],3` a simple instruction?
22. What type of dependency exists between these two instructions?

```
ADD  BX,CX
SUB  DX,BX
```

23. Name the five stages used by the U and V pipelines.
24. Are there any differences between the U and V pipelines? If so, what are they?
25. What is a pipeline stall? What role does the data cache play in a stall?
26. How is it possible, through instruction pairing, to execute ten instructions in only nine clock cycles?
27. What are the history bits used for in a BTB entry?
28. The instruction `JMP [BP]` may be correctly predicted each time it is encountered in a program, but still cause stalls in the pipeline. Why is this?
29. What is the BTB's prediction for a conditional jump the first time it is encountered?
30. What pipeline stage plays a role in branch prediction?
31. What is a cache hit?
32. What happens during a cache miss?
33. What is the average access time for a system that contains 10-ns cache and 80-ns RAM if the hit ratio is 0.95?
34. What does locality of reference mean? How does it apply to an executing program?
35. Describe the structure of the Pentium's 8KB two-way set-associative cache.
36. Explain the code and data cache activity during execution of this loop of code:

```
      MOV     CX,20
TOP:  CMP     AL,[SI]
      ADC     BL,[SI]
      INC     SI
      LOOP    TOP
```

37. What is the difference between writethrough and writeback?
38. What is second-level cache?
39. What does MESI stand for?
40. What is bus snooping?
41. What is an inquire cycle?
42. List the reasons why the Pentium's FPU is faster than the 80486's FPU.
43. Which pipeline (U or V) is part of the FPU pipeline?



44. How many stages does the FPU pipeline have? What are they?
45. What is the purpose of each bypass in the FPU pipeline?
46. Why is the floating-point register file dual ported?
47. Which pairs of instructions may execute in parallel?  
(a) FMUL      (b) FXCH      (c) FADD      (d) FSUB  
    FXCH      FSUB      DEC BX      FDIV
48. When FXCH is executed in parallel with another floating-point instruction, how many clock cycles are required for its execution? Explain.

---

# CHAPTER 14

---

## Protected-Mode Operation

---

### OBJECTIVES

In this chapter you will learn about:

- The way memory is utilized in protected mode
- Segmented addressing
- Virtual addressing
- Paged memory management
- Multitasking
- How memory and I/O are protected
- Protected-mode interrupt operation
- Virtual-8086 mode

### KEY TERMS

|                                     |                              |                              |
|-------------------------------------|------------------------------|------------------------------|
| Call gate                           | Machine status word          | Task gate                    |
| Context                             | Mode switch entry point      | Task register                |
| Demand paging                       | Monitor task                 | Task state segment           |
| Descriptor                          | Page directory               | Task switch                  |
| Interrupt descriptor table          | Page directory base register | Time slice                   |
| Interrupt table descriptor register | Page fault                   | Translation lookaside buffer |
| I/O permission bit map              | Page frame                   | Virtual machine              |
| Linear address                      | Page table                   | Virtual-8086 mode            |
|                                     | Segment selector             |                              |

---

## 14.1 INTRODUCTION

The architecture of the Pentium's protected mode is significantly different from that of real mode. In real mode, addresses are generated by shifting 16-bit segment registers to the left and adding a 16-bit offset to create a 20-bit physical address. As we will see, in protected-mode memory addresses are generated in a totally different way. Segment registers are now



CR0 contains many important control and status bits. Their functions are as follows:

PG: Paging. Enables paging when set.

CD: Cache Disable. Disables cache writes when set.

NW: Not Writethrough. Disables cache writethrough operations when set.

AM: Alignment Mask. Allows alignment checking when set.

WP: Write Protect. Enforces supervisor-level write protection when set.

NE: Numeric Error. Allows floating-point errors to be reported when set.

ET: Extension Type. Reserved.

TS: Task Switch. Set when a task switch occurs.

EM: Emulation. Indicates the presence of a coprocessor. Should be zero on the Pentium, which has an internal FPU.

MP: Monitor Coprocessor. Must be set to run 80286 and 80386 programs on the Pentium.

CR2 contains the 32-bit linear address that generated the most recent page fault.

CR3 contains the base address of the current Page Directory, which is used to support paging.

CR4 has 6 bits whose operation is as follows:

VME: Virtual-8086 Mode Extensions. When set, enables emulation of a virtual interrupt flag.

PVI: Protected Mode Virtual Interrupts. When set, allows a virtual interrupt flag to be maintained in protected mode.

TSD: Time Stamp Disable. Used to make the RDTSC instruction privileged.

DE: Debugging Extensions. Enables I/O breakpoints when set.

PSE: Allows 4MB pages when set.

MCE: Enables the machine check exception.

Figure 14.1(b) shows the eight debug registers. These registers are used to support program debugging by indicating the address at which a program breakpoint was generated, as well as the size of the breakpoint data or instruction, whether it was a read or write request, and what kind of bus cycle (instruction fetch, data, or I/O access) to generate breakpoints on. Both control and debug registers may be loaded or saved using the MOV instruction.

Additional protected-mode registers are used to support interrupts and tasks, and these will be covered as well, including the GDTR (global descriptor table register), LDTR (local descriptor table register), IDTR (interrupt descriptor table register), and TR (task register). Privileged instructions that operate on these new registers are present in protected mode, and these will be examined also. Briefly, these new instructions are as follows:

|       |                                          |
|-------|------------------------------------------|
| ARPL  | Adjust requested privilege level         |
| CLTS  | Clear task switched flag                 |
| CPUID | CPU identification                       |
| LAR   | Load access rights                       |
| LGDT  | Load global descriptor table register    |
| LIDT  | Load interrupt descriptor table register |
| LLDT  | Load local descriptor table register     |

|       |                                           |
|-------|-------------------------------------------|
| LMSW  | Load machine status word                  |
| LSL   | Load segment limit                        |
| LTR   | Load task register                        |
| MOV   | Move data to/from control register        |
| RDTSC | Read from time stamp counter              |
| SGDT  | Store global descriptor table register    |
| SIDT  | Store interrupt descriptor table register |
| SLDT  | Store local descriptor table register     |
| SMSW  | Store machine status word                 |
| STR   | Store task register                       |
| VERR  | Verify segment for reading                |
| VERW  | Verify segment for writing                |

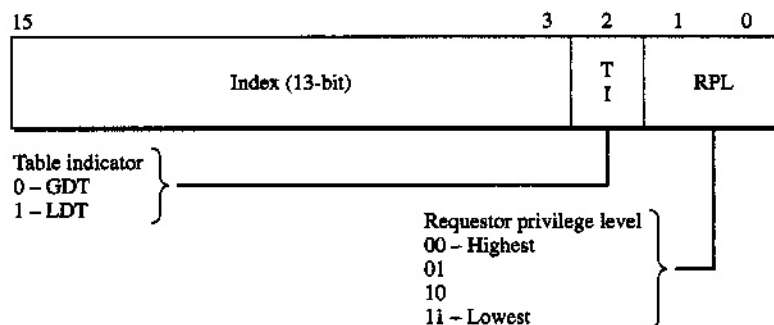
Section 14.2 describes how segmented memory is accessed in protected mode. This is followed by a discussion of the Pentium's virtual address paging mechanism in Section 14.3. Protection methods are covered in Section 14.4. Section 14.5 introduces the powerful multitasking capability of protected mode (through the use of task state segments). Exceptions and interrupts are explained in Section 14.6, followed by protected-mode input and output in Section 14.7. A third mode of execution, virtual-8086 mode, is described in Section 14.8. An example protected-mode application appears in Section 14.9. The chapter concludes with some protected-mode troubleshooting techniques in Section 14.10.

## 14.2 SEGMENTATION

Segmented memory is utilized by protected mode to allow tasks to have their own separate memory spaces, which are protected from access by other tasks. In this section, we will examine the operation of segmented memory.

### Selectors

As previously mentioned, the segment registers we are familiar with from real mode have a different function in protected mode. Figure 14.2 shows the format of a segment selector. **Segment selectors** contain a 13-bit index field that is used to select one of 8,192 segment



**FIGURE 14.2** Segment selector

**descriptors** that reside either in the global descriptor table (GDT) or the local descriptor table (LDT). There is only one GDT in protected mode. Protected-mode tasks, however, may each have their own LDT. The TI bit in the segment selector picks the appropriate descriptor table during translation.

The GDT is located in memory through use of the GDTR. The GDTR is initialized with the LGDT (load global descriptor table register) instruction. LGDT loads 6 bytes of data from a source memory operand into the GDTR.

Local descriptor tables are referenced through the LDTR, which is initialized through use of the LLDT (load local descriptor table register) instruction. LLDT requires a word-size register or memory operand, which represents the index of the LDT in the GDT.

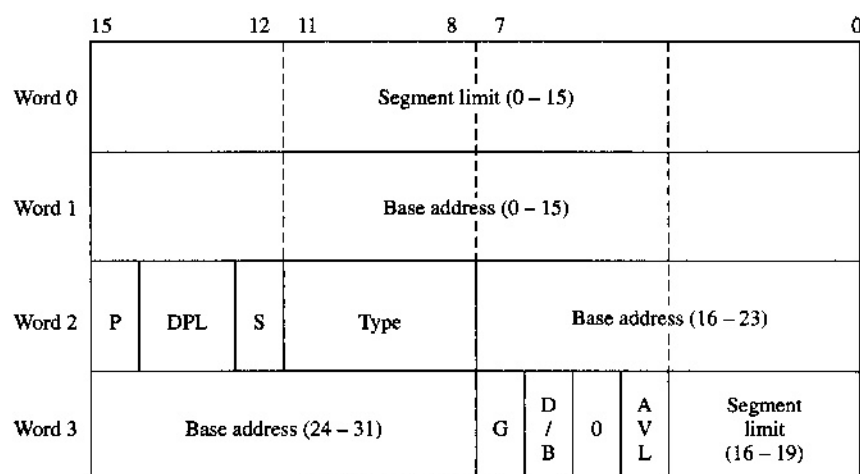
To obtain a copy of the current GDTR or LDTR, use the SGDT (store global descriptor table register) or SLDT (store local descriptor table register) instructions. SGDT requires a 6-byte destination operand in memory. The destination for SLDT is a word-size register or memory operand.

Two requestor privilege level (RPL) bits are used in protection checks to determine if access to the segment is allowed. Selectors may be loaded into any of the six segment registers (CS, DS, ES, FS, GS, and SS). A selector that has an index value of zero and points to the GDT is called a *null selector*. This selector value is reserved to provide a method of initializing segment registers, because any access using a null selector generates an exception.

## Segment Descriptors

A selector points to one of 8,192 segment descriptors stored in the GDT or LDT. The structure of a segment descriptor is shown in Figure 14.3. The segment descriptor contains a 32-bit base address that specifies the beginning of the segment of memory controlled by the descriptor. The size of the segment is indicated by a 20-bit limit field and the state of the G (granularity) bit. When G is clear, the limit bits represent a segment size up to 64KB. Any attempt to access a memory location outside the limit generates an exception.

When the granularity bit is set, the limit bits represent the number of 4KB pages contained in the segment. This allows the size of a segment to range from 4KB to 4GB!



**FIGURE 14.3** Segment descriptor format

Two descriptor privilege level (DPL) bits specify the privilege level required to access the segment. An attempt by a less privileged task to use the segment results in an exception. The remaining bits are defined as follows:

P: Indicates whether the segment is present in memory. A segment-not-present exception is generated if this bit is clear when the segment descriptor is accessed.

S: When set, indicates that the segment is a system segment. When clear, the segment is a code or data segment.

D/B: For code segments, D/B controls the default operand and address size (16 bits when D/B is clear versus 32 bits when set). For data segments, D/B controls how the stack is manipulated (via SP or ESP with 16- or 32-bit pushes/pops).

AVL: Available to the programmer.

Type: This 4-bit field determines what kind of segment descriptor is being used. Table 14.1 shows the various types of segment descriptors that may be used by applications. Table 14.2 shows the different system segments that are available.

**TABLE 14.1** Segment descriptor types

| Type | Description                             |                    |
|------|-----------------------------------------|--------------------|
| 0    | Read-only                               | } Data Descriptors |
| 1    | Read-only, accessed                     |                    |
| 2    | Read/write                              |                    |
| 3    | Read/write, accessed                    |                    |
| 4    | Read-only, expand down                  |                    |
| 5    | Read-only, expand down, accessed        |                    |
| 6    | Read/write, expand down                 |                    |
| 7    | Read/write, expand down, accessed       |                    |
| 8    | Execute-only                            | } Code Descriptors |
| 9    | Execute-only, accessed                  |                    |
| A    | Execute/read                            |                    |
| B    | Execute/read, accessed                  |                    |
| C    | Execute-only, conforming                |                    |
| D    | Execute-only, conforming, accessed      |                    |
| E    | Execute/read-only, conforming           |                    |
| F    | Execute/read-only, conforming, accessed |                    |

**TABLE 14.2** System segment descriptor types

| Type | Description           | Type | Description           |
|------|-----------------------|------|-----------------------|
| 0    | Reserved              | 8    | Reserved              |
| 1    | Available 16-bit TSS  | 9    | Available 32-bit TSS  |
| 2    | LDT                   | A    | Reserved              |
| 3    | Busy 16-bit TSS       | B    | Busy 32-bit TSS       |
| 4    | 16-bit call gate      | C    | 32-bit call gate      |
| 5    | Task gate             | D    | Reserved              |
| 6    | 16-bit interrupt gate | E    | 32-bit interrupt gate |
| 7    | 16-bit trap gate      | F    | 32-bit trap gate      |

## Generating a Linear Address

When a valid descriptor is in place in the GDT or LDT, the **linear address** associated with it is generated by the process shown in Figure 14.4. The 13-bit index from the segment selector points to a segment descriptor in the GDT or LDT. The 32-bit base address from the segment is added to the 32-bit offset to create the linear address. This address is compared with the limit information to check for illegal memory references. The segment limit of a selector can be examined with the LSL (load segment limit) instruction. The segment limit of the segment specified by the source operand is loaded into a destination register.

If paging is not used, the 32-bit linear address is the same as the 32-bit physical address output on the address lines. Otherwise, the paging hardware converts the linear address into a physical address (see Section 14.3 for details).

## Privilege Levels

The RPL and DPL bits found in the segment selector and descriptor are used to perform protection checks each time an address is generated. These protection checks are based on a four-level privilege hierarchy, with level 0 being the highest level of privilege, and level 3 the lowest. Programs execute with a particular level of privilege, and therefore make memory requests (and other types of requests, such as interrupt and subroutine requests, or task switches) based on their privilege level. The privilege level of the currently executing program is called the current privilege level (CPL). The lower 2 bits of the CS register specify the CPL of the program. The CPL is compared with the RPL and DPL during address generation to enforce protection. In general, a less privileged program may not

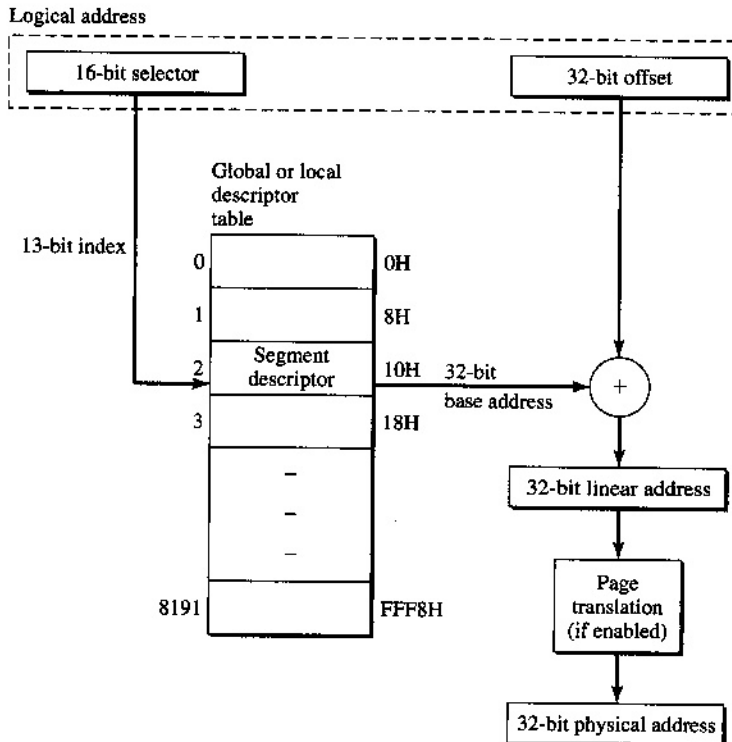
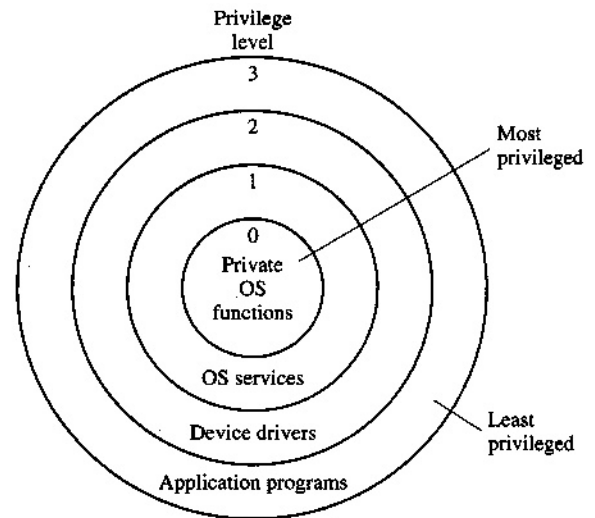


FIGURE 14.4 Segment translation



FIGURE 14.5 Rings of protection

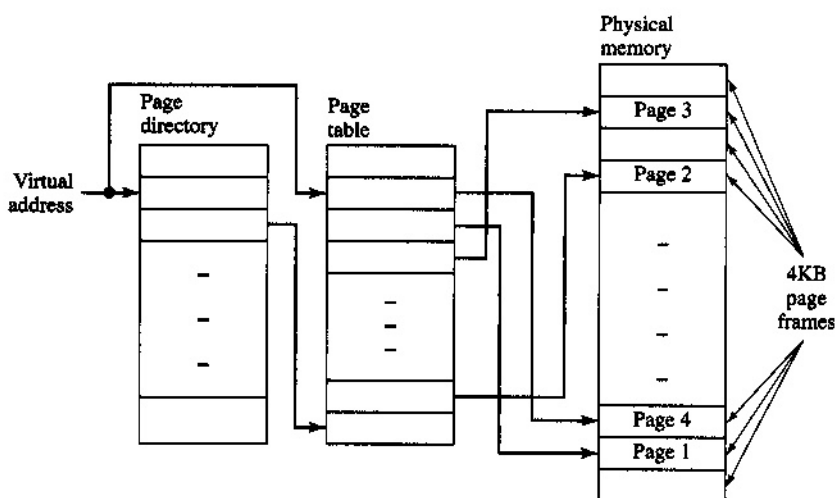


access higher privileged segments. Intel refers to the four privilege levels as *rings of protection*. As shown in Figure 14.5, a typical operating system might use privilege level 0 (the highest) for private OS functions, level 1 for OS services available to applications, level 2 for device drivers, and level 3 for application programs. This allows the operating system to have control over any code and data structures available to software running on the system. Any protection violation will cause an exception that may be serviced by the operating system. So, programs will not be able to get away with illegal memory references (overwriting important operating system data tables) or function calls (like calling a function that changes privilege levels).

A number of instructions are provided to manipulate and examine protections. These are ARPL (adjust requested privilege level), LAR (load access rights), VERR (verify segment for reading), and VERW (verify segment for writing). ARPL is used to adjust the privilege level of a selector by comparing privilege levels of source and destination operands. LAR loads a copy of the access rights of a selector containing a source operand into a destination register. VERR and VERW both compare the current privilege level with that of the source operand selector. If read or write access is allowed, the zero flag is set.

### 14.3 PAGING

The Pentium supports translation of *virtual (linear) addresses* into physical addresses through the use of special tables that map portions of the virtual address into actual physical memory locations. Figure 14.6 illustrates the general process. Physical memory is divided into fixed-size **page frames** of 4KB each. 32-bit virtual (linear) addresses generated by a running task select entries in the systems page directory and page table, which translate the upper 20 bits of the virtual address into the actual physical address where a page frame is located. The lower 12 bits of the virtual address are not translated and point to one of 4,096 byte locations within a page frame. Page translation allows the physical memory used by a system to be much smaller than the linear addressing space. For instance, the Pentium's 4GB linear addressing space may be mapped to a physical memory of only 16MB.



**FIGURE 14.6** Virtual (linear) to physical mapping

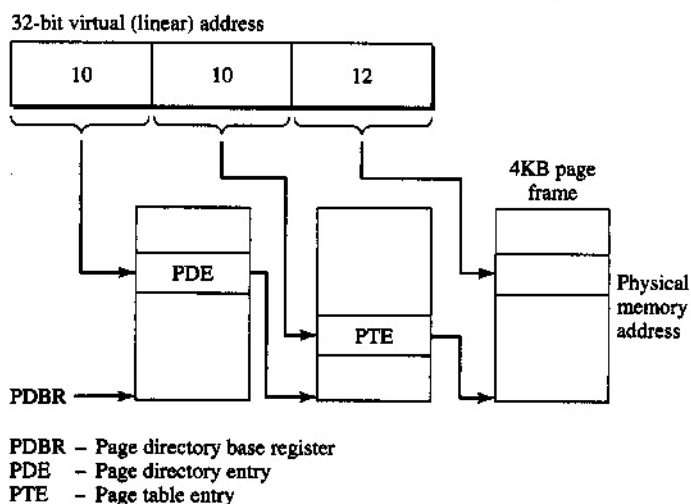
As Figure 14.6 shows, the pages used by a program do not need to be stored consecutively. A program's code and data may be spread out all over physical memory and even moved around (with help from the hard disk) while the program is executing! This helps to explain why the linear addresses are also called virtual addresses, because they have no relation to the actual physical memory address used, except for the lower 12 bits.

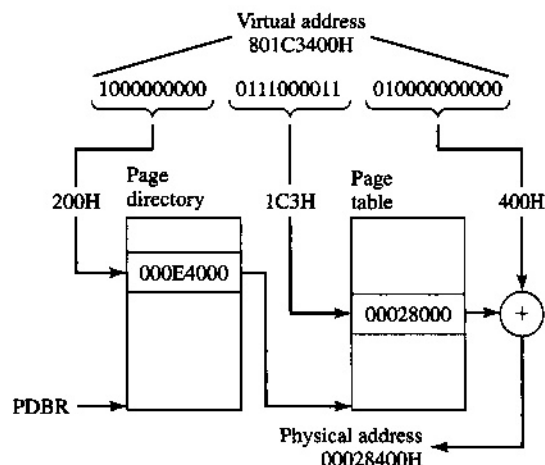
Paging is enabled when the PG bit in CR0 is set. This is a requirement for running multiple tasks in virtual-8086 mode. In addition, many operating systems employ a memory management technique called **demand paging**, which requires the kind of address translation described here.

### Page Directories and Page Tables

Figure 14.7 shows how a 32-bit virtual address is translated into a physical address. The upper 10 bits of the virtual address select one of 1,024 entries in the **page directory**. The base address of the page directory is stored in the **page directory base register (PDBR)**.

**FIGURE 14.7** Translating page addresses



**FIGURE 14.8** Actual virtual-to-physical address translation

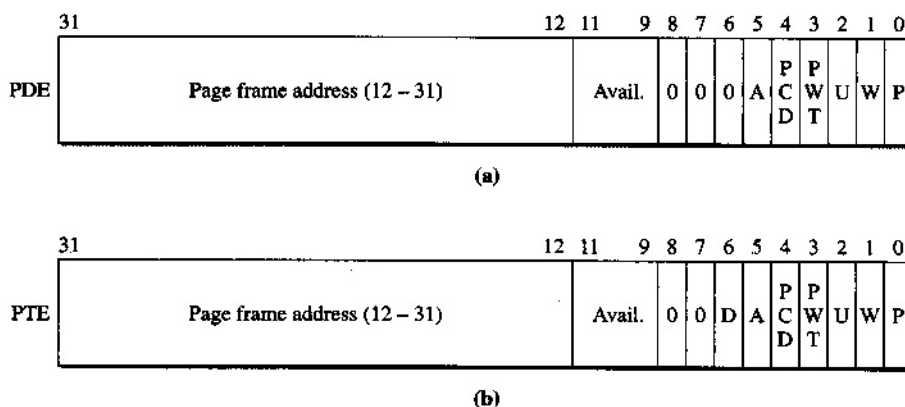
Each entry in the page directory is 4 bytes wide and contains the base address of a **page table**. The next 10 bits from the virtual address select one of 1,024 entries in the page table pointed to by the page directory entry. This entry is also 4 bytes wide and contains the base address of the actual physical memory page frame. This address is combined with the lower 12 bits of the virtual address to access the desired location in memory. Note that the page directory and page table are themselves also 4KB page frames stored in memory.

Let us examine an actual virtual-to-physical address translation. Figure 14.8 outlines the translation process. The virtual address 801C3400H is broken up into three parts. The upper 10 bits contain the value of 200H (when reinterpreted). This is the offset of the page directory entry. The PDE contains the base address 000E4000H. This is the base address of the page table.

The next 10 bits of the virtual address have a value of 1C3H, which becomes the offset into the page table. The address stored in the PTE (00028000H) is the base address of the physical memory page frame. This address is combined with the lower 12 bits of the virtual address (400H), giving a physical address of 00028400H.

### Translation Lookaside Buffers

Because the page directory and page table are 4KB pages themselves, it would be very inefficient to have to access them every time an address requires translation, as two memory reads are needed to read the entries from each table. To improve performance, the internal instruction and data caches of the Pentium contain small, special caches called **translation lookaside buffers (TLBs)** that automatically translate the upper 20 bits of the virtual address into the upper 20 physical memory address bits. The TLBs are needed because the cache must be accessed with physical, not virtual, addresses. The TLB can translate the virtual address *and* access the cache with it in a single clock cycle. Because the TLBs are small, they contain only the addresses of a few of the most recently used pages. If the required translation information is not found in the TLB, the processor accesses the page directory and page table entries stored in RAM. The operating system is responsible for loading the new translation information into the TLB. Prior to doing this, it may be necessary to invalidate the contents of the TLBs as they may contain out-of-date information. The INVLPG (invalidate TLB entry) instruction is provided for this purpose, and may only be executed in protected mode.



**FIGURE 14.9** Format of (a) page directory entry; (b) page table entry

### Page Directory Entry and Page Table Entry Formats

Figure 14.9 indicates the formats of page directory and page table entries. The upper 20 bits of each entry specify the base address of a page frame. In the PDE, this address is the base address of a page table. In the PTE, this address is the base address of the physical memory page frame.

Three bits are available for the programmer to use for any purpose, such as counting the number of times the entry is accessed. The remaining bits in each entry are defined as follows:

**D: Dirty.** This bit is set if a write has been performed to the page pointed to by the PTE. Dirty bits are used to determine if the page should be written back to hard disk when the page is swapped out (to make room for a new page coming in).

**A: Accessed.** This bit is set if a read or write was performed to the page selected by the PDE and PTE. This bit is used by the operating system to help choose a victim page to swap out when all pages are in use and a new page must be loaded into RAM. A page that has been accessed is less likely to be swapped out than a page that has not been accessed.

**PCD: Cache Disable.** This bit determines whether the current memory access is cached.

**PWT: Writethrough.** This bit enables writethrough operations between the cache and memory.

**U: User.** This bit is used when performing protection checks on the current memory address.

**W: Writeable.** This bit determines whether the page may be written to and is also used in protection checks.

**P: Present.** This bit indicates whether the page is actually stored in memory. In a demand-paging system, when a new page is needed, one of two conditions may be true:

- There is a free page frame available.
- No page frames are available.

If a page frame is available, the new page is simply copied into memory at the appropriate address, the TLBs are updated, and the P-bit is set to indicate that the page is in memory.

If no free pages exist, a victim page must be chosen to make room for the new page. The P-bit of the victim's PTE is cleared, to show that the page has been swapped out. The page may be copied back to hard disk (as required by the dirty bit) before the new page is read in.

The Pentium uses the P-bit to generate a **page fault** when attempting to access a page that is not in memory. One characteristic of a demand-paging system is that pages are only brought into memory when needed. Page faults are used to load a page into memory the first time it is needed, and to reload it if it has been swapped out.

It is interesting to note that, using demand-paged virtual memory management, all or part of many different programs may be stored in many different locations in physical memory. Page faults are used to bring in other parts (pages) of the programs as needed. Performance depends on how many pages a program is allowed to have in memory at one time.

---

## 14.4 PROTECTION

Consider a multi-user operating system based on the Pentium. Each user is capable of executing programs and using system resources, such as the hard disk, printer, and other hardware supported by the system. Now, imagine what might happen if one user's program goes out of control due to an unforeseen bug and begins writing over important operating system data structures stored in memory, or even the code and data of programs being executed by the other users. The system will most likely grind to a halt and require a complete reboot. Even worse, the problem may go undetected for a long time, causing additional bugs that may be hard to find when the initial problem is eventually discovered. This situation must not occur. Through the use of certain protection mechanisms, this catastrophe can be prevented and, possibly, even corrected before any damage occurs.

The Pentium provides protection for segmented and paged memory accesses. Protection is accomplished by comparing privilege levels during address translation. One task can be prevented from accessing code and data of another task, or even performing a task switch.

### Protecting Segmented Accesses

Prior to any memory access using segment selectors, the Pentium performs five different checks. These checks are as follows:

- Type check
- Limit check
- Addressable domain check
- Procedure entry point check
- Privileged instruction check

Any violation of these protection checks results in an exception.

Type checking is used to determine whether the current memory access (read/write) is allowed. For example, a memory write is not allowed on a read-only data segment. It may also be illegal to read from an execute-only segment. The types of accesses allowed are based on individual bits in the data and code segment descriptors. These bits are the writeable bit (data segment descriptor) and the readable bit (code segment descriptor).

Limit checking uses the twenty limit bits stored in the segment descriptor to guarantee that addresses outside the range of the segment are not generated. The granularity bit determines how the limit bits are interpreted. When the granularity is zero, the limit bits

specify the total number of byte addresses that may be used. For example, if the limit bits have been set to 01FFFFH, no addresses above xxxx1FFFFH may be generated.

When the granularity bit is one, the limit bits specify the number of 4KB pages used by the segment. Thus, a limit value of 00400H represents a segment size of 1,024 4KB pages (a total segment size of 4MB). Any attempt to generate an address above the addressable limit results in a general protection violation exception.

The addressable domain of a task is a function of the task's CPL. A CPL of 0 is the highest privilege level, and a task with a CPL of 0 may thus access data operands in segments with any privilege level. As the CPL changes to lower privilege levels (1, 2, or 3), only segments with the same, or lower, privilege level may be accessed. For example, a task with a CPL of 2 may access data segments with privilege levels of 2 or 3 only. A task with a CPL of 3 may only access data segments with a privilege level of 3. The privilege level of a segment is specified by the two DPL bits stored in its descriptor.

The procedure entry point check is performed through the use of a **call gate**. Call gates are used to control the transfer of execution between procedures of different privilege levels. The structure of a call gate is shown in Figure 14.10. The call gate has a structure similar to that used for a descriptor. The P (present) bit and DPL bits indicate the presence of the segment in memory and its privilege level. The DWORD count specifies the number of double-words to transfer from the caller's stack to the stack in the new procedure, if there is a privilege change. The calling task's CPL is compared with the DPL of the call gate to determine whether control may be transferred to the procedure entry point specified in the call gate. Call gates are used by JMP and CALL instructions and may only reside in the GDT and LDT.

Finally, some instructions are privileged and may only be executed when the CPL is 0. These instructions include:

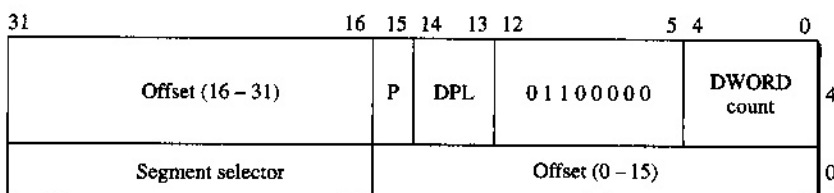
|                |      |                |        |        |
|----------------|------|----------------|--------|--------|
| CLTS           | HLT  | INVD           | INVLPG | WBINVD |
| LGDT           | LLDT | LIDT           | LMSW   | LTR    |
| MOV to/from CR |      | MOV to/from DR |        |        |

A general protection violation exception is generated if an attempt is made to execute any of these instructions with a CPL greater than 0.

## Page-Level Protection

Protection for memory pages is performed after the protection checks for segmented address generation and consists of two checks:

- Type check (reads and writes)
- Addressable domain check (via privilege levels)



**FIGURE 14.10** Structure of a call gate

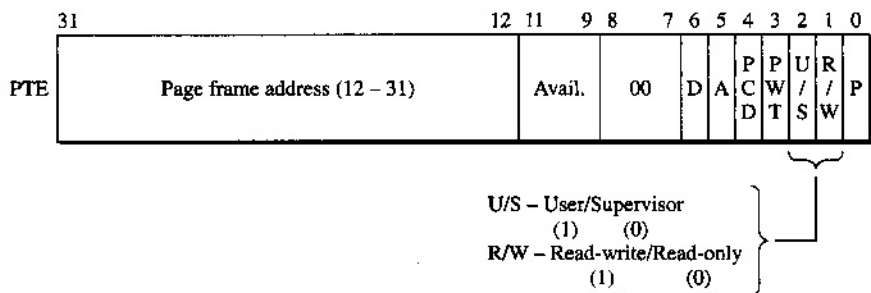


FIGURE 14.11 Page table entry protection bits

The page directory and page table entries (PDE and PTE) contain two bits that are used to perform these two checks. Figure 14.11 shows the format of a PDE or PTE. The two protection bits are U/S (user/supervisor) and R/W (read-write/read-only). Pages are marked as user (U/S equals 1) or supervisor (U/S is 0). A task is running at the supervisor level if the CPL is 0, 1, or 2. A task running at the user level (CPL equals 3) may only access a user page. A task running at supervisor level may access any page.

When R/W is 0 the page is a read-only page. A user-level task may only perform reads from the page. No user-level writes are allowed. A supervisor-level task may also read the page. If write-protection is disabled (through the WP bit in CR0) a supervisor task may write to the read-only page.

When R/W is a 1, the page is available for reads and writes. A user-level task may not read or write a supervisor-level page. If write-protection is enabled, the Pentium will catch any write to a user or supervisor page (via the page fault exception).

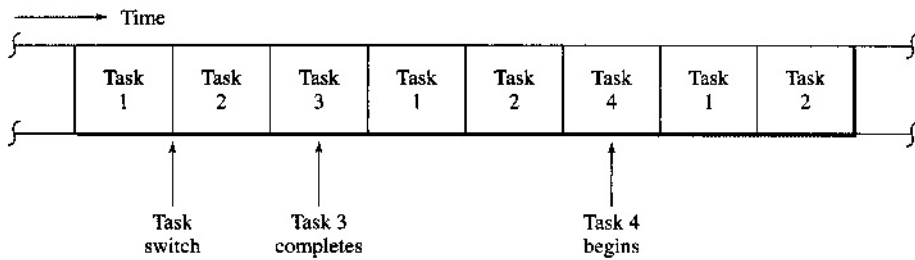
Combining segment-level and page-level protections adds a large measure of security and reliability to a Pentium-based system.

## 14.5 MULTITASKING

One of the most significant features of protected mode is its ability to support execution of multiple programs (called *tasks*) concurrently. In actuality, only one task is ever running at one point in time, because there is only one Pentium to execute on. But the ability to switch from task to task at very high speeds gives the impression that many tasks are all running at the same time. This is illustrated in Figure 14.12. Note that each task executes for a period of time (called a **time slice**) and then a **task switch** is used to switch from one task to the next. Rapidly switching from task to task gives the impression that all tasks are running at the same time.

### The Task State Segment

During a task switch, the contents of all processor registers, as well as other information, are saved for the task being suspended and new information is loaded for the next task. This information is not saved on the stack as you might expect, but in a special memory structure called the **task state segment (TSS)**. The structure of a 32-bit TSS is shown in Figure 14.13.



**FIGURE 14.12** Running multiple tasks concurrently

The TSS contains storage areas for all of the Pentium's 32-bit registers and 16-bit segment selectors, plus additional storage for the stack pointers and segment selectors for each protection level stack.

When a task is created, the task's LDT selector (offset 60H), PDBR (offset 1CH), protection level stacks, T-bit, and I/O permission bit map are filled in. During a task switch, these items are read but not changed. Only the register portion (offset 20H through 5CH) is modified during a task switch, being overwritten by the current contents of each register. These values are read during a task switch that restarts a suspended task.

**FIGURE 14.13** 32-bit TSS structure

|                      |    |    |   |      |
|----------------------|----|----|---|------|
| 31                   | 16 | 15 | 0 |      |
| Link (previous TSS)  |    |    |   | 0    |
| ESP0                 |    |    |   | 4    |
| SS0                  |    |    |   | 8    |
| ESP1                 |    |    |   | C    |
| SS1                  |    |    |   | 10   |
| ESP2                 |    |    |   | 14   |
| SS2                  |    |    |   | 18   |
| CR3 (PDBR)           |    |    |   | 1C   |
| EIP                  |    |    |   | 20   |
| EFLAGS               |    |    |   | 24   |
| EAX                  |    |    |   | 28   |
| ECX                  |    |    |   | 2C   |
| EDX                  |    |    |   | 30   |
| EBX                  |    |    |   | 34   |
| ESP                  |    |    |   | 38   |
| EBP                  |    |    |   | 3C   |
| ESI                  |    |    |   | 40   |
| EDI                  |    |    |   | 44   |
| ES                   |    |    |   | 48   |
| CS                   |    |    |   | 4C   |
| SS                   |    |    |   | 50   |
| DS                   |    |    |   | 54   |
| FS                   |    |    |   | 58   |
| GS                   |    |    |   | 5C   |
| Task LDT selector    |    |    |   | 60   |
| I/O map base address |    |    |   | T 64 |

■ - Reserved

T - Debug trap bit



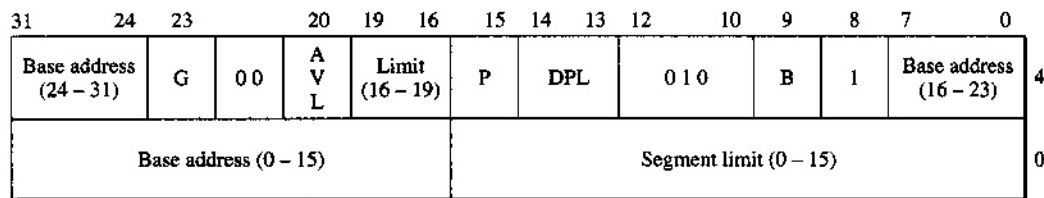


FIGURE 14.14 TSS descriptor

## TSS Descriptors

As with any segment, the TSS utilizes a descriptor that defines the various characteristics the segment will exhibit. Figure 14.14 illustrates the format of the TSS descriptor.

The individual bit fields are defined as follows:

Base address: 32-bit segment base address

Segment limit: 20-bit segment size limit

G: Granularity

AVL: Segment available

P: Segment present

DPL: Descriptor privilege level

B: Busy

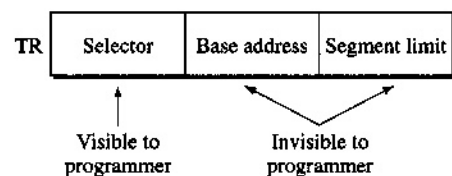
The granularity bit determines how the limit field is interpreted (size in bytes or size in 4KB chunks). When G is clear, the limit field represents a segment size from 1 byte to 1MB. When G is set, the segment size goes from 4KB to 4GB (4,096MB, in chunks of 4KB). If the segment is available for use, the AVL bit will be set. The present bit indicates whether the segment is actually in memory or not (possibly having been swapped out during a page fault). The 2-bit DPL field indicates the privilege level of the segment and is used in protection checking. The busy bit indicates that the task is currently running or waiting to run when high.

## The Task Register

TSS descriptors may only be loaded into the GDT. When multiple TSS descriptors exist in the GDT, the TSS currently in use is accessed through the use of the **task register**. The task register is used as an index pointer into the GDT to locate a TSS descriptor. The format of the task register is shown in Figure 14.15.

The task register contains two parts: a visible portion accessible by the programmer, and an invisible portion that is automatically loaded with information from the associated TSS descriptor. The task register may be loaded with a new TSS selector with the LTR (load task register) instruction. LTR requires a 16-bit register or memory operand and may

FIGURE 14.15 Task register format



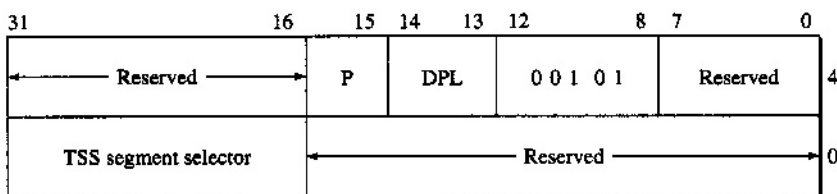


FIGURE 14.16 Format of a task gate

only be executed in protected mode with a CPL of 0. Initially, the task register is loaded with the first protected-mode task to execute via LTR. Then, during a task switch, the task register is changed to reflect the new TSS being used.

The visible portion of the task register may be read with the STR (store task register) instruction. Only the 16-bit selector portion visible to the programmer may be stored. STR may only be executed in protected mode.

### Task Gates

Because the TSS descriptor contains two DPL bits that specify the privilege level of the segment, a task switch may result in a privilege violation if the new task has a lower priority than the currently executing task. In addition, it may be necessary for an interrupt or exception to cause a task switch to a segment containing the handler code. The Pentium provides **task gates** as an additional way to facilitate task switching. Task gates may be stored in the LDT of a task, or in the IDT (see Section 14.6). The format of a task gate is illustrated in Figure 14.16.

Task gates allow a single busy bit to be used for a segment (the one contained in its TSS descriptor). Even though many different tasks might have access to a segment through their respective task gates, only one TSS descriptor is required for the segment. For example, suppose that a TSS descriptor points to the handler code for the divide error exception. Because many different tasks may generate this exception, each will require a task gate in its LDT to access the TSS descriptor of the divide error handler.

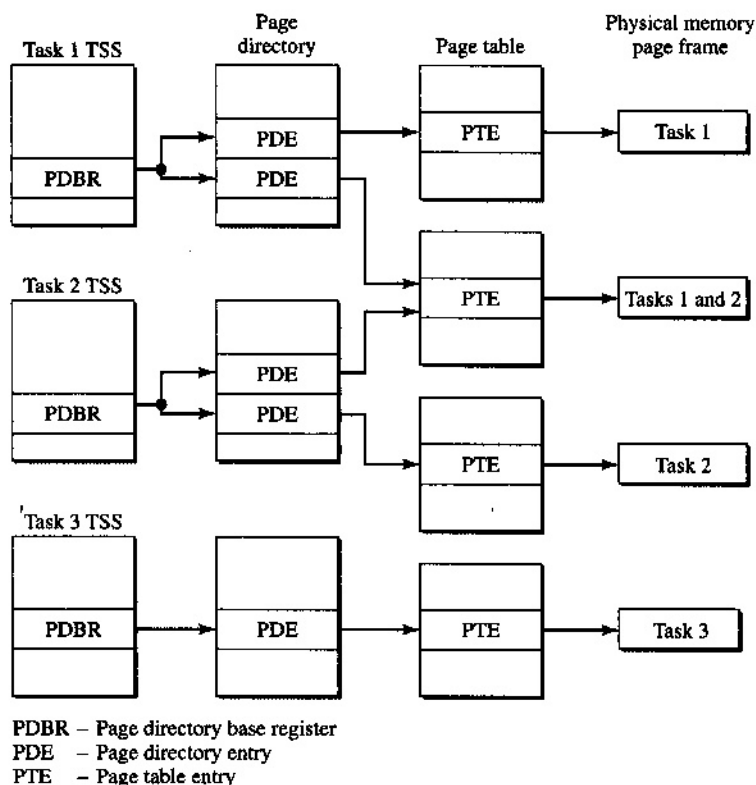
### Task Switching

Switching from one task to another is accomplished in four different ways:

- The current task JMPs or CALLs a TSS descriptor.
- The current task JMPs or CALLs a task gate.
- The current task executes an IRET when the NT flag is set.
- An interrupt or exception selects a task gate.

When a task switch is called for, the following steps take place:

1. The new TSS descriptor or task gate must have sufficient privilege to allow a task switch. The DPL, CPL, and RPL values are compared before any further processing takes place. Interrupts and exceptions do not force protection checking.
2. The new TSS descriptor must have its present bit set and have a valid limit field.
3. The state of the current task (also called its **context**) is saved. This involves copying the contents of all processor registers into the TSS for the current task.
4. The task register is loaded with the selector of the new TSS descriptor.
5. The busy bit in the new TSS descriptor is set, as is the TS bit in CR0.
6. The state of the new task is loaded from its TSS and execution is resumed.

**FIGURE 14.17** Logical to physical address mapping in multiple tasks

The selector for the old task's TSS descriptor is copied into the TSS of the new task to facilitate a return when tasks are nested (NT flag is set). An IRET instruction checks the NT flag to determine if the previous TSS selector may be used during a task switch.

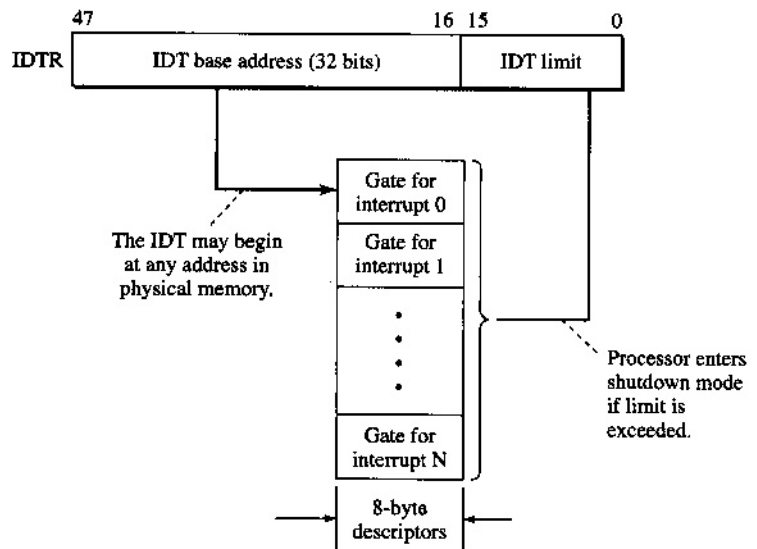
The TS bit in CR0 that is set during a task switch may be cleared by executing the CLTS (clear task switched flag) instruction.

### Task Addressing Space

If paging is not enabled, the linear addresses generated by a task are the same as the physical addresses sent to the memory system. When paging is enabled, it is possible for each task to have its own separate, protected addressing space, through the use of the PDBR stored within each TSS. As Figure 14.17 indicates, tasks may map their logical addresses into different, or overlapping, physical memory spaces. Overlapping (or *shared*) physical addresses are useful for providing the same information to many different tasks (such as the contents of DOS's interrupt vector table).

## 14.6 EXCEPTIONS AND INTERRUPTS

In this section, we will examine the operation of interrupts and exceptions in the Pentium's protected mode. An interrupt is generated in response to a hardware request on the INTR or NMI inputs, whereas an exception is generated during the course of execution. For example, divide-error is an exception generated when the Pentium's DIV or DIVI instructions are executed with a divisor operand of 0.

**FIGURE 14.18** Using the IDTR to access the IDT

Let us examine how interrupts and exceptions are supported.

## The Interrupt Descriptor Table

Real mode uses a 1KB interrupt vector table (IVT) beginning at address 00000H. Each 4-byte entry in the IVT consists of a CS:IP pair that specifies the address of the first instruction in the interrupt service routine. An 8-bit vector number is shifted 2 bits to the left to form an index into the IVT.

Protected mode relies on an **interrupt descriptor table (IDT)** to support interrupts and exceptions. The IDT comprises 8-byte gate descriptors for task, trap, or interrupt gates. The IDT has a maximum size of 256 descriptors. The size of the IDT is controlled by a 16-bit limit value stored in the **interrupt table descriptor register (ITDR)**. This 48-bit register contains the 32-bit base address for the IDT and the 16-bit size limit. Figure 14.18 shows how the IDTR is used to locate the IDT, which can be placed anywhere in physical memory.

The 8-bit vector number for the currently recognized interrupt is shifted 3 bits to the left and used as an index into the IDT. Thus, vector 10H accesses the descriptor at offset 80H in the IDT.

The LIDT (load interrupt descriptor table register) and SIDT (store interrupt descriptor table register) instructions are used in conjunction with the IDTR. Each instruction uses a single operand that specifies the address of a 48-bit memory word. This 6-byte word is used to change the location of the IDT or to find its current address. LIDT may only be executed when the CPL is 0. SIDT can be executed anytime.

## READIDTR: Reading and Displaying the IDTR

The READIDTR program uses the SIDT instruction to store a copy of the contents of the IDTR in memory, where it is then examined and converted into printable form.

```
;Program READIDTR.ASM: Read and display contents of IDTR
;
.MODEL SMALL
.586
.DATA
```

```

IDTRA    DW    ?           ;storage for limit bits
IDTRB    DD    ?           ;storage for IDT base address
MSG       DB    'The IDTR contains '
          DB    8 DUP(?)
          DB    ':'
          DB    4 DUP(?)
DBASE     DB    0DH,0AH,'$'
HEXTAB    DB    '0123456789ABCDEF'

        .CODE
        .STARTUP
SIDT      IDTRA             ;store IDTR
MOV       AX,IDTRA          ;load limit bits
LEA       SI,DBASE-1        ;set up conversion pointer
CALL      CONV              ;convert limit bits
DEC       SI                ;skip over ':'
MOV       EAX,IDTRB         ;load IDT base address
CALL      CONV              ;convert 32-bit address
CALL      CONV
LEA       DX,MSG            ;display results
MOV       AH,9
INT       21H
        .EXIT

CONV      PROC    NEAR
          MOV     CX,4        ;prepare for 4 passes
DIGIT:    MOV     DI,AX        ;get a copy of input
          AND     DI,000FH     ;mask out offset
          MOV     BL,HEXTAB[DI] ;load corresponding ASCII code
          MOV     [SI],BL      ;save it in buffer
          DEC     SI           ;adjust pointer to buffer
          SHR     EAX,4        ;get next hex digit
          LOOP    DIGIT       ;and repeat
          RET
CONV      ENDP

        END

```

When READIDTR is executed from DOS, the output looks like this:

The IDTR contains 00000000:03FF

Inside Windows, running READIDTR gives a different result:

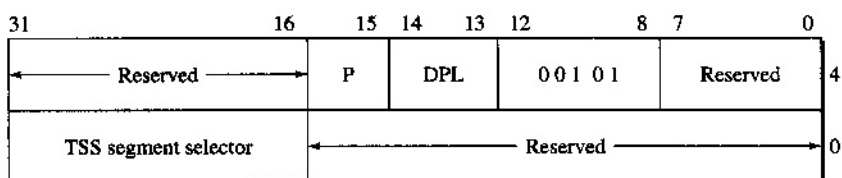
The IDTR contains 8000DA70:02FF

This indicates that Windows, which runs in protected mode, has changed the nature of the underlying interrupt system.

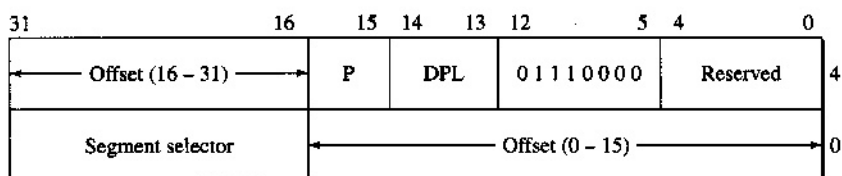
## IDT Descriptors

There are three descriptors that may be used within the IDT: task gates, trap gates, and interrupt gates. The format of each descriptor is shown in Figure 14.19.

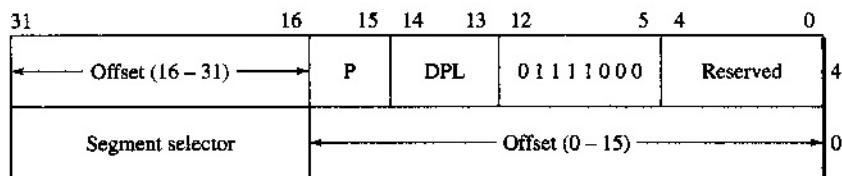
The P-bit in each descriptor stands for present and indicates whether the segment is present in memory. The 2-bit DPL field specifies the descriptor privilege level (0 is the highest). The 32-bit offset points to the first instruction in the handler's code segment. The



(a) Task gate



(b) Interrupt gate



(c) Trap gate

**FIGURE 14.19** IDT descriptors

segment selector points to an executable segment selector in the GDT or LDT for interrupt and trap gates. The TSS segment selector for a task gate points to a TSS descriptor in the GDT. Interrupt and trap gates operate like a CALL to a call gate. Task gates operate like a CALL to a task gate. Once again, there may be up to 256 descriptors in the IDT. When fewer interrupts/exceptions are required, the limit field of the IDTR is used to specify the addressable limit within the IDT. The Pentium will enter shutdown mode if the limit is exceeded.

## Interrupt and Exception Descriptions

Table 14.3 summarizes the protected-mode interrupts and exceptions available on the Pentium. Descriptions of these interrupts and exceptions are as follows:

**Vector 0: Divide Error.** This exception is generated when the divisor is 0 in a DIV or DIVI instruction.

**Vector 1: Debug Exception.** This exception has multiple uses during debugging. It is used for single-stepping, instruction, data, and task switch breakpoints.

**Vector 2: NMI Interrupt.** This interrupt is generated upon recognition of a rising edge on the Pentium's NMI input. NMI cannot be disabled through software (as the INTR interrupt can).

**Vector 3: Breakpoint.** This 1-byte instruction (opcode CCH) can be used to trigger a debugging routine by replacing the first byte of an instruction (in RAM) with the CCH opcode. The breakpoint handler is responsible for replacing the original byte of the instruction modified when the breakpoint was set up.

**TABLE 14.3** Protected-mode interrupts and exceptions

| Vector | Description          | Error Code           |
|--------|----------------------|----------------------|
| 0      | Divide error         | No                   |
| 1      | Debug exception      | No                   |
| 2      | NMI interrupt        | No                   |
| 3      | Breakpoint           | No                   |
| 4      | Overflow             | No                   |
| 5      | Bounds check         | No                   |
| 6      | Invalid opcode       | No                   |
| 7      | Device not available | No                   |
| 8      | Double fault         | Yes, 0               |
| 10     | Invalid TSS          | Yes                  |
| 11     | Segment not present  | Yes                  |
| 12     | Stack fault          | Yes                  |
| 13     | General protection   | Yes                  |
| 14     | Page fault           | Yes (special format) |
| 16     | Floating-point error | No                   |
| 17     | Alignment check      | Yes, 0               |
| 18     | Machine check        | Depends on CPU model |
| 19–31  | Reserved             | —                    |
| 32–255 | Maskable interrupts  | No                   |

**Vector 4: Overflow.** This exception is called when the INTO instruction is executed with the overflow flag set. Recall that the overflow flag is modified in accordance with the signed results of arithmetic and logical instructions.

**Vector 5: Bounds Check.** The BOUNDS instruction calls this exception when it detects an array subscript out of range.

**Vector 6: Invalid Opcode.** Any opcode not recognized by the instruction decoder generates this exception. In addition, using the wrong operand size in an instruction (presumably via self-modifying code), or using the LOCK prefix with the wrong instructions also causes an invalid opcode exception.

Opcodes D6H and F1H do not generate this exception, even though they have no designed function. They are reserved by Intel.

**Vector 7: Device Not Available.** Two bits in CR0 (EM and MP) are used to control when, if at all, an ESC or WAIT instruction generates this exception. On earlier 80x86 machines, this exception was used to indicate that there was no external floating-point coprocessor interfaced to the CPU.

**Vector 8: Double Fault.** When two exceptions occur in sequence (the second one detected while the first is being processed), there are some combinations that cause a double fault to be signaled. Double-fault exceptions are reserved for the most severe sequences, such as a page fault (Vector 14) followed by a second page fault. Interrupts and exceptions are classified into three categories: *benign* interrupts and exceptions, *contributory* exceptions, and *page faults*. Table 14.4 shows the respective classifications by vector.

Whether a double-fault interrupt is generated depends on the classification of both exceptions in the sequence, as indicated by Table 14.5. If any other exception is signaled during processing of a double fault, the processor enters shutdown mode.

**TABLE 14.4** Interrupt/exception classifications

| Class                            | Vector | Description          |
|----------------------------------|--------|----------------------|
| Benign interrupts and exceptions | 1      | Debug exceptions     |
|                                  | 2      | NMI interrupt        |
|                                  | 3      | Breakpoint           |
|                                  | 4      | Overflow             |
|                                  | 5      | Bounds check         |
|                                  | 6      | Invalid opcode       |
| Contributory exceptions          | 7      | Device not available |
|                                  | 16     | Floating-point error |
|                                  | 0      | Divide error         |
|                                  | 10     | Invalid TSS          |
|                                  | 11     | Segment not present  |
| Page faults                      | 12     | Stack fault          |
|                                  | 13     | General protection   |
|                                  | 14     | Page fault           |

**TABLE 14.5** Generating a double fault

|                 |              | Second Exception |              |            |
|-----------------|--------------|------------------|--------------|------------|
|                 |              | Benign           | Contributory | Page Fault |
| First Exception | Benign       | No               | No           | No         |
|                 | Contributory | No               | Yes          | No         |
|                 | Page fault   | No               | Yes          | Yes        |

**Vector 9: Reserved.** This vector was previously used to signal a page fault during transfer of a 387 coprocessor operand. It is not available on the Pentium.

**Vector 10: Invalid TSS.** This exception is generated when a problem is detected with a new TSS during a task switch. Depending on the error, the exception may be signaled before or after the task switch.

**Vector 11: Segment Not Present.** This exception is generated when the present bit in the current descriptor is clear. This indicates that the segment is not in memory and must be reloaded (from the hard drive). This exception is useful for virtual memory implementation.

**Vector 12: Stack Fault.** A stack fault is signaled when the limit of the SS selector is reached during execution of stack-based instructions PUSH, POP, ENTER, and LEAVE. Instructions that use SS as a segment override, or that use the BP register to reference memory, will also generate a stack fault if the limit is reached.

A stack fault is also generated when the present bit of a new descriptor for SS is clear.

**Vector 13: General Protection.** This exception is the result of many different conditions that may arise. All of the following will cause a general protection exception:

- Exceeding the segment limit with CS, DS, ES, FS, or GS.
- Reading from an execute-only code segment.
- Writing to a read-only data or code segment.



- Loading a segment register with an inappropriate segment selector (such as loading DS with an execute-only segment).
- Switching to a busy task.
- Privilege violations.
- Exceeding the instruction length limit.
- Loading CR0 with improper PE/PG combination.
- Using the wrong interrupt handler when leaving virtual-8086 mode.

Windows users know that this is a bad exception to get when running a program inside Windows. Usually, Windows must be restarted to get things back to normal after a general protection exception.

**Vector 14: Page Fault.** A page fault exception is generated when the processor attempts to access a page that is not in memory. A page fault occurs to let the operating system know that the page must be loaded into memory. The present bit in the page directory or page table entry, when clear, indicates the page is not in memory.

A page fault exception is also generated when page-level privilege is violated (the privilege is not high enough).

The Pentium pushes a 32-bit error code onto the stack of the page fault exception handler, whose meaning is described in Figure 14.20.

The error code allows the operating system to respond to the page fault in various ways. For example, the P-bit determines whether the page must be read in from the hard disk.

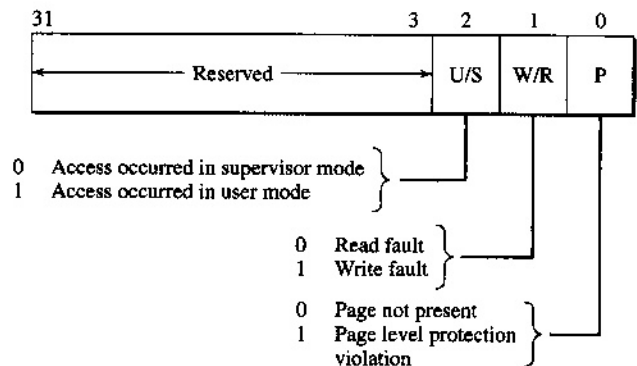
**Vector 16: Floating-Point Error.** This exception is generated if the NE bit is set in CR0 and an FPU instruction causes an error. The handler for floating-point error may adjust the operation of the FPU accordingly (via the FPU's control word).

**Vector 17: Alignment Check.** This exception is generated when a memory operand larger than 1 byte begins at an odd address, or at an even address that is not the proper multiple of 2, 4, 8, etc. For example, a word beginning at address 1001H (must be 1000H or 1002H instead), or a double-word beginning at address 1001H, 1002H, or 1003H (1000H and 1004H are acceptable) generate this exception.

To enable alignment checking, the AM bit (in CR0) must be set. Then, if the AC flag is set and the CPL is 3, alignment check will be generated.

**Vector 18: Machine Check.** This exception may or may not exist, depending on the model of the CPU. The CUID instruction returns a bit that indicates the status of this exception (available, not available).

**FIGURE 14.20** Page fault error code



## Additional Interrupt and Exception Descriptions

The remainder of the usable interrupts in Table 14.3 (32 through 255) are called maskable interrupts. These interrupts may be generated internally by software (via INT 32 through INT 255) or externally through an 8-bit vector number supplied with a hardware INTR request. Because the INTR signal may be masked by the interrupt enable flag, these hardware interrupts may be masked.

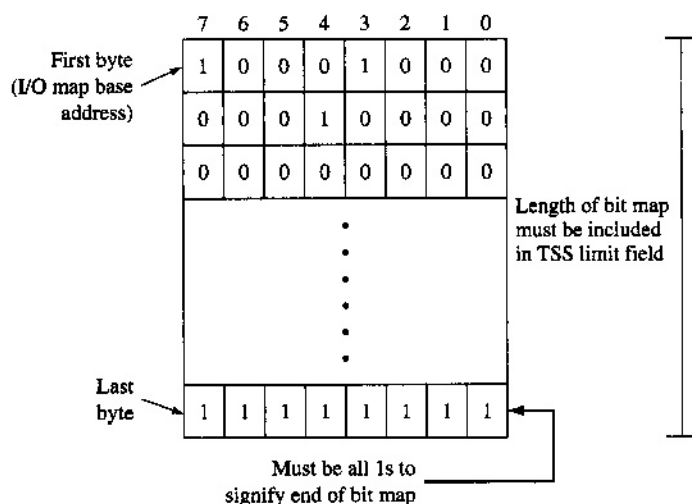
## 14.7 INPUT/OUTPUT

The Pentium provides protection for input and output operations that take place in protected mode or virtual-8086 mode. When an IN or OUT instruction executes, the processor checks the task's CPL against the IOPL (Input/Output Privilege Level) bits stored in the EFLAGS register. If the CPL is less than or equal to the IOPL, the operation is allowed. If CPL is greater than IOPL, a general protection violation exception is generated.

It is important to protect I/O operations in an operating system, because many hardware features depend on proper I/O settings. For example, an operating system might employ a counter/timer peripheral mapped to a few I/O ports that control the rate of special timing interrupts (such as a multitasking time slice interrupt). If no protection is employed, any user program may change the timer interrupt at will, causing havoc for the rest of the users. It is better to restrict I/O operations to privileged users, and force user programs to request I/O operations from the operating system. The operating system can then decide which requests to honor to keep things running smoothly.

When the CPL is greater than the IOPL, or when the processor is operating in virtual-8086 mode (see Section 14.8), I/O operations are allowed on a port-by-port basis via permission bits stored in the **I/O permission bit map** section of the task's TSS. The offset of the I/O permission bit map within the TSS is stored in the I/O map base section of the TSS. Each byte in the bit map stores permission bits for eight consecutive ports. The sample bit map in Figure 14.21 indicates that access to ports 3, 7, and 12 is not allowed. Any attempt to read or write these three ports will cause a general protection violation exception. The size of the bit map may vary according to the number of ports that must be protected.

**FIGURE 14.21** Sample I/O permission bit map



Sixteen- and thirty-two-bit ports must have two or four consecutive zeros in their associated bit map positions to be allowed access.

## 14.8 VIRTUAL-8086 MODE

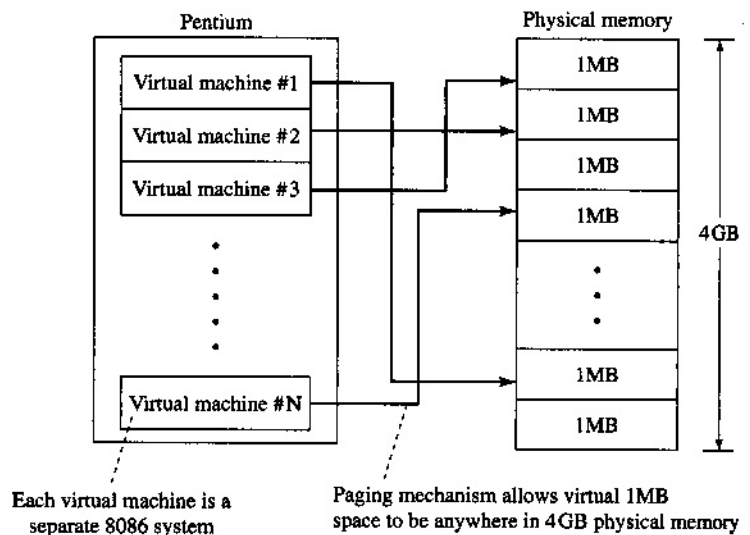
**Virtual-8086 mode** is the last of the three main operating modes of the Pentium processor. Virtual-8086 mode is entered from protected mode when the VM bit of the flag register is set, and executes programs written for the 8086 (and 8088) microprocessor. Multiple virtual-8086 programs may execute simultaneously (via the multitasking capabilities of protected mode) on **virtual machines**. A virtual machine comprises the hardware and software required to implement a particular task. Each virtual machine has its own 1MB addressing space and set of processor registers. The 1MB addressing space may be located anywhere in memory through the use of the Pentium's paging mechanism. The processor registers are maintained through their respective entries in the virtual task's TSS. As in real mode, register sizes default to 16 bits. Override prefixes may be used to allow 32-bit registers and addressing modes, with the usual restrictions. Figure 14.22 illustrates the concept of virtual machines running on the Pentium.

Address generation in virtual-8086 mode is like that of real mode. A 16-bit segment register is shifted 4 bits to the left and added to a 16-bit offset to form the desired effective address. Unlike the 8086 processor, which wraps addresses around the end of a segment from FFFFH to 0000H, a virtual-8086 task retains the carryout of the 20-bit effective address, and thus accesses a larger real-mode addressing space using 21-bit addresses. The actual range is from 000000H to 10FFEFH (FFFF0H plus FFFFH). These 21 bits are part of the 32-bit linear address used within the virtual-8086 task, and may be translated/paged to any physical address in the 4GB range of system memory.

All real-mode instructions are also available in virtual-8086 mode, though there are some differences in execution from that of the 8086. In short:

- Pentium instructions require fewer clock cycles than the 8086.
- CS:IP points to the DIV instruction, and not the following instruction, during an exception.

**FIGURE 14.22** Concept of a virtual machine



- Divide exceptions are not generated for IDIV quotients that equal 80H or 8000H.
- Undefined 8086 opcodes that represent valid Pentium instructions are executed and do not generate an invalid-opcode exception.
- The value of the SP pushed with PUSH SP is the value before it is decremented, instead of the value after it is decremented.
- Shift/rotate counts are limited to 31 bits.
- The Pentium generates an exception if a data access or instruction fetch crosses the end of a segment (offset FFFFH).
- Bits 12 through 15 of the flag register contain different values. Bit 15 is clear, and bits 12 through 15 are set according to the NT and IOPL states. On the 8086, these bits were undefined.

These differences are slight, and should not interfere with the normal operation of programs originally written for the 8086.

Virtual-8086 mode tasks always execute with a privilege level of 3 (the lowest), and may be entered/exited in a number of ways. A virtual-8086 task may be initiated by a task switch, which loads a new 32-bit TSS that has the VM bit set in its copy of the flag register. Also, a procedure with a CPL of 0 (highest priority) may execute an IRET instruction that pops a 1 into the VM bit of the flag register. This indicates that the calling procedure was a virtual-8086 task and causes the processor to reenter virtual-8086 mode.

To exit virtual-8086 mode, an interrupt or exception must be generated. If the interrupt or exception causes a task switch, the system may exit virtual-8086 mode if the new TSS is a 32-bit TSS and its copy of the VM bit is clear, or if the new TSS is only a 16-bit TSS.

If the interrupt or exception calls a procedure with a priority of 0, the processor will also exit virtual-8086 mode. Figure 14.23 shows the possible ways to enter and leave virtual-8086 mode.

As Figure 14.23 shows, interrupts and exceptions cause the processor to switch between a virtual-8086 task and a virtual-8086 **monitor task**. The monitor task is itself a protected-mode task and is responsible for initialization, interrupt and exception handling, and I/O for the running virtual-8086 task. There may be literally hundreds or thousands of virtual-8086 tasks running simultaneously. Each requires the support of a monitor task.

A handful of instructions will cause a general protection violation if executed in virtual-8086 mode with an IOPL less than 3 (recall that the CPL is always 3 in virtual-8086 mode). These instructions are CLI, STI, PUSHF, POPF, INT, and IRET. Intel calls these

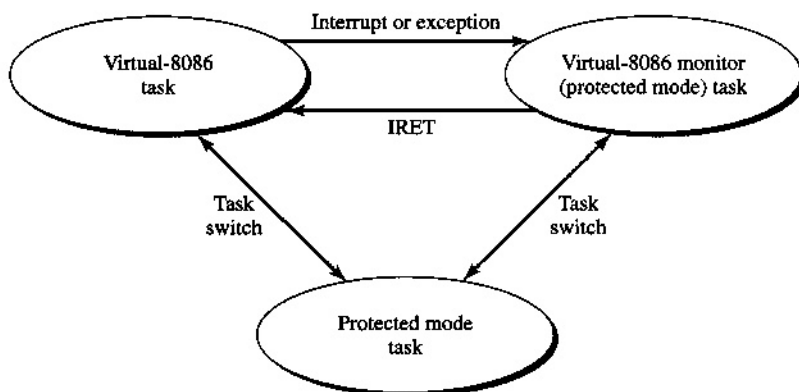


FIGURE 14.23 Entering and leaving virtual-8086 mode

instructions sensitive instructions. Sensitive instructions may need special handling by the virtual-8086 monitor.

## 14.9 A PROTECTED-MODE APPLICATION

In this final section, we will examine a program named PVIEW that has been designed to be executed in protected mode. Recall that in Section 7.11 the DPMISTAT program used function 1687H of the multiplex interrupt (INT 2FH) to detect the presence of a DPMI host (such as the one provided in the Windows environment).

If a DPMI host is present, function 1687H returns a **mode switch entry point** in registers ES:DI. Performing a CALL to the entry point places the Pentium into protected mode! The PVIEW program presented here takes advantage of the capabilities provided by the DPMI host to execute some privileged instructions, such as SGDT and STR, and display the results of the execution.

A number of new instructions are included in PVIEW. These are CPUID (CPU identification), RDTSC (read time stamp counter), and SMSW (store machine status word). The CPUID instruction requires a zero or one in EAX prior to execution and returns information such as model number and family. The RDTSC instruction reads the current value of an internal 64-bit time stamp counter that is updated every clock cycle. The SMSW instruction saves a copy of the lower 16 bits of CR0. These bits are known as the **machine status word** and can be loaded with new data using the LMSW (load machine status word) instruction. Do not play with the bits in the machine status word unless you are looking for unexpected results.

Look over PVIEW.ASM. Can you locate the protected instructions?

```
;Program PVIEW.ASM: View selected protected-mode information.
;
.MODEL SMALL
.586P
.DATA

NODPMI DB 'DPMI host missing.',0DH,0AH,'$'
NOMEM DB 'Cannot allocate memory.',0DH,0AH,'$'
NOPM DB 'Cannot switch to protected mode.',0DH,0AH,'$'

DPMI DW ?,?
HEXTAB DB '0123456789ABCDEF'
TSCLO DD ?
TSCHI DD ?
MSW DW ?
CPUMOD DW ?
TASKREG DW ?
GDT DD ?
GDTB DW ?
IDT DD ?
IDTB DW ?
LDT DW ?

TSMMSG DB 'Time Stamp Counter : '
DB 8 DUP(?)
DB ':'
DB 8 DUP(?)
```

```

TSEND  DB      0DH,0AH
        DB      'Machine Status Word : '
        DB      4 DUP(?)
MSWB   DB      0DH,0AH
        DB      'CPU identification : '
        DB      4 DUP(?)
CID    DB      0DH,0AH
        DB      'Task Register : '
        DB      4 DUP(?)
TRB    DB      0DH,0AH
        DB      'Global Descriptor Table Register : '
        DB      4 DUP(?)
G2     DB      4 DUP(?)
        DB      ':'
        DB      4 DUP(?)
G1     DB      0DH,0AH
        DB      'Interrupt Descriptor Table Register : '
        DB      4 DUP(?)
I2     DB      4 DUP(?)
        DB      ':'
        DB      4 DUP(?)
I1     DB      0DH,0AH
        DB      'Local Descriptor Table Register : '
        DB      4 DUP(?)
LI     DB      0DH,0AH, '$'

        .CODE
        .STARTUP
MOV     BX,DS                ;find size of code and data
MOV     AX,ES                ;in paragraphs
SUB     BX,AX
MOV     AX,SP                ;find stack size in paragraphs
SHR     AX,4
ADD     BX,AX                ;find total paragraphs
MOV     AH,4AH               ;adjust allocation
INT     21H
MOV     AX,1687H             ;get DPMI entry point
INT     2FH
OR      AX,AX                ;host present?
JNZ     NOHOST               ;not there
MOV     DPMI[0],DI           ;save entry address
MOV     DPMI[2],ES
OR      SI,SI                ;Any allocation needed?
JZ      GOPM                 ;No,so continue
MOV     BX,SI
MOV     AH,48H               ;try to allocate memory
INT     21H
JC      SORRY                ;not enough memory
GOPM:   MOV     ES,AX
MOV     AX,0                 ;choose 16-bit application
CALL    DWORD PTR DPMI       ;switch to protected mode
JNC     PMODE

LEA     DX,NOPM               ;display could not switch message
JMP     ERROR

```

```

NOHOST: LEA      DX,NODPMI      ;display no DPMI host message
        JMP      ERROR
SORRY:  LEA      DX,NOMEM      ;display not enough memory message
ERROR:  MOV      AH,9
        INT      21H
        JMP      BYE          ;and exit

PMODE:  RDTSC          ;otherwise, do some stuff in protected mode!
        MOV      TSCLO,EAX      ;save time stamp counter
        MOV      TSCHI,EDX
        SMSW      MSW          ;save machine status word
        MOV      EAX,1
        CPUID          ;get cpu identification
        MOV      CPUMOD,AX
        STR      TASKREG      ;save task register
        SGDT      GDT          ;save GDTR
        SIDT      IDT          ;save IDTR
        SLDT      LDT          ;save LDTR

        MOV      EAX,TSCLO      ;convert time stamp counter
        LEA      SI,TSEND
        CALL     CONV8
        MOV      EAX,TSCHI
        CALL     CONV8
        MOV      AX,MSW          ;convert machine status word
        LEA      SI,MSWB
        CALL     CONV4
        MOV      AX,CPUMOD      ;convert cpuid results
        LEA      SI,CID
        CALL     CONV4
        MOV      AX,TASKREG      ;convert task register
        LEA      SI,TRB
        CALL     CONV4
        MOV      AX,WORD PTR GDT ;convert 6-byte GDTR
        LEA      SI,G1
        CALL     CONV4
        MOV      AX,WORD PTR GDT[2]
        CALL     CONV4
        MOV      AX,GDTB
        LEA      SI,G2
        CALL     CONV4
        MOV      AX,WORD PTR IDT ;convert 6-byte IDTR
        LEA      SI,I1
        CALL     CONV4
        MOV      AX,WORD PTR IDT[2]
        CALL     CONV4
        MOV      AX,IDTB
        LEA      SI,I2
        CALL     CONV4
        MOV      AX,LDT          ;convert LDTR
        LEA      SI,L1
        CALL     CONV4
        LEA      DX,TMSG          ;view results
        MOV      AH,9
        INT      21H
BYE:    .EXIT

```

```

CONV4  PROC      NEAR
        DEC      SI                      ;adjust ascii pointer
        MOV      CX,4                    ;prepare for 4 passes
DIGIT:  MOV      DI,AX                    ;get a copy of input
        AND      DI,000FH                ;mask out offset
        MOV      BL,HEXTAB[DI]           ;load corresponding ASCII code
        MOV      [SI],BL                 ;save it in buffer
        DEC      SI                      ;adjust buffer pointer
        SHR      EAX,4                    ;get next hex digit
        LOOP     DIGIT                   ;and repeat
        RET
CONV4  ENDP

CONV8  PROC      NEAR
        CALL     CONV4                    ;convert lower 4 digits
        INC      SI                      ;adjust ascii pointer
        CALL     CONV4                    ;convert upper 4 digits
        RET
CONV8  ENDP

        END

```

Try executing PVIEW from DOS (boot the computer into DOS using a floppy). You should get the error message:

DPMI host missing.

Then try running PVIEW while inside Windows. The results should now look similar to this:

```

Time Stamp Counter : 000000B1:2F3F6733
Machine Status Word : 0013
CPU identification : 0F48
Task Register : 0018
Global Descriptor Table Register : 800517D8:010F
Interrupt Descriptor Table Register : 80BAB000:02FF
Local Descriptor Table Register : 00A8

```

Keep in mind that much of this information is privileged and only accessible through protected mode.

Many other functions are provided by the DPMI host, from memory management to interrupt servicing. These functions are best used by programmers skilled in operating system design. Even so, just taking a peek inside protected mode can be the beginning of greater things to come.

---

## 14.10 TROUBLESHOOTING TECHNIQUES

The complexity of protected mode poses a significant challenge to the programmer. Fortunately, there are many tools that aid in the development of protected-mode applications. These include compilers (capable of generating optimized Pentium code), code profilers (that determine where your program is spending its time), debuggers, disassemblers, and many good papers and books.



Online documentation on many aspects of protected-mode programming can be found on Intel's Web site, and in many other locations. Protected-mode programming is a favorite of computer game designers, who are always searching for new ways to increase speed, improve graphics, and control action.

To begin experimenting with protected-mode programming, download any of the DOS Extender packages available over the Web, or purchase your own. A DOS Extender allows your real-mode program to execute in protected mode, breaking the 640KB DOS barrier and opening up the entire 4GB addressing space of the processor. DOS Extenders come with example programs, the most basic being one that switches back and forth from real mode to protected mode. Be prepared to invest a good deal of time in learning how to run the processor in protected mode, and do not be surprised if your PC crashes during development.

---

## SUMMARY

In this chapter we have examined the features of the Pentium's powerful protected mode. These features included memory management through segmentation and paging, protection mechanisms for memory and I/O, interrupts, exceptions, and multitasking.

We finished with an examination of the Pentium's virtual-8086 mode, which emulates the operation of an 8086 machine, and a short trip into protected mode.

---

## STUDY QUESTIONS

1. What are the additional registers available in protected mode?
2. What is a selector? What is its purpose?
3. What information is contained in a segment descriptor?
4. What types of segment descriptors are available?
5. Explain how a linear address is generated using segment selectors and descriptors.
6. How is paging enabled?
7. How is a virtual address translated into a physical address?
8. How many bytes of physical memory can be accessed using a single page directory entry?
9. How many bytes of physical memory can be accessed using all the entries in a page directory?
10. Describe the operation of demand paging.
11. What are the dirty and accessed bits used for?
12. What is a page fault? When is a page fault generated?
13. Why bother with protection mechanisms in a multi-user operating system? Why are they needed?
14. How does the Pentium protect one task's memory space from being overwritten (or even read) by another task?
15. What two bits in a segment descriptor play a role in protection?
16. The limit bits in a segment selector have the value 047FFH. How many bytes does the segment contain if:
  - (a) the granularity bit is 0?
  - (b) the granularity bit is 1?

17. What is the purpose of a call gate?
18. What two instructions may specify a call gate to initiate a task switch?
19. What type of protection is provided by a call gate?
20. How is page-level protection implemented?
21. Can a user-level task access a supervisor page?
22. Can a supervisor-level task write to a read-only page when write-protection is disabled?
23. What is the purpose of a TSS? What does it contain?
24. How is the current TSS located?
25. What instructions are associated with the task register? Can they be executed in real mode?
26. What are the four ways to initiate a task switch?
27. What is a task gate? How does it differ from a TSS descriptor?
28. What controls the addressing space of a task?
29. What are the IDTR and IDT? How are they related?
30. The IDTR contains the value 4E0080000FFH. What is the base address of the IDT? What is the last address?
31. What kind of descriptors may be used in the IDT?
32. Give an example of a double-fault exception.
33. What value does the error code have to the operating system during a page fault exception?
34. The page fault error code is 0006H. What does this mean?
35. What is a nonmaskable interrupt? How many are there?
36. What is a maskable interrupt? How many are there?
37. What vector has an offset of C0H in the IDT?
38. What two methods are used to restrict access to I/O ports?
39. How many bytes are needed in the I/O permission bit map to protect access to ports 00H through FFH?
40. What is a virtual machine?
41. Why are so many virtual-8086 tasks possible?
42. What is required by a virtual-8086 task?
43. What is a virtual-8086 monitor?
44. What is the address range allowed for a virtual-8086 task?
45. Can virtual-8086 mode be entered from real mode?
46. How is virtual-8086 mode entered and exited?
47. What are some of the differences in instruction execution between virtual-8086 mode and the original 8086?
48. Which instructions are sensitive to IOPL in virtual-8086 mode?

---

# CHAPTER 15

---

## The Pentium II and Beyond

---

### OBJECTIVES

In this chapter you will learn about:

- The hardware and software architecture of the Pentium Pro, II, III, and 4 CPUs
- The Celeron and Xeon processors
- Intel's IA-64 Itanium architecture and performance
- Intel's Centrino mobile technology

### KEY TERMS

|                            |                            |                           |
|----------------------------|----------------------------|---------------------------|
| Centrino Mobile Technology | Hyperthreading             | Speculation               |
| Dynamic Execution          | In-Circuit Emulation (ICE) | Speculative execution     |
| EM64T                      | Loop unrolling             | Speculative loading       |
| EPIC                       | Micro-ops                  | Streaming SIMD Extensions |
| Execute Disable Bit        | NetBurst architecture      | Thread                    |
| Hotspot                    | Predication                |                           |
|                            | Register renaming          |                           |

---

### 15.1 INTRODUCTION

Intel has been very busy since the release of the first Pentium CPU in 1993. As shown in Table 15.1, a steady stream of new processors have been designed and unleashed over the years, with transistors being packed together in larger and larger quantities. What are all those transistors for? How come the heat from 10s of millions of transistors does not just melt the chip within its housing? How has Intel been able to increase clock speeds from 60 MHz to over 3.7 GHz, which is over 70 times faster?

One part of the answer lies in the manufacturing process. Reducing the process from 0.8 microns to 0.09 microns makes the internal signal paths almost ten times shorter, reducing the time for signal propagation. Also, Intel has lowered the core operating voltage of the processor from 5 V to under 1.3 V, which dramatically reduces power consumption.

**TABLE 15.1** Highlights in Intel's processor development

| <i>CPU</i>        | <i>Year</i> | <i>Clock Speed</i> | <i>Number of Transistors</i> | <i>Manufacturing Process</i> |
|-------------------|-------------|--------------------|------------------------------|------------------------------|
| Pentium           | 1993        | 60 MHz             | 3,100,000                    | 0.8 micron                   |
| Pentium MMX       | 1996        | 166 MHz            | 4,500,000                    | 0.35 micron                  |
| Pentium Pro       | 1996        | 180 MHz            | 5,500,000                    | 0.35 micron                  |
| Pentium II        | 1997        | 233 MHz            | 7,500,000                    | 0.35 micron                  |
| Celeron           | 1998        | 266 MHz            | 7,500,000                    | 0.25 micron                  |
| Pentium III       | 1999        | 450 MHz            | 9,500,000                    | 0.25 micron                  |
| Pentium 4         | 2000        | 1.4 GHz            | 42,000,000                   | 0.18 micron                  |
| Itanium           | 2001        | 733 MHz            | 25,000,000                   | 0.18 micron                  |
| Xeon              | 2001        | 1.4 GHz            | 42,000,000                   | 0.18 micron                  |
| Pentium M         | 2003        | 1.1 GHz            | 77,000,000                   | 0.13 micron                  |
| Pentium M 738     | 2004        | 1.4 GHz            | 140,000,000                  | 0.09 micron (90 nm)          |
| Pentium 4 Extreme | 2005        | 3.73 GHz           | 169,000,000                  | 0.09 micron (90 nm)          |

Other parts of the answer require us to take a detailed look at how Intel improved the operation of the instruction pipelines, cache, and floating-point units.

Intel has processors designed for desktops, laptops (mobile CPUs), high-end servers (with multi-processor support), and workstations. Each of these types of computing systems has different needs. For example, a high-end server may require a processor with a fast front-side bus, whereas a laptop may not.

This chapter is intended to provide an overview of each Intel processor family, with attention paid to similarities and differences between the families, and within each family. Details of new technological advances will also be examined. Let us begin our survey of Intel's microprocessor family tree.

## 15.2 THE PENTIUM PRO

The Pentium Pro's architectural differences with the fifth-generation Pentium make it Intel's first sixth-generation processor. Though the clock speed of the fastest Pentium Pro (200 MHz) is slower than the fastest Pentium (233 MHz), the Pentium Pro outperforms the Pentium for several reasons:

- The Pentium's 5-stage pipeline has been replaced by a 14-stage superpipeline. This allows each stage to contain less logic, which in turn allows the pipeline to be clocked faster.
- An internal level-2 cache has been added. The level-2 cache operates at the speed of the processor core. On earlier processors, the level-2 cache was located on the motherboard and ran at motherboard bus speeds, which are always slower than the processor core speed.
- Four additional address lines bring the address bus width to 36 bits, allowing a 64 GB physical address space.
- A Dual Independent Bus (DIB) architecture allows the Pentium Pro to access data in external memory (over the system bus) while simultaneously accessing the internal level-2 data cache.
- Four Pentium Pro CPUs may work cooperatively, an improvement over the Pentium, which only supports dual processors.

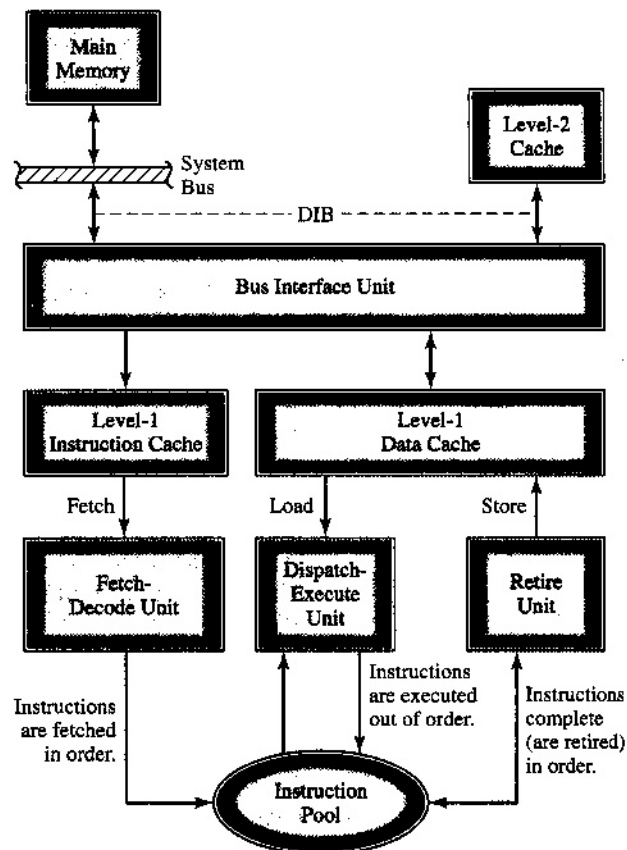
TABLE 15.2 Pentium Pro

| Year | Clock Speed                           | Cache Size                                   | Bus Speed | Manufacturing Process |
|------|---------------------------------------|----------------------------------------------|-----------|-----------------------|
| 1995 | 150 MHz, 166 MHz,<br>180 MHz, 200 MHz | 256 KB or 512KB<br>level-2 cache             | 66 MHz    | 0.6 micron            |
| 1996 | 180 MHz, 200 MHz                      | 256 KB or 512 KB<br>or 1 MB level-2<br>cache | 66 MHz    | 0.35 micron           |

- The Pentium Pro employs **speculative execution** as a portion of its **Dynamic Execution** technology, where the processor looks ahead into the instruction stream, executing instructions out of order, so that the instruction pipeline can be kept busy. To help keep track of temporary execution results, **register renaming** is used to map temporary result registers to the actual processor registers.
- The Pentium Pro improves on the Pentium's data integrity (parity checking on the data bus) by adding ECC (error checking and correction) to detect and correct single-bit errors on the data bus and detect bit errors on the address bus and control signals.

Table 15.2 lists the different Pentium Pro CPUs. Note that the processor speeds are typically three times faster than the 66 MHz bus speed. This is why moving the level-2 cache inside the CPU provides better performance. Figure 15.1 illustrates the basic flow

FIGURE 15.1 Instruction execution in the Pentium Pro



of instructions and data through the Pentium Pro. Unlike the Pentium's superscalar execution, which only tries to pair the next two instructions fetched, the Pentium Pro, in addition to being superscalar, looks ahead into the instruction pool 20 to 30 instructions, looking for instructions that can be executed out of order. Example 15.1 demonstrates this principle.

■ **EXAMPLE 15.1** Consider this sequence of instructions. The number of clock cycles assigned to each instruction is fabricated for this example.

```

1:  MOV AL,5           ;1 cycle
2:  MOV DL,[SI]        ;3 cycles
3:  MUL DL             ;2 cycles
4:  INC SI             ;1 cycle
5:  SUB BX,4           ;1 cycle
6:  ADD AX,BX          ;1 cycle
7:  MOV CX,2000        ;1 cycle

```

Executing all instructions sequentially will require ten clock cycles.

**TABLE 15.3** Scheduling instructions in order between two pipelines

| <i>Clock Cycle</i> | <i>Pipeline #1</i> | <i>Pipeline #2</i> |
|--------------------|--------------------|--------------------|
| 1                  | MOV AL,5           | MOV DL,[SI]        |
| 2                  | idle               | busy               |
| 3                  | idle               | busy               |
| 4                  | MUL DL             | INC SI             |
| 5                  | busy               | SUB BX,4           |
| 6                  | ADD AX,BX          | MOV CX,2000        |

**TABLE 15.4** Scheduling instructions out of order between two pipelines

| <i>Clock Cycle</i> | <i>Pipeline #1</i> | <i>Pipeline #2</i> |
|--------------------|--------------------|--------------------|
| 1                  | MOV AL,5           | MOV DL,[SI]        |
| 2                  | INC SI             | SUB BX,4           |
| 3                  | MOV CX,2000        | busy               |
| 4                  | MUL DL             | idle               |
| 5                  | ADD AX,BX          | idle               |

Now, suppose we are able to pair only two instructions at a time, similar to what the original Pentium was capable of doing. The schedule of instructions may look like that shown in Table 15.3. In this simple schedule, only six clock cycles are needed to execute the seven instructions, compared to the ten cycles required without scheduling. Already we can see the benefits of using two pipelines and a schedule. Unfortunately, there are several busy/idle cycles within the pipelines where nothing new is accomplished.

Table 15.4 uses the same two pipelines and seven instructions, but now the instructions may be scheduled out of order. The result is a savings of an additional clock cycle, and fewer busy/idle cycles within the pipelines. ■

Example 15.1 shows that scheduling instructions out of order can lead to a performance increase. But there will still be branches within the code that interrupt the normal flow of instructions through the pipeline (or pipelines). The Pentium Pro uses an improved branch target buffer to handle these situations.

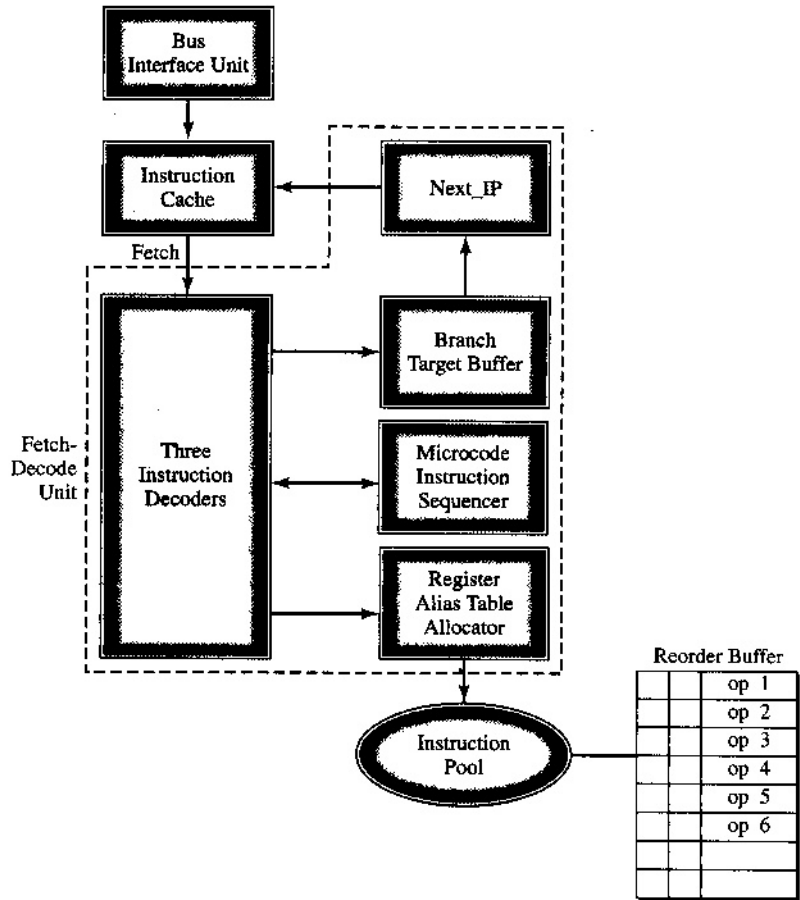
## 15.3 THE PENTIUM II

The Pentium II is essentially a Pentium Pro with MMX technology added to it. MMX stands for Multimedia Extensions, Intel's name for a set of new instructions designed to improve the performance of applications utilizing graphics and communication algorithms (such as 3D geometry, DSP filters, and audio processing). Recall that the original Pentium CPU did not have MMX technology, which was added to the architecture in later models by reusing the floating-point registers as multimedia registers (see Appendix F for details). Table 15.5 lists several Pentium II models. Unlike previous processor packages (such as the Pentium Pro's PGA), the Pentium II comes housed in a new 242-pin cartridge called a SECC (Single Edge Contact Cartridge), which contains the processor core and level-2 cache mounted on an aluminum substrate. The SECC plugs into a slot-1 connector on the motherboard. Figures 15.2 through 15.4 provide more details about the Dynamic Execution environment used in the Pentium Pro and Pentium II. This environment combines speculative execution, data flow analysis, and advanced branch prediction to achieve high instruction throughput. In Figure 15.2, we see the details of the Fetch-Decode Unit. This unit fetches instructions in program order from the Instruction cache and breaks the IA-32 instructions into smaller **micro-ops**, typically three to an instruction (two micro-op loads for source operands, one micro-op store for the destination). Complex IA-32 instructions requiring more than three micro-ops are handled by the micro-code instruction sequencer, which emits pre-designed micro-op sequences. All micro-ops flow through the register alias table allocator, which maps IA-32 registers (EAX, EBX, etc.) to temporary internal registers. The micro-ops then enter the instruction pool, also called a reorder buffer, which stores the micro-ops and their execution properties. In the Dispatch-Execute Unit, illustrated in Figure 15.3, a reservation station controls the flow of data through the integer, floating-point, MMX (not present on the Pentium Pro), and load/store units. A reservation station contains multiple entries, with each entry representing source and destination micro-op assignments for the hardware, (e.g., source register X is waiting for the integer execution unit; destination Y is waiting for the floating-point execution unit). The reservation station does not issue the micro-ops to the execution units until their operands are ready. Thus, depending on the availability of the data, micro-ops may complete out-of-sequence before being returned to the instruction pool. The reservation station allows greater on-the-fly scheduling of micro-ops than actual instructions. A maximum of five micro-ops can be scheduled at once. Micro-ops that have completed execution are extracted from the instruction pool by the Retire Unit, shown in Figure 15.4. The retire unit determines which original IA-32 register must be updated and sends the results to the retirement register file, so that registers are updated in program order. The retirements are performed correctly, even in the face of interrupts and incorrectly predicted branches, due to tags and other pieces of information stored with the micro-ops.

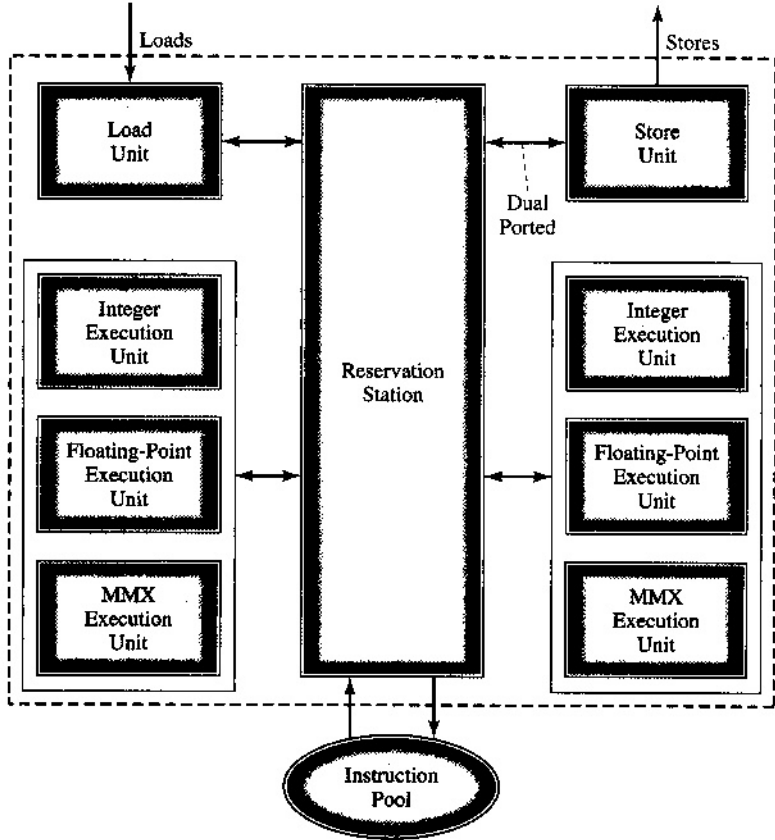
**TABLE 15.5** Selected Pentium II desktop processors

| <i>Year</i> | <i>Clock Speed</i>        | <i>Cache Size</i> | <i>Bus Speed</i> | <i>Manufacturing Process</i> |
|-------------|---------------------------|-------------------|------------------|------------------------------|
| 1997        | 233 MHz, 266 MHz, 300 MHz | 512 KB            | 66 MHz           | 0.35 micron                  |
| 1998        | 333 MHz                   | 512 KB            | 66 MHz           | 0.25 micron                  |
| 1998        | 350 MHz, 400 MHz, 450 MHz | 512 KB            | 100 MHz          | 0.25 micron                  |

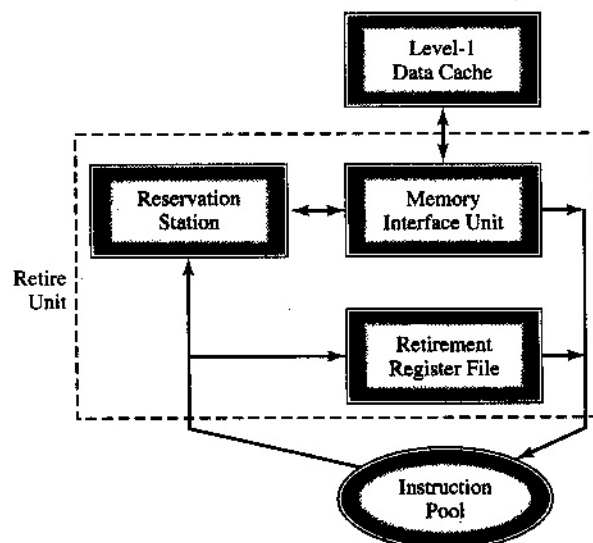
**FIGURE 15.2** Details of the Fetch-Decode Unit



**FIGURE 15.3** Details of the Dispatch-Execute Unit





**FIGURE 15.4** Details of the Retire Unit

## 15.4 THE PENTIUM III

Like the Pentium II, the Pentium III provides a Dynamic Execution environment, and utilizes the Dual Independent Bus architecture with its internal level-2 data cache. Table 15.6 lists several Pentium III models.

New to the Pentium III are Data Prefetch Logic and **Streaming SIMD Extensions (SSE)**. SIMD (single instruction, multiple data) technology enables one instruction to perform its work on multiple operands simultaneously. For example, four pairs of 16-bit numbers can be added in parallel. Appendix F provides additional details of SIMD and SSE.

The Data Prefetch Logic is used to preload the Data cache with data the application is expected to need. New instructions have been added to assist with preloading the cache. Using preloading, wait times for external data are reduced, allowing higher performance.

**TABLE 15.6** Selected Pentium III desktop processors

| Year | Clock Speed                                                            | Cache Size                             | Bus Speed          | Manufacturing Process |
|------|------------------------------------------------------------------------|----------------------------------------|--------------------|-----------------------|
| 1999 | 450 MHz, 500 MHz, 550 MHz, 600 MHz                                     | 512 KB                                 | 100 MHz            | 0.25 micron           |
| 1999 | 500 MHz, 533 MHz, 550 MHz, 600 MHz, 650 MHz, 667 MHz, 700 MHz, 733 MHz | 256 KB Advanced Transfer Level-2 cache | 100 MHz or 133 MHz | 0.18 micron           |
| 2000 | 850 MHz, 866 MHz, 933 MHz, 1 GHz                                       | 256KB Advanced Transfer Level-2 cache  | 100 MHz or 133 MHz | 0.18 micron           |

The new Streaming SIMD Extensions offer a significant improvement over the MMX technology added to the original Pentium and Pentium II. SSE adds the following features:

- Seventy additional MMX instructions. Many of these new instructions are used to work with floating-point data.
- Cache management instructions (used in prefetching and maintaining cache data).
- A set of 128-bit XMM registers (XMM0 through XMM7), capable of storing a 128-bit integer or four 32-bit floating-point numbers. These registers are in addition to the eight original floating-point registers (which also still function as the eight integer MMX registers).

Intel developed SSE to improve the performance of 3D graphical applications, which rely heavily on 3D geometry using floating-point numbers. Other applications, such as MPEG-2 encoding, speech recognition, and digital signal processing, also benefit from SSE.

The original MMX technology (see Appendix F) only contained integer MMX instructions, even though the MMX registers are mapped to the floating-point registers. With the floating-point registers already being shared between the floating-point unit and the MMX integer hardware, Intel decided to add the new 128-bit XMM registers to alleviate a potential bottleneck. SSE floating-point instructions can be executed simultaneously with IA-32 floating-point or MMX integer instructions.

The new SSE instructions are split into three groups:

1. **SIMD-FP.** New SIMD floating-point instructions allow the programmer to convert between integer and floating-point numbers, as well as perform floating-point reciprocal and square-root.
2. **New media.** These new instructions give the programmer ways to organize data for SIMD computations.
3. **Streaming memory.** Certain types of data used in a 3D application may need to remain in the cache so that they can be used over and over again in repetitive calculations (e.g., the values placed into a transform matrix), while other types of data may be used only once (e.g., a rare texture pattern on an object in a game). The new Streaming Memory instructions allow the programmer to specify the cacheability of a data object for improved performance.

Table 15.7 provides a summary of the SSE instruction set.

---

## 15.5 THE PENTIUM 4

The Pentium 4 has many architectural improvements over the Pentium III. The first is called **NetBurst® architecture**, a combination of three features:

1. **Hyper-pipelined technology.** The Pentium 4 uses a 20-stage pipeline. This allows the use of minimal logic in each stage and a higher clock speed for the pipeline. The penalty for mispredicted branches is higher, however, because more stages have to be flushed.
2. **Rapid execution engine.** The ALUs are clocked at twice the frequency of the processor core.
3. **Execution Trace Cache.** When instructions are fetched and decoded into micro-ops, a copy of the decoded instruction is placed into the Execution Trace Cache (part of the processors level-2 cache). This saves decoding time if the same instruction is fetched in the future.

**TABLE 15.7** SSE instruction set. These instructions are in addition to the original MMX instructions.

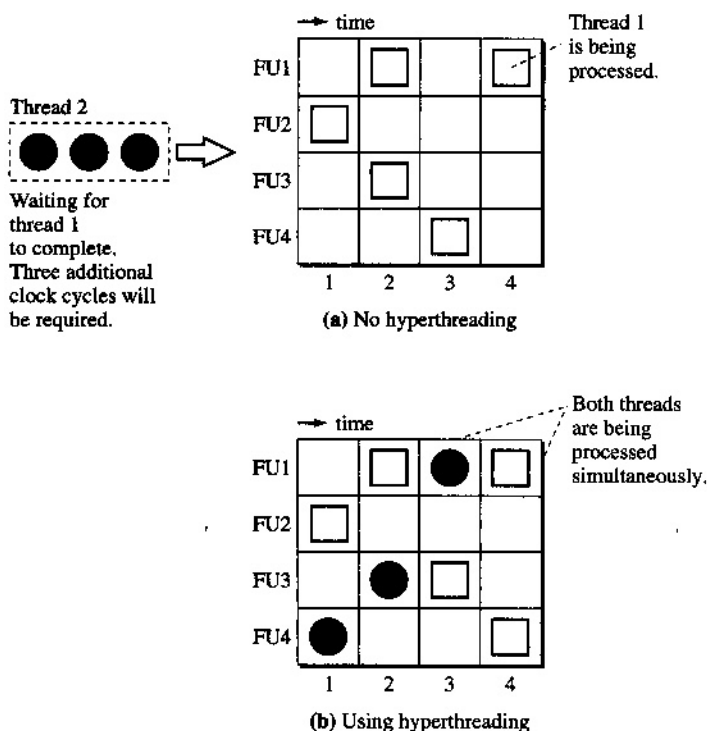
| <i>Arithmetic</i> | <i>Integer</i> | <i>Logical and Compare</i> | <i>Shuffle</i> | <i>Conversion</i> | <i>State Management</i> | <i>Cacheability Control</i> |
|-------------------|----------------|----------------------------|----------------|-------------------|-------------------------|-----------------------------|
| ADDPS             | PEXTRW         | ANDPS                      | SHUFPS         | CVTPI2PS          | LDMXCSR                 | MASKMOVQ                    |
| ADDSS             | PINSRW         | ANDNPS                     | UNPCHKPS       | CVTPI2SS          | FXSAVE                  | MOVNTQ                      |
| SUBPS             | PMAXUB         | ORPS                       | UNPCKLPS       | CVTPS2PI          | STMXSCR                 | MOVNTPS                     |
| SUBSS             | PMAXSW         | XORPS                      |                | CVTSS2SI          | FXSTOR                  | PREFETCH                    |
| MULPS             | PMINUB         | CMPPS                      |                |                   |                         | SFENCE                      |
| MULSS             | PMINSW         | CMPSS                      |                |                   |                         |                             |
| DIVPS             | PMOVMASKB      | COMISS                     |                |                   |                         |                             |
| DIVSS             | PMULHUW        | UCOMISS                    |                |                   |                         |                             |
| SQRTPS            | PSHUFW         |                            |                |                   |                         |                             |
| SQRTSS            |                |                            |                |                   |                         |                             |
| MAXPS             |                |                            |                |                   |                         |                             |
| MAXSS             |                |                            |                |                   |                         |                             |
| MINPS             |                |                            |                |                   |                         |                             |
| MINSS             |                |                            |                |                   |                         |                             |

Table 15.8 shows the progress within the Pentium 4 series. The Pentium 4 now supports **hyperthreading**, the ability to run two threads of code simultaneously. With hyperthreading, one physical Pentium 4 looks like two logical Pentium 4s to the operating system. As far as the operating system is concerned, it has two processors to work with. A Pentium 4 with hyperthreading technology has two sets of processor registers, but not two sets of hardware (the instruction pipeline is not duplicated), so the performance increase is not the same as having two physical Pentium 4s. However, Intel has shown that some server applications achieve a 30 percent improvement when using hyperthreading. Not all operating systems support hyperthreading (as support needs to be written into both the operating system and the system BIOS). Only Windows XP and selected versions of Linux can take advantage of hyperthreading technology.

But what is a thread? Simply put, a **thread** is a small portion of a program. Using techniques designed to extract parallelism from ordinary code, sections of code that are able to run independently of each other are good candidates for thread execution. For

**TABLE 15.8** Selected Pentium 4 desktop processors

| <i>Processor</i>     | <i>Year</i> | <i>Clock Speed</i> | <i>Cache Size</i>                      | <i>Bus Speed</i>   | <i>Manufacturing Process</i> |
|----------------------|-------------|--------------------|----------------------------------------|--------------------|------------------------------|
| Pentium 4            | 2001        | 1.4 GHz to 2 GHz   | 256 KB Advanced Transfer Level-2 cache | 400 MHz            | 0.18 micron                  |
| Pentium 4            | 2002        | 2 GHz to 2.8 GHz   | 512 KB Advanced Transfer Level-2 cache | 400 MHz or 533 MHz | 0.13 micron                  |
| Pentium 4 HT         | 2003        | 3 GHz              | 512 KB Advanced Transfer Level-2 cache | 800 MHz            | 0.13 micron                  |
| Pentium 4 HT         | 2004        | 3.4 GHz to 3.8 GHz | 1 MB Level-2 cache                     | 800 MHz            | 0.09 micron                  |
| Pentium 4 HT         | 2005        | 3 GHz to 3.6 GHz   | 2 MB Level-2 cache                     | 800 MHz            | 0.09 micron                  |
| Pentium 4 Extreme HT | 2005        | 3.73 GHz           | 2 MB Level-2 cache                     | 1066 MHz           | 0.09 micron                  |



**FIGURE 15.5** Executing two threads

example, when browsing the Web, you click on a link to visit a new Web page. As the Web page's HTML code is parsed, several image references are discovered. As each new image reference is processed, a thread is started to handle the image download. Thus, there will be several streams of image content being handled concurrently, rather than one at a time, in sequential order, if a single image loading process was used. Figure 15.5 illustrates the difference between executing two threads with and without hyperthreading. In Figure 15.5(a), Thread 1 requires four clock cycles to complete, with each of the four function units being used according to relationships between the operations in the thread. Thread 2 is waiting for its chance to be processed and will then require three additional clock cycles to complete. A total of seven clock cycles are needed to process both threads. In Figure 15.5(b) the operations within each thread are scheduled together, sharing the hardware, while still having their data dependencies and relationships preserved. With hyperthreading, both threads complete execution in just four clock cycles. That is a big savings.

The Pentium 4 also supports **EM64T** (Extended Memory 64 Technology), which allows 64-bit computing (via cooperation from the operating system) on integers, pointers, and registers. EM64T extends the physical address space to 1 terabyte (1024 GB) and the virtual address space to a flat 64-bit model. The following operating systems support EM64T: Windows 64-bit client OS release, Red Flag version 4.1, Novell Linux Desktop 9, and Red Hat DT 3.0. EM64T also requires 64-bit hardware (processor and chipset), and EM64T-enabled BIOS.

Also supported is the **Execute Disable Bit**, a new security feature that helps prevent malicious buffer overflow attacks, a common computer security violation technique. Areas of memory can be marked so that application code may not run in them. An attempt by a buffer overflow to execute code in marked areas will be denied, minimizing the effects of the malicious code.

**TABLE 15.9** Selected Celeron processors for desktop computers

| <i>Year</i> | <i>Clock Speed</i>   | <i>Cache Size</i>    | <i>Bus Speed</i> | <i>Manufacturing Process</i> |
|-------------|----------------------|----------------------|------------------|------------------------------|
| 1998        | 266 MHz, 300 MHz     | N/A                  | 66 MHz           | 0.25 micron                  |
| 1999        | 333 MHz to 500 MHz   | 128 KB Level-2 cache | 66 MHz           | 0.25 micron                  |
| 2000        | 533 MHz to 766 MHz   | 128 KB Level-2 cache | 66 MHz           | 0.18 micron                  |
| 2001        | 800 MHz to 1.1 GHz   | 128 KB Level-2 cache | 100 MHz          | 0.18 micron                  |
| 2002        | 1.3 GHz to 2 GHz     | 128 KB Level-2 cache | 400 MHz          | 0.13 micron                  |
| 2003        | 2.3 GHz to 2.8 GHz   | 128 KB Level-2 cache | 400 MHz          | 0.13 micron                  |
| 2004*       | 2.26 GHz to 3.06 GHz | 256 KB Level-2 cache | 533 MHz          | 0.09 micron                  |

\*These features are for the Celeron D processor

## 15.6 THE CELERON

The Celeron was originally offered as a lower-cost alternative to the Pentium II. This is why there was no level-2 cache in the original Celeron and a 66 MHz bus speed, compared to 100 MHz on the Pentium II. Though the Celeron outperformed the Pentium MMX, the lack of level-2 cache caused performance to suffer. As Table 15.9 shows, Intel quickly added 128 KB of level-2 cache in the next Celeron model. The popularity of the Celeron surpassed the Pentium II, which is clear when you compare Celeron's from Table 15.9 with Pentium II's from Table 15.5. Intel went on to greater things with the Pentium III, and continued offering improved versions of the Celeron in FC-PGA packages (part of the lower cost). In fact, the 0.18 and 0.13 micron process Celerons are similar to the Pentium III, supporting Dynamic Execution and SSE. The Celeron D processor contains support for EM64T and the Execute Disable Bit, as well as SSE3, and is similar in architecture to the Pentium 4.

## 15.7 THE XEON

The Xeon is Intel's server and workstation powerhouse. Not intended for personal computing, the Xeon processors are designed specifically to service the high-demand server and workstation market, especially dual-processor systems. Evidence of this are the large level-2 and level-3 cache sizes available. Xeon processors using the 0.13 micron process provide 32-bit processing. The 0.09 micron process CPUs deliver 64-bit computing power based on the hyperthreading and NetBurst technologies used in the Pentium 4, and support EM64T, the Execute Disable Bit, and SSE2. The Xeon MP is specifically designed for use in multi-processor systems. When coupled with high-speed workstation and server chipsets (such as the E7500 and E8500) and dual-channel DDR or DDR2 RAM, the Xeon and Xeon MP processors provide 4.8 GB/sec or higher I/O bandwidth for high-demand applications. Tables 15.10 and 15.11 list selected Xeon and Xeon MP models. Dual-core versions of the Xeon, Xeon MP, and Itanium 2 processors contain two complete processing cores, each with its own pipeline. These processors allow true simultaneous execution of two threads at the same time. Combined with hyperthreading technology, four threads are able to be processed concurrently. Two threads run concurrently in each core, but also simultaneously between the cores. Note that concurrent

**TABLE 15.10** Selected Xeon processors for workstations and servers

| <i>Year</i> | <i>Clock Speed</i>  | <i>Cache Size</i>                                    | <i>Bus Speed</i>   | <i>Manufacturing Process</i> |
|-------------|---------------------|------------------------------------------------------|--------------------|------------------------------|
| 2001        | 1.4 GHz to 2 GHz    | 256 KB Advanced Transfer Level-2 cache               | 400 MHz            | 0.18 micron                  |
| 2002        | 1.8 GHz to 2.8 GHz  | 512 KB Advanced Transfer Level-2 cache               | 400 MHz or 533 MHz | 0.13 micron                  |
| 2003        | 3.06 GHz to 3.2 GHz | 512 KB Level-2 or 1 MB Level-3 or 2 MB Level-3 cache | 533 MHz            | 0.13 micron                  |

**TABLE 15.11** Selected Xeon MP processors for workstations and servers

| <i>Year</i> | <i>Clock Speed</i> | <i>Cache Size</i>             | <i>Bus Speed</i> | <i>Manufacturing Process</i> |
|-------------|--------------------|-------------------------------|------------------|------------------------------|
| 2004        | 2.8 GHz to 3.6 GHz | 1 MB Integrated Level-2 cache | 800 MHz          | 0.09 micron                  |
| 2005        | 3 GHz to 3.6 GHz   | 2 MB Integrated Level-2 cache | 800 MHz          | 0.09 micron                  |

execution means that all threads complete execution over a period of time (taking turns), whereas simultaneous execution means the threads are executing in hardware at the same time without sharing resources.

## 15.8 IA-64 ARCHITECTURE: THE ITANIUM AND ITANIUM 2

The success of the 80x86 series of microprocessors in the ever-growing market of personal computers came with a price. Each new processor Intel introduced needed to support the huge base of customers and installed software applications already in use. So, even though great architectural advances were made in the evolution of the Pentium series, the need to support legacy software limited what Intel was able to do with the architecture. The Pentium could never be a pure RISC machine, as it had to support the CISC-based 80x86 instructions and addressing modes.

With the Itanium, Intel has embraced the newer EPIC philosophy, co-developed with HP. EPIC, for Explicitly Parallel Instruction Computing, utilizes the compiler extensively to rearrange instructions from the original program to extract and exploit parallelism. Because the compiler is scheduling the instructions, the hardware for decoding and issuing instructions can be simplified. The compiler must know explicit details about the hardware, such as the number of function units, the latency of the various pipelines, and what registers are available.

There are many compiler techniques that can be used to extract parallelism from raw code, even if there is no inherent parallelism to begin with. One technique, called **loop unrolling**, duplicates the instructions found within a loop in such a way that one pass through the new loop produces two or more results, instead of one result in each pass of the original loop. Example 15.2 illustrates the benefit of this technique.

---

**■ EXAMPLE 15.2** Consider this short loop of instructions:

```

        MOV    CX, 1000
TOP:    INC     BX
        MUL    BX
        ADD    BP, AX
        ADC    DI, DX
        LOOP   TOP

```

The same loop, unrolled twice, looks like this:

```

        MOV    CX, 500
TOP:    INC     BX
        MUL    BX
        ADD    BP, AX
        ADC    DI, DX
        INC     BX
        MUL    BX
        ADD    BP, AX
        ADC    DI, DX
        LOOP   TOP

```

In this example, it was easy to unroll the loop by duplicating the loop body and changing the initial value loaded into register CX. What is the benefit? We have reduced the number of branches during execution from 1000 to 500. Recall that branches can cause the instruction pipeline to stall as new target instructions are fetched, particularly when a branch is mispredicted. Because a stalled pipeline represents wasted computations, reducing the number of branches in the instruction stream will lead to better performance. ■

---

One way around the problem of mispredicted branches is to use a technique called **predication**. To see how predication is used, consider this simple if() statement:

```

if(value > 10)
{
    idx++;
    a = data[idx] * 2;
}
else
{
    idx--;
    a = data[idx] / 2;
}

```

Normally, the code needed to perform the condition test (`value > 5`) must be executed before either of the true/false sections of code can execute. Once the result of the test is known, then the appropriate code section is executed. This results in a waste of pipeline utilization.

With predication, the true and false sections of code are both executed, along with the condition testing code, in order to keep the pipeline busy. Instructions from the true and false sections are marked with predicate flags to identify which section they belong to. When the results of the test are known, the appropriate group of instructions is marked for completion. So, even though we calculate the results for both the true section and the false section, only one set of results will be utilized and the other discarded. The compiler builds the schedule of condition-test, true, and false instructions and sets the predicate flags. The processor uses the flags during execution to retire the appropriate section of instructions.

**TABLE 15.12** Selected Itanium processors for enterprise servers

| <i>Processor</i> | <i>Year</i> | <i>Clock Speed</i>  | <i>Cache Size</i>               | <i>Number of Transistors</i> | <i>Manufacturing Process</i> |
|------------------|-------------|---------------------|---------------------------------|------------------------------|------------------------------|
| Itanium          | 2001        | 733 MHz,<br>800 MHz | 2 MB or 4MB<br>Level-3 cache    | 25,000,000                   | 0.18 micron                  |
| Itanium 2        | 2002        | 900 MHz,<br>1 GHz   | 3 MB or 1.5 MB<br>Level-3 cache | 220,000,000                  | 0.18 micron                  |
| Itanium 2        | 2003        | 1.5 GHz             | 6 MB Level-3 cache              | 410,000,000                  | 0.13 micron                  |
| Itanium 2 LV     | 2003        | 1 GHz               | 1.5 MB Level-3 cache            | 410,000,000                  | 0.13 micron                  |
| Itanium 2        | 2004        | 1.6 GHz             | 9 MB Level-3 cache              | 592,000,000                  | 0.13 micron                  |
| Itanium 2 LV     | 2004        | 1.3 GHz             | 3 MB Level-3 cache              | 592,000,000                  | 0.13 micron                  |

The EPIC philosophy also includes the use of **speculation**, or **speculative loading**. In this technique, data is preloaded before it is needed, while other instructions are also being executed. The preloading is used to hide the latency encountered when reading data from memory. Then, when the instructions that require the data come up for execution, the data is already there, and no waiting is required. The only problem with this technique: what happens if the data changes after we preload it? The Itanium contains instructions designed to keep track of preloaded data, so that stale data is not used in a computation.

Table 15.12 lists several different types of Itanium CPUs. The number of transistors used in the chips is staggering, but when you look at the size of the level-3 cache on these processors, you can see where Intel is spending its transistor budget. The Itanium comes with 128 64-bit general purpose registers and 128 floating-point registers, which is plenty of registers to support parallel operations.

## 15.9 INTEL'S MOBILE CPUS

When you look inside a desktop computer, what do you see? A power-hungry CPU giving off so much heat that a large metal heat sink the size of a softball is needed to keep it cool with the help of a muffin fan. In addition, because the desktop computer is plugged into a wall outlet, there is no lack of available power. Last, with its permanent place on a desk or lab bench, the desktop computer may be hardwired into its LAN using a networking cable.

The situation is different for a laptop computer. Consider the following reasons:

1. With space at a minimum, there is no room for a large heat sink or a fan to blow air around and cool things off.
2. Unlike the desktop computer, which contains a power supply to convert AC into DC, the laptop computer contains a rechargeable battery, whose useable life after a charge is limited and affected by how the laptop is used.
3. Because the laptop computer is designed to be portable, there is also the problem of maintaining a network connection, because using a cable while moving around is not very convenient.



**TABLE 15.13** Selected Intel mobile processors and their power versus voltage-speed relationships

| <i>Mobile CPU</i>                              | <i>Clock Frequency</i> | <i>Core Voltage</i> | <i>Power</i>    |
|------------------------------------------------|------------------------|---------------------|-----------------|
| Pentium II                                     | 266 MHz                | 1.7 V               | 8.6 W           |
|                                                | 300 MHz                | 1.6 V               | 9.0 W           |
|                                                | 400 MHz                | 1.5 V               | 7.5 W           |
| Celeron M                                      | 1.5 GHz                | 1.356 V             | 24.5 W          |
| Ultra Low Voltage Celeron M                    | 1 GHz                  | 0.94 V              | 5 W             |
| Pentium III                                    | 700 MHz                | 1.35 V              | Less than 2 W   |
| Pentium III – Processor M                      | 1.33 GHz               | 1.15 V              | Less than 1.5 W |
| Ultra Low Voltage<br>Pentium III – Processor M | 933 MHz                | 0.95 V              | Less than 0.5 W |
| Pentium M                                      | 1 GHz                  | 0.85 V              | 7W              |

Intel's **Centrino Mobile Technology** addresses all three of these critical issues. First, and most important, Intel has paid a lot of attention to designing mobile CPUs that are more power-efficient than their desktop counterparts. Two important factors determine the power consumption of a processor: its clock speed and its operating voltage. An increase in the clock frequency or an increase in the power supply voltage will cause the power consumed to increase. So, to save power, you can lower the clock frequency or lower the operating voltage, or both. Lowering the clock frequency means less performance, so this option is not very attractive. Even so, you will notice that clock speeds of mobile processors are lower than those of similar desktop CPUs. Table 15.13 lists several of Intel's mobile CPUs. By steadily lowering the core operating voltage of the processor, Intel is able to keep power consumption down even while clock speeds are increasing. This is not done by lowering the voltage alone, but also by designing the CPU for low power using several approaches, as Intel has done with the Pentium M processor:

- Micro-ops fusion combines different CPU operations to reduce execution time and save power.
- Advanced instruction prediction uses past program behavior to make predictions about which instructions will be needed in the future for more efficient execution of instructions.
- A power-optimized processor system bus that is only powered-up when needed.
- Enhanced Intel SpeedStep® technology extends battery life by continuous monitoring of the power needs and utilization within the processor.

In addition, Intel uses new power-saving transistors and a power-aware level-2 cache to reduce processor power consumption.

The second component of Intel's Centrino technology are the 855 and 915 Express chipset families that govern the flow of information between the CPU, memory, and I/O devices (such as IDE, USB, PCI, and audio). These chipsets allow front-side bus speeds of 400 MHz, 533 MHz, or 667 MHz, and provide AGP graphics and high-speed DDR memory interfacing, as summarized in Table 15.14. Intel's GMA 900 (Graphics Media Accelerator) is their newest generation of accelerated graphics technology, containing support for Direct X 9, Open GL 1.4, and 3D graphics with four pixel pipelines for a superfast 1.3 G pixels/sec rendering speed.

**TABLE 15.14** Centrino technology chipsets

| Chipset | Processor                                                                  | FSB Speed                              | Max. Memory | Memory Type             | Graphics  |
|---------|----------------------------------------------------------------------------|----------------------------------------|-------------|-------------------------|-----------|
| 855GME  | Pentium M<br>Celeron M                                                     | 400 MHz                                | 2 GB        | 200/266/333<br>DDR      | AGP 4x    |
| 855PM   | Pentium M<br>Celeron M                                                     | 400 MHz                                | 2 GB        | 200/266/333<br>DDR      | AGP 2x/4x |
| 855GM   | Pentium M<br>Celeron M                                                     | 400 MHz                                | 2 GB        | 200/266 DDR4            | N/A       |
| 910GML  | Celeron M<br>Celeron M ULV*                                                | 400 MHz                                | 2 GB        | 400 DDR2<br>333 DDR     | GMA 900   |
| 915PM   | Pentium M<br>Pentium M LV**<br>Pentium M ULV<br>Celeron M<br>Celeron M ULV | 400 MHz<br>533 MHz<br>(Pentium M only) | 2 GB        | 400/533 DDR2<br>333 DDR | N/A       |
| 915GMS  | Pentium M LV<br>Pentium M ULV<br>Celeron M ULV                             | 400 MHz                                | 2 GB        | 400 DDR2                | GMA 900   |
| 915GM   | Pentium M<br>Pentium M LV<br>Pentium M ULV<br>Celeron M<br>Celeron M ULV   | 400 MHz<br>533 MHz<br>(Pentium M only) | 2 GB        | 400/533 DDR2<br>333 DDR | GMA 900   |

\*Ultra Low Voltage, \*\* Low Voltage

The third component of Intel's Centrino technology allows the laptop computer to connect to wireless networks (including public-access Wi-Fi networks called **hotspots**) through its 802.11 b/g-compliant Intel PRO/Wireless Network Connection hardware. Intel uses its Intelligent Scanning technology to reduce power usage while scanning for wireless access points. Secure communication at 11 Mbps (802.11b) or 54 Mbps (802.11 g), including Virtual Private Networking, is possible using WPA (Wi-Fi Protected Access), a successor to WEP, the original security standard for wireless transmissions.

In summary, Intel has invested heavily in providing low-power, high-performance mobile computing technology.

## 15.10 TROUBLESHOOTING TECHNIQUES

The art of troubleshooting microprocessor-based systems has evolved over the years. In the old days (late 70s, early 80s), you could diagnose and solve almost every problem using a multi-meter and dual-channel oscilloscope. Occasionally, a logic analyzer would be required for really tough problems, but hooking up eight or even sixteen channels was all that was ever necessary.

With the advanced microprocessors we have today, more sophisticated techniques are required. Sixteen channels on a logic analyzer would not even capture the entire data bus on most newer CPUs. On the other hand, in terms of practicality, what engineer or technician wants to spend hours connecting 32 or 64 data bus probes and 32 or more address line probes? Would you be able to connect probes to the CPU pins, or does the package

style (PGA, SOIC) prevent that? Could you even get to the CPU? Is the logic analyzer probe fast enough to capture the CPU signals, or is the CPU clock too fast? How would someone be able to troubleshoot a system with a sea of wires surrounding it?

Into this scenario jumps the world of **In-Circuit Emulation (ICE)**, where a specially designed interface is inserted between the CPU and the motherboard socket in order to tap into every pin of the processor. An ICE control module connects the interface to a PC running emulation software, which is able to monitor and control the test CPU and system. Note that this method does not often work for real-time applications, because the emulation adds its own delay to the operation of the system.

Even with newer tools and techniques, sometimes a little common sense is all that is needed. A test technician once explained how he performed a quick initial test to determine the functionality of a newly manufactured processor card. The card contained a 100 MHz Pentium CPU and a large FPGA containing all the “glue logic” required to interface the CPU with external hardware. The test technician would connect the board to a test socket, apply power briefly and note how much current the board required. If the current was low, the initial test passed, and the board went on to normal burn-in testing. If the current was high, this typically indicated the FPGA was not programmed and too many devices were incorrectly enabled at the same time. A simple test for a complex system, yet it worked. This is a good reminder that, no matter how complex a system may become, the basics will always come in handy.

---

## SUMMARY

In this chapter we examined the many architectural changes and improvements to the Pentium family of processors and also investigated Intel’s Xeon, Celeron, Itanium, and mobile processors. Each CPU family contains many different models, with varying clock speeds, cache size, front-side bus speeds, and power levels, including low voltage and ultra low voltage processors.

We also investigated the newer EPIC design philosophy behind the introduction of the Itanium family of processors. The Itanium is Intel’s first IA-64 architecture. Support for existing IA-32 applications is provided via operating system emulation and/or IA-32 hardware. EPIC techniques, relying heavily on the compiler to extract parallelism from the code, allow simpler instruction issue logic than an equivalent superscalar machine.

Finally, we explored Intel’s Centrino mobile technology, which uses low-power CPUs, chipsets, and wireless technology to improve the performance of mobile systems.

---

## STUDY QUESTIONS

1. Give several reasons why the Pentium Pro performs better than a Pentium running at the same clock speed.
2. By moving the level-2 cache inside the Pentium Pro CPU, its access time essentially decreases by a factor of three. Explain the affect on cache hits and misses, for both reads and writes (writethrough and writeback).
3. Repeat Example 15.1 for this sequence of instructions:

```
1:  MOV AL, [SI]      ;3 cycles
2:  MOV DL, [DI]      ;3 cycles
3:  MUL DL            ;2 cycles
```

```

4:  ADD SI,2           ;1 cycle
5:  DEC DI             ;1 cycle
5:  MOV [BP],AX        ;3 cycles
6:  INC BX             ;1 cycle
7:  AND AL,AH          ;1 cycle
8:  ADC AL,0           ;1 cycle

```

4. What is involved in Dynamic Execution?
5. What are micro-ops?
6. What is the main difference between the Pentium Pro and the Pentium II?
7. Explain how it is possible for instructions to execute out of order.
8. What is the purpose of a reservation station?
9. What is SSE? How does it improve the capabilities of MMX?
10. What is streamed when using SSE?
11. How does SSE improve the performance of 3D graphical applications?
12. Perform your own research and discover why advanced transfer level-2 cache is different than ordinary level-2 cache.
13. What is hyperthreading?
14. What is required to support hyperthreading?
15. What are the components of NetBurst architecture?
16. Summarize all the ways the Pentium 4 is superior to the original Pentium.
17. What is EM64T? What is its purpose?
18. What is the purpose of the Execute Disable Bit?
19. Intel has had competitors for a long time. In particular, AMD has been very innovative in developing their own architectural features compatible with IA-32 technology. Perform some research and discover how AMD has been head-to-head with Intel for each major processor release and feature (such as 3DNow!).
20. What was the original intent of the Celeron CPU?
21. What are the main differences between the Celeron and the Celeron D?
22. Does the Xeon microprocessor provide all the system performance by itself?
23. What is the difference between simultaneous execution and concurrent execution? How are these activities related to hyperthreading and dual-core processors?
24. How many threads can be processed concurrently using two Xeon MP dual-core processors?
25. If the loop code from Example 15.2 was unrolled four times instead of twice, what changes would be needed? How would this new loop take advantage of MMX or SSE instructions?
26. What does EPIC stand for and what does it involve?
27. Unroll the following loop so that the number of passes is lowered to 20:

```

MOV  BL,0
SUB  AX,AX
TOP: ADD  AL,[SI]
      ADC  AH,0
      INC  SI
      INC  BL
      CMP  BL,100
      JNZ  TOP

```

28. Suppose that 100,000,000 transistors are used to make a 2 MB cache. How many transistors/bit does that represent? What do you think all the transistors are for? Refer back to Chapter 13 for an explanation of cache hardware.
29. What are the three components of Intel's Centrino mobile technology?
30. What makes a mobile CPU different from a desktop or server CPU?

---

# APPENDIX A

---

## Instruction Execution Times

---

---

### INSTRUCTION EXECUTION TIMES

The instruction execution times listed here are for the 8088 and 8086. The newer processors list only one cycle for many instructions, due to pipelining and other techniques.

The following addressing-mode abbreviations are used throughout this section:

**Reg**—Register      **Mem**—Memory  
**Imm**—Immediate    **Acc**—Accumulator

#### Arithmetic Instructions

| <i>Instruction</i> | <i>Clock Cycles</i> | <i>Transfers</i> |
|--------------------|---------------------|------------------|
| AAA                | 4                   |                  |
| AAD                | 60                  |                  |
| AAM                | 83                  |                  |
| AAS                | 4                   |                  |
| ADC                |                     |                  |
| Reg to Reg         | 3                   |                  |
| Mem to Reg         | 9 + EA              | 1                |
| Reg to Mem         | 16 + EA             | 2                |
| Imm to Reg         | 4                   |                  |
| Imm to Mem         | 17 + EA             | 2                |
| Imm to Acc         | 4                   |                  |
| ADD                |                     |                  |
| Reg to Reg         | 3                   |                  |
| Mem to Reg         | 9 + EA              | 1                |
| Reg to Mem         | 16 + EA             | 2                |
| Imm to Reg         | 4                   |                  |
| Imm to Mem         | 17 + EA             | 2                |
| Imm to Acc         | 4                   |                  |

| <i>Instruction</i> | <i>Clock Cycles</i> | <i>Transfers</i> |
|--------------------|---------------------|------------------|
| CBW                | 2                   |                  |
| CWD                | 5                   |                  |
| DAA                | 4                   |                  |
| DAS                | 4                   |                  |
| DEC                |                     |                  |
| 8-bit              | 3                   |                  |
| 16-bit             | 2                   |                  |
| Mem                | 15 + EA             | 2                |
| DIV                |                     |                  |
| 8-bit Reg          | 80-90               |                  |
| 16-bit Reg         | 144-162             |                  |
| 8-bit Mem          | 86-96 + EA          | 1                |
| 16-bit Mem         | 150-168 + EA        | 1                |
| IDIV               |                     |                  |
| 8-bit Reg          | 101-112             |                  |
| 16-bit Reg         | 165-184             |                  |
| 8-bit Mem          | 107-118 + EA        | 1                |
| 16-bit Mem         | 171-190 + EA        | 1                |
| IMUL               |                     |                  |
| 8-bit Reg          | 80-98               |                  |
| 16-bit Reg         | 128-154             |                  |
| 8-bit Mem          | 86-104 + EA         | 1                |
| 16-bit Mem         | 134-160 + EA        | 1                |
| INC                |                     |                  |
| 8-bit              | 3                   |                  |
| 16-bit             | 2                   |                  |
| Mem                | 15 + EA             | 2                |
| MUL                |                     |                  |
| 8-bit Reg          | 70-77               |                  |
| 16-bit Reg         | 118-133             |                  |
| 8-bit Mem          | 76-83 + EA          | 1                |
| 16-bit Mem         | 124-139 + EA        | 1                |
| NEG                |                     |                  |
| Reg                | 3                   |                  |
| Mem                | 16 + EA             | 2                |
| SBB                |                     |                  |
| Reg from Reg       | 3                   |                  |
| Mem from Reg       | 9 + EA              | 1                |
| Reg from Mem       | 16 + EA             | 2                |
| Imm from Reg       | 4                   |                  |
| Imm from Mem       | 17 + EA             | 2                |
| Imm from Acc       | 4                   |                  |
| SUB                |                     |                  |
| Reg from Reg       | 3                   |                  |
| Mem from Reg       | 9 + EA              | 1                |
| Reg from Mem       | 16 + EA             | 2                |
| Imm from Reg       | 4                   |                  |
| Imm from Mem       | 17 + EA             | 2                |
| Imm from Acc       | 4                   |                  |

## Logical Instructions

| <i>Instruction</i> | <i>Clock Cycles</i> | <i>Transfers</i> |
|--------------------|---------------------|------------------|
| <b>AND</b>         |                     |                  |
| Reg to Reg         | 3                   |                  |
| Mem to Reg         | 9 + EA              | 1                |
| Reg to Mem         | 16 + EA             | 2                |
| Imm to Reg         | 4                   |                  |
| Imm to Mem         | 17 + EA             | 2                |
| Imm to Acc         | 4                   |                  |
| <b>CMP</b>         |                     |                  |
| Reg to Reg         | 3                   |                  |
| Mem to Reg         | 9 + EA              | 1                |
| Reg to Mem         | 9 + EA              | 1                |
| Imm to Reg         | 4                   |                  |
| Imm to Mem         | 10 + EA             | 1                |
| Imm to Acc         | 4                   |                  |
| <b>NOT</b>         |                     |                  |
| Reg                | 3                   |                  |
| Mem                | 16 + EA             | 2                |
| <b>OR</b>          |                     |                  |
| Reg to Reg         | 3                   |                  |
| Mem to Reg         | 9 + EA              | 1                |
| Reg to Mem         | 16 + EA             | 2                |
| Imm to Reg         | 4                   |                  |
| Imm to Mem         | 17 + EA             | 2                |
| Imm to Acc         | 4                   |                  |
| <b>RCL</b>         |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |
| <b>RCR</b>         |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |
| <b>ROL</b>         |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |
| <b>ROR</b>         |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |

| <i>Instruction</i> | <i>Clock Cycles</i> | <i>Transfers</i> |
|--------------------|---------------------|------------------|
| <b>SAL/SHL</b>     |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |
| <b>SAR</b>         |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |
| <b>SHR</b>         |                     |                  |
| 1-bit Reg          | 2                   |                  |
| Multi-bit Reg      | 8 + 4/bit           |                  |
| 1-bit Mem          | 15 + EA             | 2                |
| Multi-bit Mem      | 20 + EA + 4/bit     | 2                |
| <b>XOR</b>         |                     |                  |
| Reg to Reg         | 3                   |                  |
| Mem to Reg         | 9 + EA              | 1                |
| Reg to Mem         | 16 + EA             | 2                |
| Imm to Reg         | 4                   |                  |
| Imm to Mem         | 17 + EA             | 2                |
| Imm to Acc         | 4                   |                  |

### String Instructions

| <i>Instruction</i>      | <i>Clock Cycles</i> | <i>Transfers</i> |
|-------------------------|---------------------|------------------|
| <b>CMPS/CMPSB/CMPSW</b> |                     |                  |
| Not Repeated            | 22                  | 2                |
| Repeated                | 9 + 22/rep          | 2/rep            |
| <b>LDS</b>              | 16 + EA             | 2                |
| <b>LEA</b>              | 2 + EA              |                  |
| <b>LES</b>              | 16 + EA             | 2                |
| <b>LODS/LODSB/LODSW</b> |                     |                  |
| Not Repeated            | 12                  | 1                |
| Repeated                | 9 + 13/rep          | 1/rep            |
| <b>MOVS/MOVS/MOVSW</b>  |                     |                  |
| Not Repeated            | 18                  | 2                |
| Repeated                | 9 + 17/rep          | 2/rep            |
| <b>SCAS/SCASB/SCASW</b> |                     |                  |
| Not Repeated            | 15                  | 1                |
| Repeated                | 9 + 15/rep          | 1/rep            |
| <b>STOS/STOSB/STOSW</b> |                     |                  |
| Not Repeated            | 11                  | 1                |
| Repeated                | 9 + 10/rep          | 1/rep            |



## Loop and Jump Instructions

| <i>Instruction</i>        | <i>Clock Cycles</i> | <i>Transfers</i> |
|---------------------------|---------------------|------------------|
| JA/JNBE                   | 16/4                |                  |
| JAE/JNB/JNC               | 16/4                |                  |
| JB/JNAE/JC                | 16/4                |                  |
| JBE/JNA                   | 16/4                |                  |
| JCXZ                      | 18/6                |                  |
| JE/JZ                     | 16/4                |                  |
| JG/JNLE                   | 16/4                |                  |
| JGE/JNL                   | 16/4                |                  |
| JL/JNGE                   | 16/4                |                  |
| JLE/JNG                   | 16/4                |                  |
| JMP                       |                     |                  |
| Intrasegment direct short | 15                  |                  |
| Intrasegment direct       | 15                  |                  |
| Intersegment direct       | 15                  |                  |
| Intrasegment Mem-indirect | 18 + EA             | 1                |
| Intrasegment Reg-indirect | 11                  |                  |
| Intersegment indirect     | 24 + EA             | 2                |
| JNE/JNZ                   | 16/4                |                  |
| JNO                       | 16/4                |                  |
| JNP/JPO                   | 16/4                |                  |
| JNS                       | 16/4                |                  |
| JO                        | 16/4                |                  |
| JP/JPE                    | 16/4                |                  |
| JS                        | 16/4                |                  |
| LOOP                      | 17/5                |                  |
| LOOPE/LOOPZ               | 18/6                |                  |
| LOOPNE/LOOPNZ             | 19/5                |                  |

x/y – x cycles when jump is not taken.  
           y cycles when jump is taken.

## Data Transfer Instructions

| <i>Instruction</i>   | <i>Clock Cycles</i> | <i>Transfers</i> |
|----------------------|---------------------|------------------|
| IN                   |                     |                  |
| Fixed-port           | 10                  | 1                |
| Variable-port        | 8                   | 1                |
| LAHF                 | 4                   |                  |
| MOV                  |                     |                  |
| Acc to Mem           | 10                  | 1                |
| Mem to Acc           | 10                  | 1                |
| Reg to Reg           | 2                   |                  |
| Mem to Reg           | 8 + EA              | 1                |
| Reg to Mem           | 9 + EA              | 1                |
| Imm to Reg           | 4                   |                  |
| Imm to Mem           | 10 + EA             | 1                |
| Reg to SS, DS, or ES | 2                   |                  |
| Mem to SS, DS, or ES | 8 + EA              | 1                |
| Segment Reg to Reg   | 2                   |                  |
| Segment Reg to Mem   | 9 + EA              | 1                |
| OUT                  |                     |                  |
| Fixed-port           | 10                  | 1                |
| Variable-port        | 8                   | 1                |
| POP                  |                     |                  |
| Reg                  | 8                   | 1                |
| SS, DS, or ES        | 8                   | 1                |
| Mem                  | 16 + EA             | 2                |
| POPF                 | 8                   | 1                |
| PUSH                 |                     |                  |
| Reg                  | 11                  | 1                |
| Segment Reg          | 10                  | 1                |
| Mem                  | 16 + EA             | 2                |
| PUSHF                | 10                  | 1                |
| SAHF                 | 4                   |                  |
| XCHG                 |                     |                  |
| Reg with Acc         | 3                   |                  |
| Reg with Mem         | 17 + EA             | 2                |
| Reg with Reg         | 4                   |                  |
| XLAT/XLATB           | 11                  | 1                |

## Subroutine and Interrupt Instructions

| <i>Instruction</i>        | <i>Clock Cycles</i> | <i>Transfers</i> |
|---------------------------|---------------------|------------------|
| CALL                      |                     |                  |
| Intrasegment direct       | 19                  | 1                |
| Intersegment direct       | 28                  | 2                |
| Intrasegment Mem-indirect | 21 + EA             | 2                |
| Intrasegment Reg-indirect | 16                  | 1                |
| Intersegment indirect     | 37 + EA             | 4                |
| INT                       |                     |                  |
| Type-3                    | 52                  | 5                |
| Not Type-3                | 51                  | 5                |
| INTO                      |                     |                  |
| Taken                     | 53                  | 5                |
| Not Taken                 | 4                   |                  |
| INTR                      | 61                  | 7                |
| IRET                      | 24                  | 3                |
| NMI                       | 50                  | 5                |
| RET                       |                     |                  |
| Intrasegment              | 8                   | 1                |
| Intrasegment with pop     | 12                  | 1                |
| Intersegment              | 18                  | 2                |
| Intersegment with pop     | 17                  | 2                |

Note: INTR and NMI are included for timing purposes only.

## Processor Control Instructions

| <i>Instruction</i> | <i>Clock Cycles</i> | <i>Transfers</i> |
|--------------------|---------------------|------------------|
| CLC                | 2                   |                  |
| CLD                | 2                   |                  |
| CLI                | 2                   |                  |
| CMC                | 2                   |                  |
| ESC                |                     |                  |
| Reg                | 2                   |                  |
| Mem                | 8 + EA              | 1                |
| HLT                | 2                   |                  |
| LOCK               | 2                   |                  |
| NOP                | 3                   |                  |
| REP                | 2                   |                  |
| REPE/REPZ          | 2                   |                  |
| REPNE/REPNZ        | 2                   |                  |
| STC                | 2                   |                  |
| STD                | 2                   |                  |
| STI                | 2                   |                  |
| TEST               |                     |                  |
| Reg with Reg       | 3                   |                  |
| Mem with Reg       | 9 + EA              | 1                |
| Imm with Acc       | 4                   |                  |
| Imm with Reg       | 5                   |                  |
| Imm with Mem       | 11 + EA             |                  |
| WAIT               | 3 + 5n              |                  |

---

**EFFECTIVE ADDRESS (EA) CALCULATION TIMES**

| <i>Addressing Mode</i> | <i>Clock Cycles</i> |
|------------------------|---------------------|
| Displacement           | 6                   |
| Base or Index          | 5                   |
| Disp + Base or Index   | 9                   |
| Base + Index           |                     |
| BP + DI, BX + SI       | 7                   |
| BP + SI, BX + DI       | 8                   |
| Disp + Base + Index    |                     |
| BP + DI + Disp         | 11                  |
| BX + SI + Disp         | 11                  |
| BP + SI + Disp         | 12                  |
| BX + DI + Disp         | 12                  |

Notes: Add two clock cycles for segment override (e.g., MOV AL,ES:[100]). Add four clock cycles for each word transfer to/from memory.

---

## APPENDIX B

---

### A Review of Number Systems, Binary Arithmetic, and Logic Functions

---

When working with microprocessors, you will find it necessary to have a good grasp of the binary and hexadecimal number systems, and the arithmetic associated with them. This appendix is intended as a review (or a brief introduction) to this material.

---

#### DECIMAL VERSUS BINARY

A decimal number is composed of one or more digits chosen from a set of ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Each digit in a decimal number has an associated *weight* that is used to give the digit meaning. For example, the decimal number 357 contains three 100s, five 10s, and seven 1s. The weight of the digit 3 is 100, the weight of the digit 5 is 10, and the weight of the digit 7 is 1. The weights of each digit in a decimal number are related to its *base*. Decimal numbers are base-10 numbers. Thus, the weights are all multiples of 10. Look at our example decimal number again:

|        |        |        |                           |
|--------|--------|--------|---------------------------|
| 3      | 5      | 7      | — digits                  |
| $10^2$ | $10^1$ | $10^0$ | — weights as powers of 10 |
| 100    | 10     | 1      | — actual weight value     |
| <hr/>  |        |        |                           |
| 300    | 50     | 7      | — components of number    |

Notice that the weights are all powers of 10 beginning with 0. The components of the number are found by multiplying each digit value by its respective weight. The number itself is found by adding the individual components together. This technique applies to numbers in *any* base.

Now, a binary number is a number composed of digits (called **bits**) chosen from a set of only **two** digits (0, 1). Base-2 is used for binary numbers because there are only two legal digits in a binary number. This means that the weights of the bits in a binary number will

all be multiples of two! Consider the binary number 10110. The associated weights are as follows:

|       |       |       |       |       |                          |
|-------|-------|-------|-------|-------|--------------------------|
| 1     | 0     | 1     | 1     | 0     | — bits                   |
| $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | — weights as powers of 2 |
| 16    | 8     | 4     | 2     | 1     | — actual weight value    |
| <hr/> |       |       |       |       |                          |
| 16    |       | 4     | 2     |       | — components of number   |

The components are again found by multiplying each bit in the number by its respective power of 2. The individual components add up to 22. Thus, 10110 binary equals 22 decimal. We now have a technique for determining the value of any binary number.

Going from one base to another requires a *conversion*. As we just saw, going from 10110 to 22 required us to perform a **binary-to-decimal** conversion. How do we go the other way? For example, what binary number represents the decimal number 37? This requires a **decimal-to-binary** conversion. One way to do this conversion is as follows:

$$\begin{aligned}
 37 / 2 &= 18 \text{ with } 1 \text{ left over} \\
 18 / 2 &= 9 \text{ with } 0 \text{ left over} \\
 9 / 2 &= 4 \text{ with } 1 \text{ left over} \\
 4 / 2 &= 2 \text{ with } 0 \text{ left over} \\
 2 / 2 &= 1 \text{ with } 0 \text{ left over} \\
 1 / 2 &= 0 \text{ with } 1 \text{ left over}
 \end{aligned}$$

The number is repeatedly divided by 2 and the remainder recorded. When we get to  $1 / 2 = 0$  with 1 left over, we are done dividing. The binary result is found by reading the remainder bits from the bottom up. So, 37 decimal equals 100101 binary. To check, we use the binary-to-decimal conversion technique:

|       |    |   |   |   |   |
|-------|----|---|---|---|---|
| 1     | 0  | 0 | 1 | 0 | 1 |
| 32    | 16 | 8 | 4 | 2 | 1 |
| <hr/> |    |   |   |   |   |
| 32    |    |   | 4 |   | 1 |

Adding 32, 4, and 1 gives 37, the original number!

Here are some common binary and decimal numbers:

| Binary     | Decimal |
|------------|---------|
| 1010       | 10      |
| 1111       | 15      |
| 1100100    | 100     |
| 10000000   | 128     |
| 11111111   | 255     |
| 1111101000 | 1000    |

Clearly, it helps to have a good understanding of the various powers of 2 to perform conversions. The first 20 powers of 2 are as follows:

$$\begin{array}{ll}
 2^0 = 1 & 2^5 = 32 \\
 2^1 = 2 & 2^6 = 64 \\
 2^2 = 4 & 2^7 = 128 \\
 2^3 = 8 & 2^8 = 256 \\
 2^4 = 16 & 2^9 = 512
 \end{array}$$

|                  |                   |
|------------------|-------------------|
| $2^{10} = 1024$  | $2^{15} = 32768$  |
| $2^{11} = 2048$  | $2^{16} = 65536$  |
| $2^{12} = 4096$  | $2^{17} = 131072$ |
| $2^{13} = 8192$  | $2^{18} = 262144$ |
| $2^{14} = 16384$ | $2^{19} = 524288$ |

## THE HEXADECIMAL NUMBER SYSTEM

It is not easy to remember large binary numbers. For instance, examine the 20-bit binary number shown here for 5 seconds. Then close your eyes and try to repeat it:

1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 0 1 1

Were you able to do it? Most people cannot, because their short-term memory is not capable of storing so many bits of information. It is possible, however, to remember shorter sequences of characters. Try this character sequence:

A F 6 B 3

Were you able to remember it? Those who have difficulty with the 20-bit example can usually do the 5-character example easily. Here is where the trick comes in: The 20-bit binary number *is the same* as the 5-character sequence. This is because AF6B3 represents a **hexadecimal** number (base 16). Here is how the examples relate to each other:

- Separate the number into groups of 4 bits each

1 0 1 0   1 1 1 1   0 1 1 0   1 0 1 1   0 0 1 1

- Find the individual decimal equivalents of each group

1 0 1 0   1 1 1 1   0 1 1 0   1 0 1 1   0 0 1 1  
10        15        6        11        3

- Replace each 10 through 15 value with its A through F equivalent

1 0 1 0   1 1 1 1   0 1 1 0   1 0 1 1   0 0 1 1  
10        15        6        11        3  
A        F        6        B        3

This technique makes it possible to work with large binary numbers through their hexadecimal equivalents.

The word **hexadecimal** refers to “6” and “10,” or “16.” This number system contains numbers composed of digits and letters chosen from the set (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). The decimal number 10 is represented in hexadecimal as A. Letters B, C, D, E, and F are equivalent to 11 through 15, respectively. Each digit or letter in a hexadecimal number represents 4 binary bits (as we have seen). The binary patterns associated with the 16 hexadecimal symbols are as follows:

|          |          |          |          |
|----------|----------|----------|----------|
| 0 = 0000 | 4 = 0100 | 8 = 1000 | C = 1100 |
| 1 = 0001 | 5 = 0101 | 9 = 1001 | D = 1101 |
| 2 = 0010 | 6 = 0110 | A = 1010 | E = 1110 |
| 3 = 0011 | 7 = 0111 | B = 1011 | F = 1111 |

It is much more convenient (and easier to remember) to use 2 hexadecimal symbols than 8 binary bits. For instance, 3EH (the H stands for Hex) means 00111110B (B for Binary). Larger binary numbers prove this point even better (as our AF6B3H example showed). Because microprocessor address and data lines commonly utilize 8, 16, or even 32 bits of data, the 2-, 4-, or 8-symbol hexadecimal equivalents are easier to deal with.

## BINARY ADDITION

Once a decimal number has been translated into binary, what can we do with it? Usually, a number is loaded into a computer so that it can be manipulated. When two or more numbers are input, we often need to find their sum. So, it is necessary for the microprocessor to know how to add in binary. The rules for binary addition of two bits are short and straightforward:

|          |          |          |           |           |
|----------|----------|----------|-----------|-----------|
|          |          |          |           | 1         |
| 0        | 0        | 1        | 1         | + 1       |
| + 0      | + 1      | + 0      | + 1       | + 1       |
| <u>0</u> | <u>1</u> | <u>1</u> | <u>10</u> | <u>11</u> |

Because the binary number system allows only the two symbols 0 and 1, the sum of  $1 + 1$  cannot be 2! So, it is 0 with 1 to carry into the next column of bits.

Likewise, the sum of  $1 + 1 + 1$  is not 3, but 1 with 1 to carry. These are all the rules we have to remember. Let us use them to add two 8-bit numbers:

|                        |                              |
|------------------------|------------------------------|
| 1 1 1 1 1              | —carry bits                  |
| 1 0 0 1 0 1 1 0        | —first number = 150 (or 96H) |
| + 0 0 1 1 1 0 1 1      | —second number = 59 (or 3BH) |
| <u>1 1 0 1 0 0 0 1</u> | —result = 209 (or D1H)       |

This is the type of addition performed by microprocessors.

## BINARY SUBTRACTION

Oddly enough, we use binary *addition* to perform subtraction in binary. For example, subtracting these two numbers:

$$\begin{array}{r} 50 \\ - 18 \\ \hline \end{array}$$

is the same as **adding** these two numbers:

$$\begin{array}{r} 50 \\ + -18 \\ \hline \end{array}$$



So, we need a method for converting positive 18 into negative 18. We use a technique called *2's complement* to perform this conversion. To find the 2's complement of 18, we do the following:

$$\begin{array}{r}
 00010010 \text{— positive 18 in 8-bit binary} \\
 1101101 \text{— complement of each bit} \\
 + \quad \quad \quad 1 \text{— add 1} \\
 \hline
 1101110 \text{— the 2's complement of 18}
 \end{array}$$

It is not possible to distinguish a 2's complement binary number from an ordinary binary number. Two's complement refers to the way in which we *interpret* a binary number. The  $-18$  representation 1101110 must be interpreted as a **signed** binary number. A signed binary number uses its MSB as a *sign* bit, where 0 means positive and 1 means negative. So, we have:

$$\begin{array}{rcl}
 00010010 & +18, & \text{sign bit is 0} \\
 1101110 & -18, & \text{sign bit is 1}
 \end{array}$$

The value of the sign bit in these two numbers supports our interpretation.

Getting back to the original problem, we can now subtract 18 from 50 by adding the 2's complement of 18 (which is  $-18$ ) to 50:

$$\begin{array}{r}
 11111 \text{— carry bits} \\
 00110010 \text{— 50} \\
 + 1101110 \text{— } -18 \\
 \hline
 00100000 \text{— 32}
 \end{array}$$

The carry out of the MSB is ignored. Note that the MSB of the result is 0, indicating a positive result.

This brief discussion should have familiarized you with the types of numbers encountered when dealing with microprocessors. Now, when you encounter instructions like:

```

CODES      DW      14H, 3EA9H, 2200

            MOV     AL, 4CH
            AND     BL, 10110101B
  
```

you will have an idea what the numbers mean.


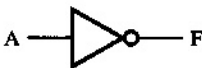






## BASIC LOGIC GATES

A microprocessor-based system requires basic logic gates and functions to interface the microprocessor with its various peripherals and memory devices. This interface logic, commonly called *glue logic*, may reside in individual integrated circuits or may be totally contained in a single PAL, GAL, FPGA, or other programmable logic device.

Table B.1 shows a summary of the basic logic gates that may be used in a microprocessor-based system.

Individual gates like these are used to combine two or more signals to determine when an operation, such as memory write, is being performed by the CPU. It is important to be familiar with the truth tables of all the basic logic gates.

**Table B.1** Basic logic gates

| Function | Symbol                                                                              | Boolean Equation                                       | Truth Table                                                                                                                                                           | TTL Package |   |    |   |    |   |      |   |    |   |       |
|----------|-------------------------------------------------------------------------------------|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|---|----|---|----|---|------|---|----|---|-------|
| Buffer   |    | $F = A$                                                | <table><tr><th>A</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>                                                                | A           | F | 0  | 0 | 1  | 1 | 7407 |   |    |   |       |
| A        | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 0        | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 1        | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| Inverter |    | $F = \bar{A}$                                          | <table><tr><th>A</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>                                                                | A           | F | 0  | 1 | 1  | 0 | 7404 |   |    |   |       |
| A        | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 0        | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 1        | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| AND      |    | $F = AB$                                               | <table><tr><th>AB</th><th>F</th></tr><tr><td>00</td><td>0</td></tr><tr><td>01</td><td>0</td></tr><tr><td>10</td><td>0</td></tr><tr><td>11</td><td>1</td></tr></table> | AB          | F | 00 | 0 | 01 | 0 | 10   | 0 | 11 | 1 | 7408  |
| AB       | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 00       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 01       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 10       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 11       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| NAND     |    | $F = \overline{AB}$                                    | <table><tr><th>AB</th><th>F</th></tr><tr><td>00</td><td>1</td></tr><tr><td>01</td><td>1</td></tr><tr><td>10</td><td>1</td></tr><tr><td>11</td><td>0</td></tr></table> | AB          | F | 00 | 1 | 01 | 1 | 10   | 1 | 11 | 0 | 7400  |
| AB       | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 00       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 01       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 10       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 11       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| OR       |    | $F = A + B$                                            | <table><tr><th>AB</th><th>F</th></tr><tr><td>00</td><td>0</td></tr><tr><td>01</td><td>1</td></tr><tr><td>10</td><td>1</td></tr><tr><td>11</td><td>1</td></tr></table> | AB          | F | 00 | 0 | 01 | 1 | 10   | 1 | 11 | 1 | 7432  |
| AB       | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 00       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 01       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 10       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 11       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| NOR      |   | $F = \overline{A + B}$                                 | <table><tr><th>AB</th><th>F</th></tr><tr><td>00</td><td>1</td></tr><tr><td>01</td><td>0</td></tr><tr><td>10</td><td>0</td></tr><tr><td>11</td><td>0</td></tr></table> | AB          | F | 00 | 1 | 01 | 0 | 10   | 0 | 11 | 0 | 7402  |
| AB       | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 00       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 01       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 10       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 11       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| XOR      |  | $F = A \oplus B$<br>$= A\bar{B} + \bar{A}B$            | <table><tr><th>AB</th><th>F</th></tr><tr><td>00</td><td>0</td></tr><tr><td>01</td><td>1</td></tr><tr><td>10</td><td>1</td></tr><tr><td>11</td><td>0</td></tr></table> | AB          | F | 00 | 0 | 01 | 1 | 10   | 1 | 11 | 0 | 7486  |
| AB       | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 00       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 01       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 10       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 11       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| XNOR     |  | $F = \overline{A \oplus B}$<br>$= AB + \bar{A}\bar{B}$ | <table><tr><th>AB</th><th>F</th></tr><tr><td>00</td><td>1</td></tr><tr><td>01</td><td>0</td></tr><tr><td>10</td><td>0</td></tr><tr><td>11</td><td>1</td></tr></table> | AB          | F | 00 | 1 | 01 | 0 | 10   | 0 | 11 | 1 | 74266 |
| AB       | F                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 00       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 01       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 10       | 0                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |
| 11       | 1                                                                                   |                                                        |                                                                                                                                                                       |             |   |    |   |    |   |      |   |    |   |       |

## COMMON LOGIC FUNCTIONS

When the basic logic gates are combined together into more complex circuits, we get the logic functions illustrated in Figure B.1.

These devices and their application to microprocessor-based systems are as follows:

- **Flip-flops:** Edge-sensitive devices used to store bits of data, make counters and shift registers, and divide a signals frequency by two or more. Groups of flip-flops (eight, for example) are used to store parallel binary data in devices called latches. Latches

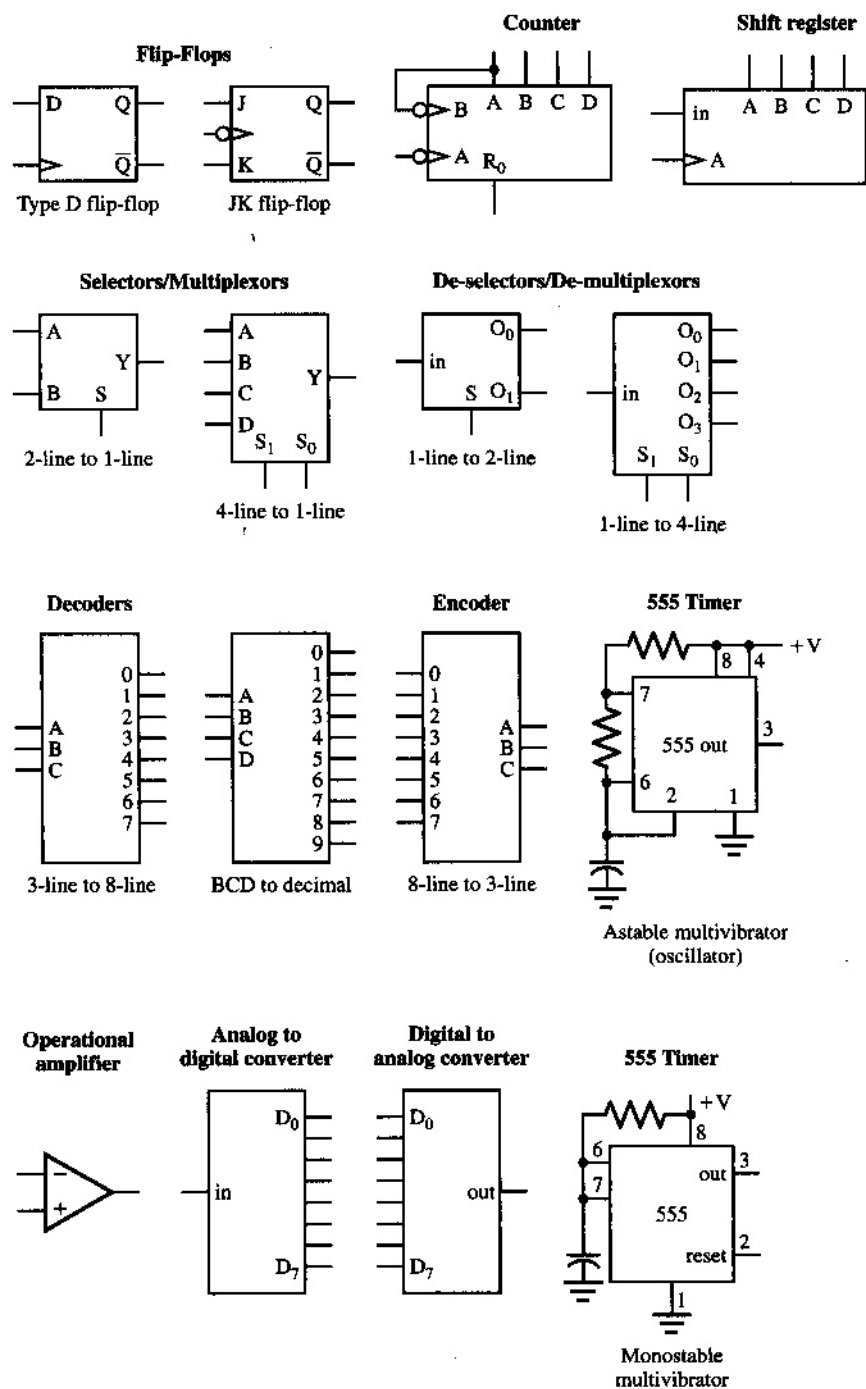


FIGURE B.1 Common logic functions

are used to make output ports and to store data temporarily. The D-type flip-flop is positive-edge triggered. The JK flip-flop is negative edge triggered.

- Counter: Counts in binary or BCD. May be used in a serial I/O circuit to divide a high-frequency clock down to a low-frequency baud rate. Also used to count clock cycles during a bus operation to identify faulty operations that take too many cycles.
- Shift register: Converts serial data to parallel and vice versa. May be used to delay a digital signal by multiple clock cycles. This may be required when interfacing a slow peripheral or memory device to the CPU.
- Selector/multiplexer: Guides one signal from many input signals to a single output. Used in dynamic RAM circuitry to select signals from one of two different address buses and in input circuitry to choose one signal from many input signals.
- De-selector/de-multiplexer: A single input signal is routed to one of many outputs. Used in output circuitry to control multiple output signals.
- Decoders: Enables a single output based on the binary or BCD number present on the inputs. Used in memory-addressing circuitry and for driving 7-segment displays.
- Encoder: Generates an output pattern based on which input is active. Used in interrupt circuitry to determine which interrupt should be serviced.
- 555 Timer: Can be used as a one-shot (mono-stable multivibrator) or as a digital oscillator (astable multivibrator, frequency up to 250 KHz). As a one-shot, the 555 timer is useful in the power-on reset circuitry. As an oscillator, the 555 timer may be used to control the sample rate of an A/D converter, or the rate displays are refreshed in a multiplexed display.
- Operational amplifier: A multipurpose amplifier, typically used to condition an analog input or output signal (change its amplitude, filter it). Used with the A/D and D/A converters.
- Analog to Digital (A/D) converter: Converts an analog voltage into a digital data value (such as an 8-bit binary number). Used to interface with real-world analog devices, such as variable resistors, temperature sensors, microphones, etc.
- Digital to Analog (D/A) converter: Converts digital data into analog voltage or current. Used to create an analog waveform (such as audio) and to generate a control voltage in an analog control system.

Being familiar with the basic logic gates and functions is good preparation for analyzing the schematic of a microprocessor-based system or as the first step when beginning the design of a new system.

---

## APPENDIX C

---

### Assembler Reference

---

Microsoft MASM Version 6.11 contains updated software capable of processing Pentium instructions. Machine codes and instruction cycle counts are generated by MASM for all instructions on each processor beginning with the 8086.

The combined assembler and linker is the program ML. To see the format of ML's command line, and the possible option switches, use the DOS command:

```
ML /?
```

or

```
ML /help
```

You will get an output that looks like this:

```
ML [ /options ] filelist [ /link linkoptions ]
```

|                                           |                                        |
|-------------------------------------------|----------------------------------------|
| /AT Enable tiny model (.COM file)         | /nologo Suppress copyright message     |
| /Bl<linker> Use alternate linker          | /Sa Maximize source listing            |
| <b>/c Assemble without linking</b>        | <b>/Sc Generate timings in listing</b> |
| /Cp Preserve case of user identifiers     | /Sf Generate first pass listing        |
| /Cu Map all identifiers to uppercase      | /Sl<width> Set line width              |
| /Cx Preserve case in publics, externs     | /Sn Suppress symbol-table listing      |
| /coff generate COFF format object file    | /Sp<length> Set page length            |
| /D<name>[=text] Define text macro         | /Ss<string> Set subtitle               |
| /EP Output preprocessed listing to stdout | /St<string> Set title                  |
| /F <hex> Set stack size (bytes)           | /Sx List false conditionals            |
| /Fe<file> Name executable                 | /Ta<file> Assemble non-.ASM file       |
| <b>/Fl[file] Generate listing</b>         | /w Same as /W0 /WX                     |
| /Fm[file] Generate map                    | /WX Treat warnings as errors           |
| /Fo<file> Name object file                | /W<number> Set warning level           |
| /FPi Generate 80x87 emulator encoding     | /X Ignore INCLUDE environment path     |
| /Fr[file] Generate limited browser info   | /Zd Add line number debug info         |
| /FR[file] Generate full browser info      | /Zf Make all symbols public            |
| /G<c d z> Use Pascal, C, or Stdcall calls | /Zi Add symbolic debug info            |
| /H<number> Set max external name length   | /Zm Enable MASM 5.10 compatibility     |
| /I<name> Add include path                 | /Zp[n] Set structure alignment         |
| /link <linker options and libraries>      | /Zs Perform syntax check only          |

The more commonly used options are highlighted in bold. For example, when a new program is being written and debugged, it is helpful to generate a list file to aid in debugging. The list file will identify any errors that show up.

To assemble the file **PROG.ASM** and create a list file, use this command line:

```
ML /F1 PROG.ASM
```

The ML program will assemble and link (if no errors are found) the **PROG.ASM** file, creating **PROG.LST**, **PROG.OBJ**, and **PROG.EXE**. It is important to use the correct case in the option switches, as ML command options are case sensitive.

To generate instruction clock counts in the list file, use this command line:

```
ML /F1 /Sc PROG.ASM
```

Clock counts are useful when determining the execution time of a program. It may be necessary to assemble a source file, but not link it. This situation may occur when a multi-source program is being written by a group of individuals, and all the modules are not available yet. Or, the program may be in the debug stage and not require an executable file yet. To assemble without linking, use this command line:

```
ML /c /F1 PROG.ASM
```

which generates list and object files but no **.EXE** file. To create an **.EXE** file, you must use the **LINK** utility provided with MASM. **LINK** has its own set of command line parameters (use **LINK /?** to see them) and is normally called automatically by ML. When the **/c** option is used, ML does not call **LINK**. To create **PROG.EXE** from **PROG.OBJ**, use this **LINK** command:

```
LINK PROG, , ;
```

which converts the contents of **PROG.OBJ** into **PROG.EXE**. To link more than one object file, use **+** signs between their names, as in:

```
LINK PA + PB + PC, , ;
```

The linker will name the **.EXE** file after the first file in the object name list.

The following is a list of MASM's reserved words used throughout the text:

|               |                               |
|---------------|-------------------------------|
| <b>ASSUME</b> | assume definition             |
| <b>BYTE</b>   | byte (as in <b>BYTE PTR</b> ) |
| <b>.CODE</b>  | begin code segment            |
| <b>.DATA</b>  | begin data segment            |
| <b>DB</b>     | define byte                   |
| <b>DD</b>     | define double-word            |
| <b>DQ</b>     | define quadword               |
| <b>DS</b>     | define storage                |
| <b>DUP</b>    | duplicate                     |
| <b>DW</b>     | define word                   |
| <b>ELSE</b>   | else statement                |
| <b>END</b>    | end program                   |
| <b>ENDIF</b>  | end if statement              |

|          |                       |
|----------|-----------------------|
| ENDM     | end macro             |
| ENDP     | end procedure         |
| ENDS     | end segment           |
| EQU      | equate                |
| .EXIT    | generate exit code    |
| EXTRN    | external reference    |
| FAR      | far reference         |
| IF       | if statement          |
| MACRO    | define macro          |
| .MODEL   | model type            |
| NEAR     | near reference        |
| OFFSET   | offset                |
| ORG      | origin                |
| PARA     | paragraph             |
| PROC     | define procedure      |
| PTR      | pointer               |
| PUBLIC   | public reference      |
| SEG      | locate segment        |
| SEGMENT  | define segment        |
| .STARTUP | generate startup code |
| WORD     | word (as in WORD PTR) |

It might be useful to design a program *template* to assist you when writing new programs. A program template consists of the minimal instructions most programs need to function, such as those that set up the segment registers and others that return to DOS. The program template shown here is a good beginning template:

```
;Program -----.ASM: <Brief description goes here>
;
;      .MODEL SMALL
;      .586
;      .DATA
;
;Put your application data here
;
;      .CODE
;      .STARTUP
;
;Put your main application code here
;
;      .EXIT
;
;Put your application procedures here
;
;      END
```

A different assembler, called TASM, may also be used. TASM (by Borland International) uses different command line options than ML does. To see a display of TASM's options, just enter TASM with no parameters. To generate list and object files use:

```
TASM /L PROG
```

TASM does not require the .ASM extension.

To create PROG.EXE you must use Borland's TLINK utility. This linker operates the same way LINK does.

With a little patience and creativity, you can also locate many 80x86 assemblers available for download over the World Wide Web. One popular 80x86 assembler is NASM (for Netwide Assembler). Documentation and installation instructions for NASM can be found at: <http://www.fifi.org/doc/nasm/html/nasmdoc0.html>

Another 32-bit assembler is MASM32, available for download at <http://www.masm32.com>. MASM32 comes with an easy-to-learn IDE.



---

## APPENDIX D

---

# DEBUG and CodeView Reference

---

The DEBUG and CodeView utilities are discussed in this appendix. DEBUG comes with DOS and allows the user instruction-level control over an .EXE or .COM program. CodeView does the same, with a fancier environment (multiple color windows) and with the addition of being able to handle newer 80386, 80486, and Pentium instructions. CodeView is supplied with Microsoft MASM.

Let us examine DEBUG first.

---

### USING DEBUG TO EXECUTE AN 80x86 PROGRAM

#### What Is DEBUG?

DEBUG is a utility program that allows a user to load an 80x86 program into memory and execute it step by step. DEBUG displays the contents of all processor registers after each instruction executes, allowing the user to determine if the code is performing the desired task. DEBUG only displays the 16-bit portion of the general purpose registers. CodeView is capable of displaying the entire 32 bits. DEBUG is a very useful debugging tool. We will use DEBUG to step through a number of simple programs, gaining familiarity with DEBUG's commands as we do so. DEBUG contains commands that can display and modify memory, assemble instructions, disassemble code already placed into memory, trace through single or multiple instructions, load registers with data, and do much more.

DEBUG loads into memory like any other program, in the first available slot. The memory space used by DEBUG for the user program begins *after* the end of DEBUG's code. If an .EXE or .COM file were specified, DEBUG would load the program according to accepted DOS conventions.

## Getting Started

The best way to get familiar with DEBUG is to work through some examples with it. The first example we will use is this three-instruction sequence:

```
MOV  AL, 7
MOV  BH, 2
ADD  AL, BH
```

The first instruction places the number 7 into register AL. The second instruction places 2 into register BH. These two registers are added together in the third instruction, with the results going into AL. With DEBUG, we will be able to type in the instructions as they appear. DEBUG will automatically assemble them and place their respective codes into memory. We will then be able to examine the results of each instruction by tracing through the instructions one at a time.

The first step is to invoke DEBUG. This is done with a simple command, printed here in boldface. Make sure your floppy or hard disk has a copy of DEBUG.COM installed on it, and that it is in your current directory. At the DOS command prompt, enter:

```
C> debug <cr>
```

The expression <cr> indicates that you should hit the return key. Because DEBUG is a .COM file, DOS will load it into memory and execute it. DEBUG uses a minus sign as its command prompt, so you should see a “-” appear on your display.

To get a list of all commands available with DEBUG, enter a question mark at DEBUG’s command prompt and press <cr>. The command summary appears like this:

```
-?<cr>
assemble      A [address]
compare       C range address
dump          D [range]
enter         E address [list]
fill          F range list
go            G [=address] [addresses]
hex           H value1 value2
input         I port
load          L [address] [drive] [firstsector] [number]
move          M range address
name          N [pathname] [arglist]
output        O port byte
proceed       P [=address] [number]
quit          Q
register       R [register]
search        S range list
trace         T [=address] [value]
unassemble    U [range]
write         W [address] [drive] [firstsector] [number]
allocate expanded memory  XA [#pages]
deallocate expanded memory XD [handle]
map expanded memory pages XM [Lpage] [Ppage] [handle]
display expanded memory status XS
-
```

We will begin with a small group of these commands to get the feel for how DEBUG is used.

To enter the three instructions we wish to execute, we need to use the *assemble* command. Entering the command letter "a" at the prompt should result in a display similar to this:

```
-a<cr>
1539:0100
```

This is the familiar CS:IP format used throughout the text. Instead of using the actual effective address, DEBUG shows us the CS value and the IP value. The 1539 address will most likely be different on your machine, because it is probably not configured like the one used to generate the DEBUG examples. The second address, 0100, should be the same. This is DEBUG telling us that it will place the first instruction into the code segment at offset 0100H. Bear in mind that DEBUG interprets *all* numbers as hexadecimal numbers.

When DEBUG begins execution, it sets the value of each processor register to a default value. The segment registers (CS, DS, ES, and SS) are all set to the beginning of free memory. This accounts for the 1539 address in the CS register at the beginning of the **a** command. The instruction pointer is always set to address 0100 and all other processor registers are set to 0000 with the exception of the stack pointer (SP), which is set to address FFEE. In addition, all processor *flags* are cleared. It is useful to know how the processor registers are initialized. This will become clear as we proceed through the example.

At this point you can enter the three instructions. If you make a typing mistake, use the backspace key to correct your errors before hitting return. You should see something similar to this on your display:

```
-a<cr>
1539:0100 mov al,7<cr>
1539:0102 mov bh,2<cr>
1539:0104 add al,bh<cr>
1539:0106 <cr>
-
```

Notice that the IP address changes after each instruction. The fourth instruction, if there were one, would begin at address 0106. Because we are entering only three instructions, hitting a return on the fourth line will terminate the assemble command and get us back to the command prompt.

To examine the code that was generated by each instruction, we use the *unassemble* command. Unassemble will show us the addresses, opcodes, and instruction mnemonics for the three instructions we have just entered. If *unassemble* is used without parameters it will show the next 20H bytes and their corresponding 80x86 instruction equivalents. We can shorten this display by using a range parameter, like this:

```
-u 100 104<cr>
```

This command tells DEBUG to unassemble the bytes between addresses 0100 and 0104. The resulting display looks like this:

```
1539:0100 B007    MOV  AL,07
1539:0102 B702    MOV  BH,02
1539:0104 00F8    ADD  AL,BH
```

Here, we can see that each instruction entered the required 2 bytes of machine code.

To begin execution we should examine the contents of each register. Then we will know which registers change as we step through the program. DEBUG's *register* command

**TABLE D.1** Flag codes

| <i>Flag</i> | <i>Set</i> | <i>Clear</i> |
|-------------|------------|--------------|
| Overflow    | OV         | NV           |
| Direction   | DN         | UP           |
| Interrupt   | EI         | DI           |
| Sign        | NG         | PL           |
| Zero        | ZR         | NZ           |
| Aux. Carry  | AC         | NA           |
| Parity      | PE         | PO           |
| Carry       | CY         | NC           |

can be used to display (and modify) any of the processor's registers. To display their contents, simply enter "t" and return. You should get a display similar to this:

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0100 NV UP EI PL NZ NA PO NC
1539:0100 B007          MOV AL,07
```

Spend a few moments looking at the value displayed for each register. Note that all general purpose registers have been loaded with 0000. Also, the CS, DS, SS, and ES registers have all been initialized to the 1539 address we have seen earlier. IP points to address 0100, where we placed the first instruction. The end of the second line indicates the state of the flags. Table D.1 explains the meaning of each flag code. We can see that currently there is no carry, odd parity, no auxiliary carry, not zero, and plus indicated, along with enabled interrupts, up direction (for use with string operations), and no overflow. It is sometimes important to watch the changes in flags as we step through a program.

The last line of the display shows the instruction that will be executed next. Because we have not executed any instructions yet, this is our first instruction! To execute a single instruction we use DEBUG's *trace* command, abbreviated "t," and get the following display:

```
-t <cr>
AX=0007 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0102 NV UP EI PL NZ NA PO NC
1539:0102 B702          MOV BH,02
```

By comparing this display with the previous one, we can determine that only the lower byte of AX has been changed (it now contains 07H). No other registers except IP have been affected. No condition codes have been changed. Isn't that what MOV AL,7 should do?

Tracing through the next two instructions should look something like this:

```
-t <cr>
AX=0007 BX=0200 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0104 NV UP EI PL NZ NA PO NC
1539:0104 00F8          ADD AL,BH

-t <cr>
AX=0009 BX=0200 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0106 NV UP EI PL NZ NA PE NC
1539:0106 8B0EDF47      MOV CX,[47DF]
```

As expected, the final value in AL is 09H (the sum of 7 and 2). The flag display indicates that the result of the ADD instruction changed the parity flag. All other flags kept their states.

As a point of interest, look at the instruction located at address 0106. Where did it come from? We did not enter this code during any point of our exercise. Nonetheless, DEBUG thinks this is the next instruction to be executed. The reason for this is that each of the PC's 640KB comes up in a random pattern when power is first applied. DEBUG will try to interpret these random patterns as valid 80x86 instructions, as it is doing with the MOV CX instruction.

To return to DOS, exit DEBUG by entering "q" (for *quit*) at the command prompt.

You, or your instructor, may find it necessary to save all of the work you perform while using DEBUG. This is easily accomplished by pressing the **Control-PrintScreen** button on the keyboard. This will cause all characters entered from the keyboard, and all characters output to the screen, to be *echoed* to the printer, which must be turned on and loaded with paper. This will produce a hard copy of your work and will allow you to scan the results of each instruction by watching how the registers change.

To stop echoing text to your printer, press the **Control-PrintScreen** button again. You may wish to use this feature of DOS as you work through the next few examples.

---

■ **EXAMPLE D.1:** What is the final value in AL after this sequence of instructions executes?

```
MOV AL, 27H
MOV BL, 37H
ADD AL, BL
DAA
```

Use DEBUG to enter these four instructions and trace their execution. What are the machine codes for each instruction? What is the final value in AL? What changes must be made when entering the instructions with the assemble command?

**Solution:** The purpose of this example was to use the properties of the DAA instruction. When we add 27 to 37 we expect to get 64, the correct *decimal* answer. Using DEBUG to trace through the ADD instruction shows that AL contains 5EH. The next trace command executes the DAA instruction, which corrects the value in AL to 64H, the correct packed-decimal result. You should also see that the auxiliary carry flag has been set as a result of the DAA instruction.

The machine codes for each instruction are as follows:

```
B027 MOV AL, 27
B337 MOV BL, 37
00D8 ADD AL, BL
27 DAA
```

The machine codes can be obtained in two ways. During a trace, the machine codes and mnemonics for each instruction are displayed after the register list. The unassemble command can also be used. Try u 100 106 for this example and see what you get.

Also, notice that the "H" was left off the 27 and 37 operands, because DEBUG expects all numbers to be in hexadecimal form. If you include the "H" by accident, DEBUG will display an error message and wait for you to reenter the instruction. ■

---

- **EXAMPLE D.2:** The following sequence of DEBUG instructions converts an integer stored in AX from a miles-per-hour (mph) value into a feet-per-second (fps) value. For example, 60 mph equals 88 fps. The conversion is accomplished by first multiplying AX by 5280 (14A0H) and then dividing the result by 3600 (0E10H).

```
MOV  BX,14A0
MUL  BX
MOV  BX,E10
DIV  BX
```

Add the necessary instructions to convert 60 mph into 88 fps. Show that the results are correct at each step in the conversion.

**Solution:** Because DEBUG works with hexadecimal numbers only, we must first convert 60 into hex. This gives us 3CH. Register AX must be loaded with this value prior to execution. The resulting code is as follows:

```
MOV  AX,3C
MOV  BX,14A0
MUL  BX
MOV  BX,E10
DIV  BX
```

When the instructions are entered with the assemble command and executed with trace, the contents of registers AX, BX, and DX are as follows:

```
MOV  AX,3C      ->  AX=003C  BX=0000  DX=0000
MOV  BX,14A0    ->  AX=003C  BX=14A0  DX=0000
MUL  BX         ->  AX=D580  BX=14A0  DX=0004
MOV  BX,E10     ->  AX=D580  BX=0E10  DX=0004
DIV  BX         ->  AX=0058  BX=0E10  DX=0000
```

Notice that after the MUL BX instruction, AX contains D580H and DX contains 0004H. Remember that when a 16-bit register is used in MUL, the result of the multiplication is 32 bits wide and is saved in registers DX and AX (with DX holding the upper 16 bits). Thus, the product is 4D580H, which is 316,800 decimal. Check for yourself that 316,800 equals 60 times 5,280.

After the DIV BX, the final result in AX is 0058H. When converted into decimal, we get 88. ■

- **EXAMPLE D.3:** Give a sequence of instructions that will add up all integers from 1 to 10. The sum should be found in register BL.

**Solution:** One way to find the required sum is to add each integer from 1 to 10 to BL one at a time, like this:

```
SUB  BL,BL      ;set result to zero
ADD  BL,1       ;add 1 to BL
ADD  BL,2       ;add 2 to BL
.
.
.
```

```
ADD    BL,9      ;add 9 to BL
ADD    BL,0A     ;add 10 to BL
```

Perhaps a better solution can be found by using a *loop*. A loop is a technique used to repeat a group of instructions any number of times. These instructions are called *loop instructions* in general, and in this example the loop instructions need to be executed ten times. Examine the following DEBUG session:

```
-a<cr>
19DC:0100 SUB BL,BL<cr>
19DC:0102 MOV AL,0A<cr>
19DC:0104 ADD BL,AL<cr>
19DC:0106 DEC AL<cr>
19DC:0108 JNZ 104<cr>
19DC:010A <cr>
```

Register AL is used as the *loop counter*. The loop counter is decremented once each time the loop is executed. The JNZ instruction causes the processor to jump back to address 104 until AL gets to 0. A long sequence of trace commands will show how the sum builds in BL as AL keeps getting smaller. To keep from having to enter the trace commands so many times, you might try:

```
-t 1e<cr>
```

This will cause DEBUG to trace 1EH (or 30 decimal) instructions. The last two traces should be similar to these:

```
AX=0001 BX=0037 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=19DC ES=19DC SS=19DC CS=19DC IP=0106 NV UP EI PL NZ NA PO NC
19DC:0106 FEC8          DEC     AL
-t
```

```
AX=0000 BX=0037 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=19DC ES=19DC SS=19DC CS=19DC IP=0108 NV UP EI PL ZR NA PE NC
19DC:0108 75FA          JNZ     0104
-
```

Notice how the zero flag changes (from **NZ** to **ZR**) when AX goes from 0001 to 0000. Because the JNZ command watches the state of the zero flag, execution will now continue at address 10A. The final result in register BL is 37H, which is 55 decimal, the correct sum of all the integers between 1 and 10. ■

## Calling DOS Functions from DEBUG

Now that you have a little experience using DEBUG to execute simple sequences of instructions, we can move on to more complicated applications. We will make use of three new DEBUG commands: *enter*, *dump*, and *proceed*. We will also use a DOS function call through INT 21H. This is a very versatile DOS function, capable of performing many different operations. For our first example, we will use the display-string function of INT 21H. This function is selected by placing 9 into register AH, the register used by INT 21H to determine which of its many functions have been selected. Display-string requires that the address of the first byte of the text string be placed into DS:DX before using INT 21H. This means that the string must be contained in the data segment pointed to by

DS and have an offset equal to the value in DX. Use the assemble command of DEBUG to enter these instructions:

```
MOV  AH,9
MOV  DX,200
INT  21
```

Notice that we do not initialize the DS register. Remember that DEBUG automatically sets DS, CS, ES, and SS to the same value. This guarantees that our text string will be placed into the current data segment area. The offset value of 200H used to initialize DX is not a special value; it just happened to be a round number. Because the machine codes for the instructions are being placed into memory around address 100H, 200H seemed a good place to put the text string.

We can enter the text string two ways. First, we will make use of a new DEBUG command called *enter*. Enter allows memory to be modified on a byte-by-byte basis, beginning at the address specified in the instruction. To load the text string "Hello!\$" into memory at address 200H enter **e 200** and each individual ASCII byte for every character in the text string. You will get a display similar to this:

```
-e200 <cr>
1539:0200  66.48  6F.65  75.6C  6E.6C  64.6F  0D.21  0A.24  00.<cr>
```

The first pair of numbers are the actual address where the string is being loaded. The next number (66H) is the byte stored at location 200H. DEBUG follows it with a period and waits for you to enter the new byte value. The new value of 48H is entered, followed by the spacebar. Hitting the spacebar without entering any new value will skip over the location without changing its value. The display shows that 7 new bytes were entered. The last byte displayed (00) is not followed by anything because Return was hit to terminate the enter command. The 7 bytes entered are the ASCII values for the characters in the "Hello!\$" string. The "\$" character must be at the end of a text string for display-string to know where the string ends.

We can check our work with another new command: *dump*. Dump displays the bytes stored in a range of memory locations. To verify that the text string has been properly stored, do the following:

```
-d200 20f <cr>
1539:0200  48 65 6C 6C 6F 21 24 00-F6 38 53 79 6E 74 61 78 Hello!$. .8Syntax
```

The dump command displays the starting address, followed by 16 bytes of data read from memory. The final part of each line of a dump display is the ASCII-equivalent characters for each of the 16 bytes. The dump display clearly shows that we entered the string correctly.

A second technique that can be used to enter strings uses the assemble command. To place a different string at address 400H, do this:

```
-a400<cr>
1539:0400  db 'Try this string too...$' <cr>
1539:0417  <cr>
```

The "db" directive stands for *define-byte*, and causes DEBUG to look up the ASCII values of any characters surrounded by single quotes. You could easily enter numeric values with db as well. If you count all of the characters in the new string, including the blanks, you should get 23. This indicates that locations 400H through 416H will be loaded with the corresponding ASCII byte values. The next possible location to put anything in is 417H, which DEBUG is already indicating.



Getting back to the example at hand, we have placed a text string into memory at address 200H and entered the instructions necessary to INT 21H's display-string function. Use the trace command until it gets to the INT 21 instruction. You should see that AH contains 09 and DX contains 0200. Unfortunately, there may be hundreds of instructions involved in the execution of INT 21H. It would be a waste of time to trace through every one of them. It would be nice if INT 21 could be treated as a single instruction by DEBUG, with all instructions of INT 21, including the final RETURN, executing by entering a single DEBUG command. Fortunately for us, DEBUG does have such a command: *proceed!* Proceed causes all INT, CALL, and REP instructions to be treated as single instructions. So, if a subroutine contains forty-five instructions, using the proceed command when the CALL instruction shows up in the trace will cause DEBUG to execute all forty-five instructions, and show the contents of each register *upon return from the subroutine!* We can use proceed to see what happens when we call INT 21. Your last DEBUG trace should look something like this:

```
AX=0900 BX=0000 CX=0000 DX=0200 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0105 NV UP EI PL NZ NA PO NC
1539:0105 CD21          INT     21
```

If the proceed command is now used, the resulting display becomes:

```
-p<cr>
Hello!
AX=0924 BX=0000 CX=0000 DX=0200 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0107 NV UP EI PL NZ NA PO NC
1539:0107 6C          DB      6C
```

You can see that the "Hello" string appeared on the screen in the current cursor location, and that DEBUG will get its next instruction from 1539:0107. Obviously, INT 21H must have done its job, or the string would not have appeared. The proceed command is very useful for tracing programs that involve DOS function calls.

---

■ **EXAMPLE D.4:** What must be done to display the second string, which was entered with the assemble command and the DB directive?

**Solution:** Because the string was placed at address 400H, the MOV DX,200 instruction must be changed to MOV DX,400. Then the entire sequence of instructions is executed again. DEBUG updates the IP register after each instruction executes, so it will be necessary to load IP with 100 again (if the first instruction is at 100). The register command can be used to do this. The following steps will set IP back to 100:

```
-r ip<cr>
IP 0107
:100<cr>
```

The instructions for the second string can now be traced as those for the first were, with the proceed command used when INT 21 shows up again. ■

---

We will finish this section with one final example using two more DOS function calls. INT 21H can also read the computer's time and date if the appropriate value is

placed into AH. To read the time, AH must contain 2CH. To read the date, AH must contain 2AH. The time and date are returned in various registers, as Example D.5 shows.

---

■ **EXAMPLE D.5:** INT 21H requires no register setup before it is called when we are reading only the time or date. The time is returned in the following way: CH contains hours, CL contains minutes, and DH seconds. Hundredths of seconds are returned in DL. The date is returned with AL containing the day of the week, CX the year (1980 to 2099 only), DH the month, and DL the day. Can you determine the time and date from these DEBUG trace displays?

Time trace:

```
AX=2C00 BX=0000 CX=0F1C DX=0235 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0104 NV UP EI PL NZ NA PO NC
```

Date trace:

```
AX=2A02 BX=0000 CX=07CD DX=0417 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1539 ES=1539 SS=1539 CS=1539 IP=0108 NV UP EI PL NZ NA PO NC
```

**Solution:** The values contained in CX and DX in the time trace indicate that the computer's time was 15:28:02 when INT 21H was called. The values in CX and DX in the date trace show the date to be 4/23/97. The day of the week stored in AL is 2, corresponding to Tuesday. Sunday is indicated by 0. ■

---



---

## RUNNING .EXE AND .COM PROGRAMS WITH DEBUG

In this section we will examine one more command: *go*. DEBUG can automatically load an .EXE or .COM file into memory, performing all necessary relocation and initialization. The great advantage here is that we do not have to enter the program by hand. We can also use DEBUG to examine existing programs, executing portions of them to determine how they work. Microcode exploration with DEBUG can be a tremendous learning experience.

To load a program with DEBUG, include the name of the program in the command line. For example:

```
C> debug hello.exe<cr>
```

will cause DEBUG to load the HELLO.EXE code into memory. Once done, we can dump, unassemble, trace, or modify the code as we see fit. If we need only to execute the program, we issue the *go* command, and see the following display:

```
-g<cr>
Hello!
Program terminated normally
-
```

A nice advantage of using DEBUG to execute a newly developed program in this manner is that the program exits to DEBUG, not DOS, when completed. If we desire, we can now examine the stack area to see what values were pushed onto the stack. Or, if the program performed calculations and saved the results in the data segment, we can use *dump* to examine the results and verify their correctness. An example of this technique involves the

data summing program FINDAVE. Ten numbers are added and averaged. The results are then saved in the data segment (via the SUM and AVERAGE variables). The list file is included here to give an indication of what the program looks like and some idea as to where the results can be found.

```

;Program FINDAVE.ASM: Find the average of ten words.
;
.MODEL SMALL
.DATA
0000
0000 0A          COUNT    DB    10
0001 0000          SUM      DW    ?
0003 0000          AVERAGE DW    ?
0005 0064 00C8 012C 0190  VALUES DW    100,200,300,400,500
01F4
000F 03E8 07D0 0BB8          DW    1000,2000,3000,4000,5000
0FA0 1388

0000          .CODE
          .STARTUP
0017 8D 36 0005 R      LEA     SI,VALUES ;set up pointer to data
001B B8 0000          MOV     AX,0 ;set initial sum to zero
001E 8A 0E 0000 R      MOV     CL,COUNT ;set up loop counter
0022 03 04          ADDLOOP: ADD     AX,[SI] ;add new data item to sum
0024 83 C6 02          ADD     SI,2 ;point to next data item
0027 FE C9          DEC     CL ;decrement loop counter
0029 75 F7          JNZ     ADDLOOP ;jump if CL is not zero
002B A3 0001 R      MOV     SUM,AX ;save sum
002E 99          CWD     ;convert sum to double-word
002F BB 000A          MOV     BX,10 ;prepare for division by ten
0032 F7 FB          IDIV    BX ;divide to find average
0034 A3 0003 R      MOV     AVERAGE,AX ;save average
          .EXIT

END

```

By examining the list file, we can see that the sum will be stored at address 0001 and the average at address 0003 within the data segment. After execution by DEBUG, we can dump these locations to view the results. We must first use the unassemble command to find out where DEBUG located the data segment.

```

C> debug findave.exe<cr>
-g<cr>

```

```

Program terminated normally
-u<cr>

```

```

2B60:0000 BA632B      MOV     DX,2B63
2B60:0003 8EDA        MOV     DS,DX
2B60:0005 8CD3        MOV     BX,SS
2B60:0007 2BDA        SUB     BX,DX
2B60:0009 D1E3        SHL     BX,1
2B60:000B D1E3        SHL     BX,1
2B60:000D D1E3        SHL     BX,1
2B60:000F D1E3        SHL     BX,1
2B60:0011 FA        CLI
2B60:0012 8ED2        MOV     SS,DX

```

```

2B60:0014 03E3      ADD     SP,BX
2B60:0016 FB        STI
2B60:0017 8D361100  LEA     SI,[0011]
2B60:001B B80000     MOV     AX,0000
2B60:001E 8A0E0C00  MOV     CL,[000C]

```

The unassembled code shows that the data segment begins at 2B63:0000. However, the address of `VALUES` loaded into `SI` is no longer 0005 as shown in the list file, but 0011. The linker relocated the entire data segment when it created `FINDAVE.EXE` from `FINDAVE.OBJ`. Further use of the `u` command will show that `SUM` and `AVERAGE` have the new offset addresses 000D and 000F, respectively. Now we can use the `dump` command to examine the results.

```

-d 2B63:0 1f
2B63:0000 0A 00 F7 FB A3 0F 00 B4-4C CD 21 00 0A 74 40 72 .....L!...t@r
2B63:0010 06 64 00 C8 00 2C 01 90-01 F4 01 E8 03 D0 07 B8 .d.....
-q

```

The sum stored at address 000D is 4074H. This equates to 16,500, the correct sum for the ten data items included in the program. The average stored at address 000F is 0672H, which is the proper average of 1650. Thus, we see that `DEBUG` can be used to verify the operation of an `.EXE` file and let us know if the program is working properly.

## Using Script Files with DEBUG

There are times when a new programming exercise presents a significant challenge and requires many `DEBUG` sessions to finally get it right. It is not difficult to imagine that typing in the same group of instructions over and over quickly leads to frustration, even when the last attempt provides the correct solution. One way to avoid having to duplicate the same work in a series of `DEBUG` sessions while a programming exercise is being developed is to use a *script* file. A script file contains all of the `DEBUG` commands and statements you might enter during a session and is created using an ordinary text editor. For example, consider the following script file:

```

a
mov al,5
mov cl,6
mul cl
sub al,12
mov bl,al

r

t 5

q

```

This script file contains ten lines. The first line contains `DEBUG`'s `a` command. This will put `DEBUG` into assembly mode. The next five lines are the instructions we wish to assemble and place into memory. Line seven is a blank line and is important because it gets `DEBUG` out of assembly mode. Line eight contains `DEBUG`'s `r` command. This will display the current register contents prior to execution. Line nine uses `DEBUG`'s trace command to show the execution results of the five instructions entered earlier. Finally, line ten allows us to quit `DEBUG` from within the script file.

To use the script file, create it with a text editor and save it as **FILENAME.SCR**. Then, use the following command to allow **DEBUG** to access the script file:

```
DEBUG < FILENAME.SCR
```

The **<** symbol is a DOS function that allows the standard input device (the keyboard in this case) to be *redirected*. This means that instead of **DEBUG** waiting for a keystroke for each new command or statement, it will simply read it from the script file. So to **DEBUG**, using a script file is not any different from simply typing in all the statements really fast.

Once again, to get a printout of your **DEBUG** session, use the **Control-PrintScreen** function to toggle the printer prior to using the script file.

The script file previously discussed is used to solve the following equation:

$$BL = (6 \times AL) - 18$$

Note that the 18 in the equation is a 12 in instruction line five, because **DEBUG** uses only hexadecimal numbers.

In the script file, **AL** is initially loaded with the number 5. The equation predicts **BL** to have a value of 12. Examine the following **DEBUG** session (which was generated by the script file) to verify that the given instructions performed the calculation correctly.

```
-a
1CE2:0100 mov al,5
1CE2:0102 mov cl,6
1CE2:0104 mul cl
1CE2:0106 sub al,12
1CE2:0108 mov bl,al
1CE2:010A

-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1CE2 ES=1CE2 SS=1CE2 CS=1CE2 IP=0100 NV UP EI PL NZ NA PO NC
1CE2:0100 B005          MOV     AL,05
-t 5

AX=0005 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1CE2 ES=1CE2 SS=1CE2 CS=1CE2 IP=0102 NV UP EI PL NZ NA PO NC
1CE2:0102 B106          MOV     CL,06

AX=0005 BX=0000 CX=0006 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1CE2 ES=1CE2 SS=1CE2 CS=1CE2 IP=0104 NV UP EI PL NZ NA PO NC
1CE2:0104 F6E1          MUL     CL

AX=001E BX=0000 CX=0006 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1CE2 ES=1CE2 SS=1CE2 CS=1CE2 IP=0106 NV UP EI PL NZ NA PO NC
1CE2:0106 2C12          SUB     AL,12

AX=000C BX=0000 CX=0006 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1CE2 ES=1CE2 SS=1CE2 CS=1CE2 IP=0108 NV UP EI PL NZ NA PE NC
1CE2:0108 88C3          MOV     BL,AL

AX=000C BX=000C CX=0006 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=1CE2 ES=1CE2 SS=1CE2 CS=1CE2 IP=010A NV UP EI PL NZ NA PE NC
1CE2:010A 48           DEC     AX
-q
```

The final value in register **BL** is **0CH**, which is the correct result.

---

## A BRIEF LOOK AT CODEVIEW

CodeView is a fancier version of DEBUG that allows user control over the execution of a program being developed or examined. CodeView displays its information in different windows, such as the source window, register window, command window, etc. To switch from one window to another just press the F6 button.

The source window can display source code in the original form of the source file or in machine language and mnemonics. Pressing F3 switches the display format.

The register window displays the hexadecimal contents of all processor registers, both 16- and 32-bit. The state of each arithmetic flag is also displayed, along with the address and data of the last data access. When a program is traced instruction by instruction, the contents of any registers that change during execution are highlighted. This makes it easy to follow the progress of the program. The contents of any window can be sent to the printer or to a file (called CODEVIEW.LST by default). The register window looks like this when printed:

```
AX = 0000
BX = 0000
CX = 0000
DX = 0000
SP = 0000
BP = 0000
SI = 0000
DI = 0000
DS = 2B71
ES = 2B71
SS = 2B81
CS = 2B81
IP = 0010
FL = 0200
```

```
NV UP EI PL
NZ NA PO NC
```

A command window is provided to enable the user to enter debugging commands, in a fashion similar to that of DEBUG. Some of the more useful commands are as follows:

|    |                    |
|----|--------------------|
| A  | Assemble           |
| BC | Breakpoint Clear   |
| BD | Breakpoint Disable |
| BE | Breakpoint Enable  |
| BL | Breakpoint List    |
| BP | Breakpoint Set     |
| E  | Animate            |
| G  | Go                 |
| H  | Help               |
| I  | Port Input         |
| K  | Stack Trace        |
| L  | Restart            |

|    |                 |
|----|-----------------|
| MC | Memory Compare  |
| MD | Memory Dump     |
| ME | Memory Enter    |
| MF | Memory Fill     |
| MM | Memory Move     |
| MS | Memory Search   |
| N  | Radix           |
| O  | Options         |
| O  | Port Output     |
| P  | Program Step    |
| Q  | Quit            |
| R  | Register        |
| T  | Trace           |
| U  | Unassemble      |
| VM | View Memory     |
| X  | Examine Symbols |

Help on every command is available online. For example, the command:

```
H VM
```

displays the help information that describes the VM command.

There are other windows that allow the user to watch the contents of a variable change as the program executes and see a display of the contents of a block of memory.

The assembler places special symbolic debugging information into an .EXE file when it is assembled with the /Zi option, as in:

```
ML /Zi FINDAVE.ASM
```

This symbolic information helps CodeView keep track of things at the source-file level. You must specify the name of an .EXE file when starting CodeView (as a command line parameter):

```
CV FINDAVE
```

CodeView does not require the .EXE extension.

CodeView is also capable of handling not just the 32-bit registers (EAX, EBX, etc.) but the newer 80386 and 80486 instructions as well. Recall that DEBUG is only capable of displaying the 16-bit register sizes, and cannot assemble or unassemble instructions other than those of the 8086.

CodeView allows data operands to be displayed in their intended format. For example, a 4-byte integer and a 4-byte real number must be interpreted and displayed differently. This is easily accomplished with the MD (or VM) command. A format specifier is used to control the data format. These specifiers are:

|   |                   |
|---|-------------------|
| A | ASCII characters  |
| B | Byte              |
| C | Code              |
| I | Integer (2 bytes) |

|    |                                   |
|----|-----------------------------------|
| IU | Integer unsigned (2 bytes)        |
| IX | Integer hexadecimal (2 bytes)     |
| L  | Long (4-byte decimal)             |
| LU | Long unsigned (4 byte)            |
| LX | Long hexadecimal (4 bytes)        |
| R  | Real (4-byte floating point)      |
| RL | Real long (8-byte floating point) |
| RT | Real 10-byte floating point       |

Altogether, CodeView offers a significant improvement over the debugging capabilities of DEBUG. It is worth the time invested in learning how to use all of CodeView's features.



---

## APPENDIX E

---

### ASCII Character Set

---

The code that is used for the transfer of information between computers and their peripheral devices is the American Standard Code for Information Exchange, commonly referred to as ASCII code. This code is shown in Table E.1. Below is a summary of the ASCII chart.

The portion of the ASCII code in the range 20 to 7E represents readable characters corresponding to the Roman alphabet in both upper and lower case, the numbers from 0 to 9, special symbols such as (, ), \$, &, as well as punctuation marks. Thus a 50 sent to a computer monitor or a printer will cause a letter *P* to appear. If the character *w* is typed on a keyboard, then the code 77 is sent to the computer. Some symbols in the ASCII table worth noting are the capital letters, which begin with 41 representing the letter *A*, and ascending in proper order. The lowercase letters begin with 61 representing the letter *a*. The lowercase and uppercase letters differ from each other by 20. It is also noteworthy that 20 is the code for a space.

The ASCII codes in the range 0 to 1F represent control characters. As an example, they are the codes sent to a computer by the keyboard when a letter key is struck while the CTRL key is held down. To determine the HEX code for this operation, simply subtract 40 from the ASCII value of the capital letter. Thus CTRL-I produces HT, a horizontal tab. A CTRL-H sends the code for a backspace, whereas CTRL-L produces a form feed which can be used to clear a screen or to have a printer eject a sheet of paper.

Typing CTRL-S produces a DC3, which is also known as an XOFF. It can be used in most programs to stop screen scrolling. The complementary action is obtained by typing CTRL-Q, which produces a DC3, also known as an XON. This can be used in most programs to restart screen scrolling. The ACK is often used in serial MODEM communication systems to acknowledge receipt of a block of valid data, whereas NAK can be used as a negative acknowledgement. And when the computer sends to the terminal a CTRL-G, which is a BEL, we used to get a ring but now get a beep. We should not omit that a carriage return (CR) is a 0D and is used to bring the screen cursor to the beginning of the current line. This is the control code generated when you press "Enter" on the keyboard. In addition, a line feed (LF) is a 0A, used to bring the screen cursor to the next line. Both are needed at the end of a line to make an orderly transition from one line to the next.

TABLE E.1 The ASCII character set

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------|-----|-----|------|-----|-----|------|-----|-----|------|
| 0   | 00  | NUL  | 32  | 20  | SP   | 64  | 40  | @    | 96  | 60  | `    |
| 1   | 01  | SOH  | 33  | 21  | !    | 65  | 41  | A    | 97  | 61  | a    |
| 2   | 02  | STX  | 34  | 22  | "    | 66  | 42  | B    | 98  | 62  | b    |
| 3   | 03  | ETX  | 35  | 23  | #    | 67  | 43  | C    | 99  | 63  | c    |
| 4   | 04  | EOT  | 36  | 24  | \$   | 68  | 44  | D    | 100 | 64  | d    |
| 5   | 05  | ENQ  | 37  | 25  | %    | 69  | 45  | E    | 101 | 65  | e    |
| 6   | 06  | ACK  | 38  | 26  | &    | 70  | 46  | F    | 102 | 66  | f    |
| 7   | 07  | BEL  | 39  | 27  | '    | 71  | 47  | G    | 103 | 67  | g    |
| 8   | 08  | BS   | 40  | 28  | (    | 72  | 48  | H    | 104 | 68  | h    |
| 9   | 09  | HT   | 41  | 29  | )    | 73  | 49  | I    | 105 | 69  | i    |
| 10  | 0A  | LF   | 42  | 2A  | *    | 74  | 4A  | J    | 106 | 6A  | j    |
| 11  | 0B  | VT   | 43  | 2B  | +    | 75  | 4B  | K    | 107 | 6B  | k    |
| 12  | 0C  | FF   | 44  | 2C  | ,    | 76  | 4C  | L    | 108 | 6C  | l    |
| 13  | 0D  | CR   | 45  | 2D  | -    | 77  | 4D  | M    | 109 | 6D  | m    |
| 14  | 0E  | SO   | 46  | 2E  | .    | 78  | 4E  | N    | 110 | 6E  | n    |
| 15  | 0F  | SI   | 47  | 2F  | /    | 79  | 4F  | O    | 111 | 6F  | o    |
| 16  | 10  | DLE  | 48  | 30  | 0    | 80  | 50  | P    | 112 | 70  | p    |
| 17  | 11  | DC1  | 49  | 31  | 1    | 81  | 51  | Q    | 113 | 71  | q    |
| 18  | 12  | DC2  | 50  | 32  | 2    | 82  | 52  | R    | 114 | 72  | r    |
| 19  | 13  | DC3  | 51  | 33  | 3    | 83  | 53  | S    | 115 | 73  | s    |
| 20  | 14  | DC4  | 52  | 34  | 4    | 84  | 54  | T    | 116 | 74  | t    |
| 21  | 15  | NAK  | 53  | 35  | 5    | 85  | 55  | U    | 117 | 75  | u    |
| 22  | 16  | SYN  | 54  | 36  | 6    | 86  | 56  | V    | 118 | 76  | v    |
| 23  | 17  | ETB  | 55  | 37  | 7    | 87  | 57  | W    | 119 | 77  | w    |
| 24  | 18  | CAN  | 56  | 38  | 8    | 88  | 58  | X    | 120 | 78  | x    |
| 25  | 19  | EM   | 57  | 39  | 9    | 89  | 59  | Y    | 121 | 79  | y    |
| 26  | 1A  | SUB  | 58  | 3A  | :    | 90  | 5A  | Z    | 122 | 7A  | z    |
| 27  | 1B  | ESC  | 59  | 3B  | ;    | 91  | 5B  | [    | 123 | 7B  | {    |
| 28  | 1C  | FS   | 60  | 3C  | <    | 92  | 5C  | \    | 124 | 7C  |      |
| 29  | 1D  | GS   | 61  | 3D  | =    | 93  | 5D  | ]    | 125 | 7D  | }    |
| 30  | 1E  | RS   | 62  | 3E  | >    | 94  | 5E  | ^    | 126 | 7E  | -    |
| 31  | 1F  | US   | 63  | 3F  | ?    | 95  | 5F  | _    | 127 | 7F  | DEL  |

---

## APPENDIX F

---

### MMX Technology

---

Shortly after the first Pentium CPU was released, it seemed as if everywhere you looked, multimedia was grabbing all the attention. Intel jumped on the bandwagon too, and after a clever reuse of existing CPU hardware, offered the first Pentium with MMX technology. MMX stands for Multimedia Extensions, Intel's name for a set of new instructions designed to improve the performance of applications utilizing graphics and communication algorithms (such as 3D geometry, DSP filters, and audio processing). The operations typically performed in these applications have a built-in parallelism, which the new MMX instructions and data types are designed to exploit. A processor capable of executing instructions with multiple data operands is called an SIMD (single instruction multiple data) processor.

---

#### MMX DATA TYPES

Figure F.1 shows the four data types used by the MMX instructions. These data types are defined as follows:

- **Packed Byte:** Eight bytes of data are packed into a 64-bit number. Each byte stores eight bits of data. The eight bytes can be loaded with any values or bit patterns required by the calculation.
- **Packed Word:** The 64-bit number is composed of four words, with each word occupying 16-bits.
- **Packed Double-Word:** Two 32-bit double-words are packed into the 64-bit number.
- **Quadword:** A single 64-bit number is the data value.

In Figure F.1, some sample byte, word, double-word, and quadword values or patterns are shown to illustrate the large number of bits capable of being processed simultaneously with a single MMX instruction. We will soon see what we can do with all this data.

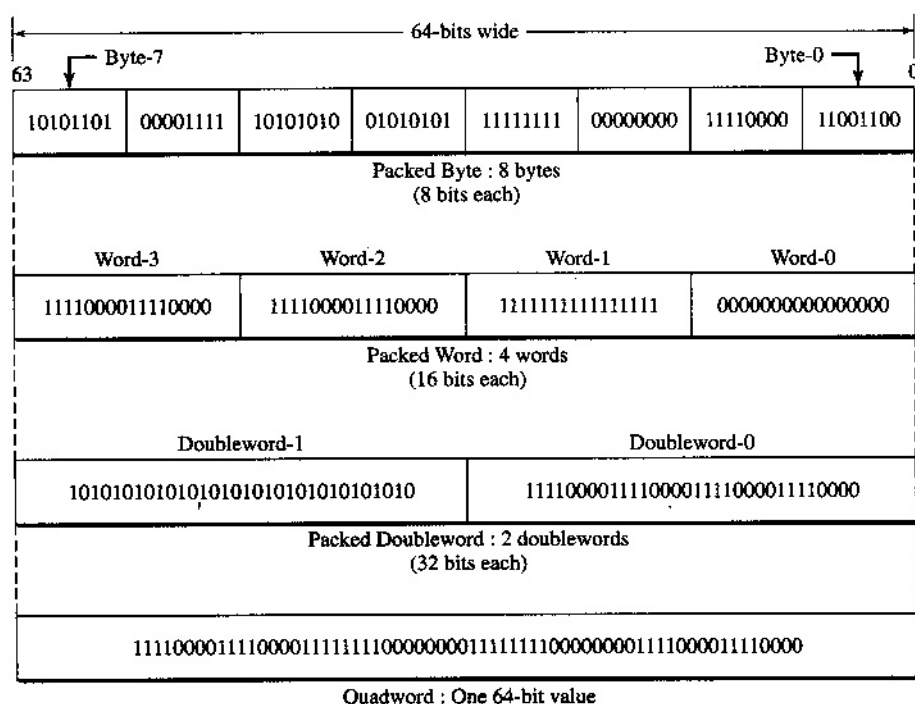


FIGURE F.1 MMX data types

## MMX REGISTERS

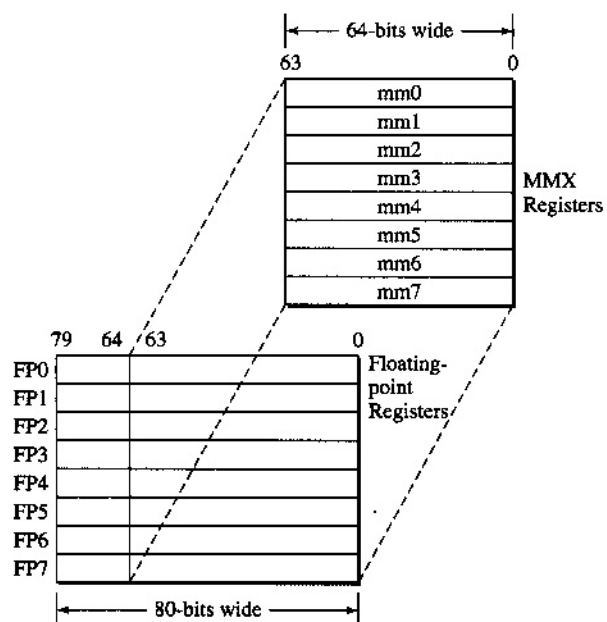
One challenge facing Intel (among the many posed by adding a major new component to the 80x86 architecture) was this: where are the 64-bit MMX data values going to be stored? To get the best performance, there is a need for multiple 64-bit registers inside the CPU, just like there are for the floating-point unit.

Now wait just a minute. There is the answer! Intel chose to *reuse* existing hardware, specifically the lower 64-bits of the eight floating-point registers FP0 through FP7, as illustrated in Figure F.2. The eight new MMX registers, MM0 through MM7, are mapped to the lower 64 bits of each floating-point register. Recall from Chapter 13 that the lower 64 bits of each floating-point register are used to store the mantissa, with the remaining upper 16 bits used for the sign and exponent. These 16 bits are not used by the MMX registers.

### MMX Instructions

Just as we have special instructions associated with the floating-point unit, there are MMX instructions designed to work with the MMX registers (as well as data values stored in memory). The MMX instructions are divided into five categories:

1. Packed Arithmetic, where basic math operations are performed on bytes, words, double-words, or quadwords.
2. Conversions, where hi- and low-precision numbers are converted back and forth.
3. Logical, where the basic Boolean operations and left/right bit-shifting are performed.

**FIGURE F.2** MMX registers and floating-point registers

4. Transfers, where data is moved between memory and the MMX registers, or between the registers themselves.
5. Miscellaneous, which contains the single instruction EMMS (Empty MultiMedia State).

Figure F.3 shows a summary of the instructions within each group. We will examine each group in more detail after covering some additional preliminary information.

Bear in mind that many of the instructions shown are able to operate on all four MMX data types. For example, the PADD instruction comes in the following forms:

- PADDB, for adding packed bytes.
- PADDW, for adding packed words.

**FIGURE F.3** MMX instruction set summaryPacked Arithmetic

|        |        |       |       |      |
|--------|--------|-------|-------|------|
| PADD   | PSUB   | PMULL | PMADD | PSRA |
| PADDQ  | PSUBS  | PMULH |       |      |
| PADDUS | PSUBUS |       |       |      |

PCMPEQ  
PCMPGT

Conversions

|        |         |
|--------|---------|
| PACKSS | PUNPCKL |
| PACKUS | PUNPCKH |

Logical

|       |      |      |
|-------|------|------|
| PAND  | POR  | PSLL |
| PANDN | PXOR | PSRL |

Transfers

MOVD  
MOVQ

Miscellaneous

EMMS

- PADD, for adding packed double-words.
- PADDSB, for performing saturated addition on packed bytes.
- PADDSW, for performing saturated addition on packed words.
- PADDUSB, for performing unsigned saturated addition on packed bytes.
- PADDUSW, for performing unsigned saturated addition on packed words.

Not all MMX instructions have this many options, but there is still a great deal of flexibility available to the programmer using combinations of data types.

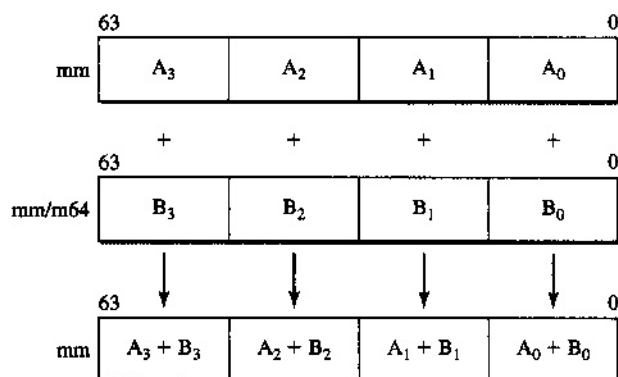
If the words **saturated addition** are new to you, hold on; an explanation is coming up. In the following five tables, the operands for each MMX instruction are listed in symbolic form. Here are the symbols and their interpretation:

|     |                                       |
|-----|---------------------------------------|
| k   | Shift count                           |
| mm  | Any MMX register                      |
| m32 | A 32-bit number stored in memory      |
| m64 | A 64-bit number stored in memory      |
| r32 | Any integer register (EAX, EBX, etc.) |

**MMX Packed Arithmetic Instructions** The Packed Arithmetic instructions listed in Table F.1 provide addition, subtraction, and multiplication of parallel groups of data values. Figure F.4 shows an example of the PADDW instruction. Here, the four 16-bit words stored in each 64-bit MMX register (or memory location) are added together in parallel, giving four

**Table F.1** MMX Packed Arithmetic instructions

| <i>Instructions</i>           | <i>Operand (s)</i> | <i>Meaning</i>                            |
|-------------------------------|--------------------|-------------------------------------------|
| PADDB<br>PADDW<br>PADDD       | mm,mm/m64          | Packed add.                               |
| PADDSB<br>PADDSW              | mm,mm/m64          | Packed add with saturation.               |
| PADDUSB<br>PADDUSW            | mm,mm/m64          | Packed add unsigned with saturation.      |
| PSUBB<br>PSUBW<br>PSUBD       | mm,mm/m64          | Packed subtract.                          |
| PSUBSB<br>PSUBSW              | mm,mm/m64          | Packed subtract with saturation.          |
| PSUBUSB<br>PSUBUSW            | mm,mm/m64          | Packed subtract unsigned with saturation. |
| PMULLW                        | mm,mm/m64          | Packed multiply low.                      |
| PMULHW                        | mm,mm/m64          | Packed multiply high.                     |
| PMADDWD                       | mm,mm/m64          | Packed multiply with add.                 |
| PSRAW<br>PSRAD                | mm,k               | Packed shift-right arithmetic.            |
| PCMPEQB<br>PCMPEQW<br>PCMPEQD | mm,mm/m64          | Packed compare for equal.                 |
| PCMPGTB<br>PCMPGTW<br>PCMPGTD | mm,mm/m64          | Packed compare for greater-than.          |

**FIGURE F.4** Operation of PADDW mm,mm/m64

different 16-bit sums. The four addition operations are performed *at the same time*, not one after another. To illustrate, examine the following non-MMX code sequence that adds four pairs of words together:

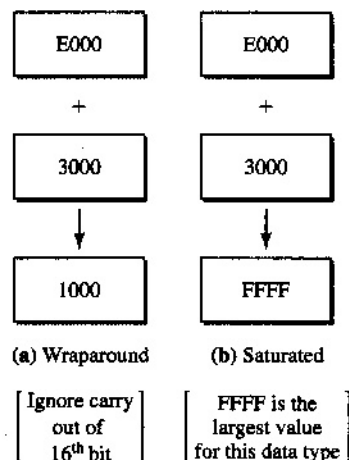
```
03 04    ADD    AX, [SI]
03 5C 02  ADD    BX, [SI+2]
03 4C 04  ADD    CX, [SI+4]
03 54 06  ADD    DX, [SI+6]
```

Here, the four general purpose registers AX, BX, CX, and DX hold one group of four words, and the SI register must point to the beginning of the four words stored in memory. The four ADD instructions execute sequentially, not in parallel, and require eleven bytes of machine code to be fetched and executed. Compare this with the single PADDW instruction that accomplishes the same task:

```
0F FD 24 PADDW  MM4, [SI]
```

The 64-bit MM4 register holds one group of four words, and again the SI register points to the second group of words stored in memory. Only three bytes of machine code are required, which means the PADDW instruction is fetched and executed quicker than the four-instruction ADD sequence, in addition to the gains from parallelism.

A little more attention needs to be paid to the way addition is performed within the MMX environment. Consider the normal addition of two unsigned 16-bit words, as illustrated in Figure F.5(a). The values E000H and 3000H, being unsigned integers, would

**FIGURE F.5** Wraparound and saturated addition of two unsigned words

normally add up to 11000H, but this would require a 17<sup>th</sup> bit for the carry to be saved. Ignoring the carry gives 1000H as the result. The answer has wrapped-around the maximum value of FFFFH possible with 16 bits.

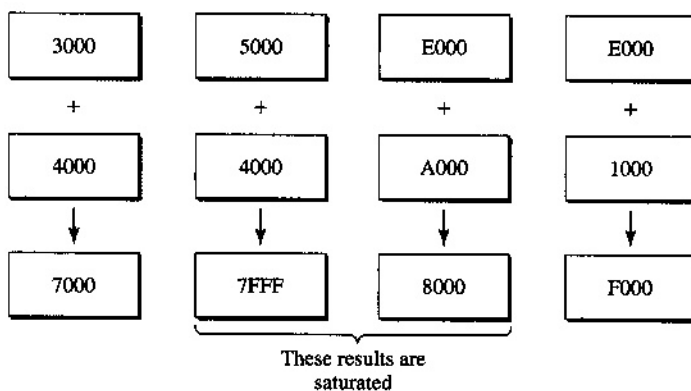
In Figure F.5(b) the result is different because **saturated addition** was used. In saturated addition, there is no wraparound. The sum is limited to a maximum value of FFFFH in this case. Once you have reached FFFFH in an unsigned 16-bit number, you cannot go any higher.

Now, because we also have signed integers as well as unsigned integers, what role does saturated addition play with this data type? Recall that for a 16-bit unsigned integer, the range is 0000H (0) to FFFFH (65,535). When the MSB is used as a sign bit, the range splits in half, with positive and negative ranges as follows:

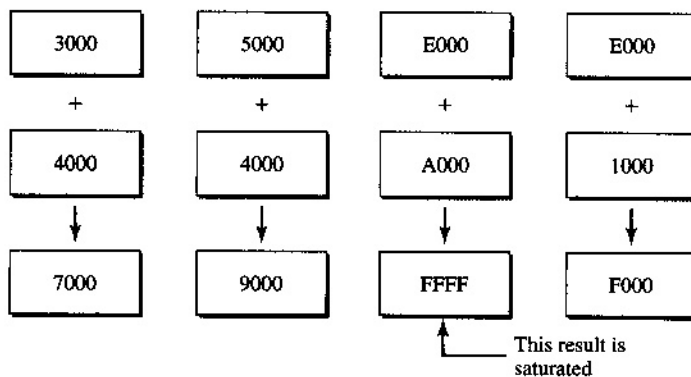
- Positive: 0000H (0) to 7FFFH (32,767)
- Negative: 8000H (−32,768) to FFFFH (−1)

When adding signed integers using saturated arithmetic, a positive result cannot exceed 7FFFH and a negative result cannot exceed 8000H. Strange, but a look at Figure F.6 helps show how this works.

**MMX Conversion Instructions** The MMX Conversion instructions, listed in Table F.2, allow us to pack and unpack groups of data values. Packing data involves converting higher-precision numbers into lower-precision numbers (32-bit integers to 16-bit integers, for



(a) Signed integers



(b) Unsigned integers

**FIGURE F.6** Saturated addition using signed and unsigned integers



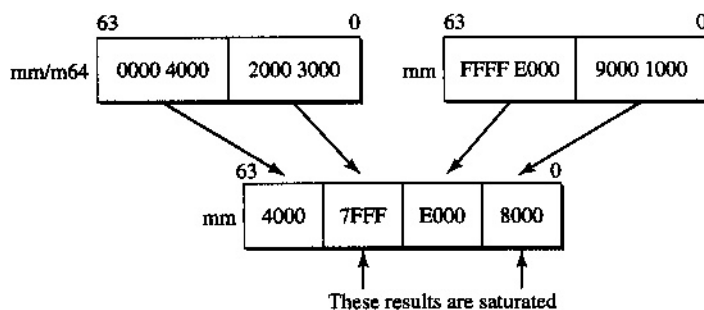
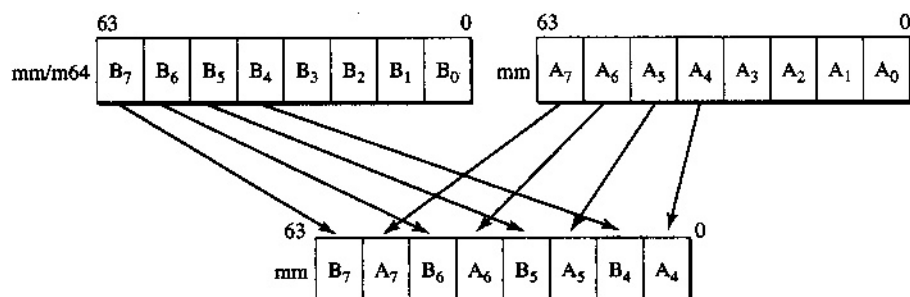
**Table F.2** MMX Conversion instructions

| <i>Instructions</i> | <i>Operand(s)</i> | <i>Meaning</i>                 |
|---------------------|-------------------|--------------------------------|
| PACKSSWB            | mm,mm/m64         | Pack with signed saturation.   |
| PACKSSDW            |                   |                                |
| PACKUSWB            | mm,mm/m64         | Pack with unsigned saturation. |
| PUNPCKLBW           | mm,mm/m32         | Unpack low-packed data.        |
| PUNPCKLWD           |                   |                                |
| PUNPCKLDQ           |                   |                                |
| PUNPCKHBW           | mm,mm/m64         | Unpack high-packed data.       |
| PUNPCKHWD           |                   |                                |

example). Figure F.7 shows an example of packing four double-words into four words using signed saturation.

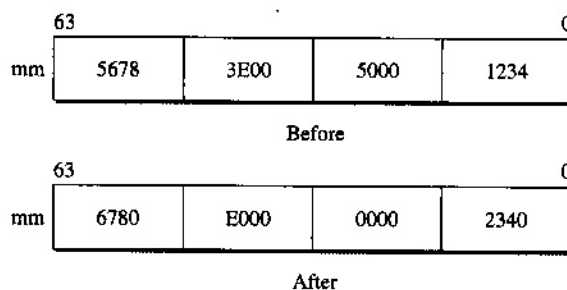
Unpacking data involves interweaving data values from two locations. This operation is useful for transposing rows and columns in matrix operations. Figure F.8 shows an example of how data is unpacked.

**MMX Logical Instructions** The MMX Logical instructions shown in Table F.3 provide the essential bitwise Boolean operations as well as useful left and right shifting. Figure F.9 shows an example of a packed shift left operation.

**FIGURE F.7** Operation of PACKSSDW mm,mm/m64**FIGURE F.8** Operation of PUNPCKHBW mm,mm/m64

**Table F.3** MMX Logical instructions

| <i>Instruction</i> | <i>Operand(s)</i> | <i>Meaning</i>              |
|--------------------|-------------------|-----------------------------|
| PAND               | mm,mm/m64         | Bitwise logical AND.        |
| PANDN              | mm,mm/m64         | Bitwise logical AND NOT.    |
| POR                | mm,mm/m64         | Bitwise logical OR.         |
| PXOR               | mm,mm/m64         | Bitwise logical XOR.        |
| PSLLW              | mm,k              | Packed shift-left logical.  |
| PSLLD              |                   |                             |
| PSLLQ              |                   |                             |
| PSRLW              | mm,k              | Packed shift-right logical. |
| PSRLD              |                   |                             |
| PSRLQ              |                   |                             |

**FIGURE F.9** Operation of PSLLW  
mm,4

A Logical instruction may be followed by a Compare instruction (such as PCMPSEQ) to test the results of a Boolean operation.

**MMX Transfer Instructions** Naturally, the MMX registers need a mechanism in which they can be loaded with data, or moved to another register or memory location after a computation. Table F.4 shows how 32- and 64-bit numbers may be transferred.

**MMX Miscellaneous Instruction** The sophistication of the floating-point unit requires that it be properly synchronized with the processor core. In Chapter 13, we saw that this synchronization is accomplished with the FINIT and FWAIT instructions. Now, with the addition of the MMX registers, cooperation between the floating-point unit and the MMX hardware is even more important. For example, the processor needs to know if an MMX register contains valid information, or if it is empty. Conflicts between the floating-point unit and the MMX registers are possible, with resulting floating-point exceptions, unless the EMMS instruction is used at the end of a section of MMX code. Table F.5 shows the EMMS instruction details.

**Table F.4** MMX Transfer Instructions

| <i>Instruction</i> | <i>Operand(s)</i> | <i>Meaning</i>    |
|--------------------|-------------------|-------------------|
| MOVD               | m32,mm<br>mm,r32  | Move double-word. |
| MOVQ               | mm,m64            | Move quadword.    |

**Table F.5:** MMX Miscellaneous Instruction

| <i>Instruction</i> | <i>Operand(s)</i> | <i>Meaning</i>                                                         |
|--------------------|-------------------|------------------------------------------------------------------------|
| EMMS               | ---               | Empty MultiMedia State. Used for cleanup at the end of an MMX routine. |

**FIGURE F.10** FPU tag word

|           |           |           |           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 15        |           |           |           |           |           |           | 0         |
| FPU Tag-7 | FPU Tag-6 | FPU Tag-5 | FPU Tag-4 | FPU Tag-3 | FPU Tag-2 | FPU Tag-1 | FPU Tag-0 |

Tag Values: 00, 01, 10 – valid; 11 – empty

The processor maintains an FPU tag word, shown in Figure F.10, that identifies the state of each of the eight FPU registers (which double as the eight MMX registers). The EMMS instruction sets all eight tags to the Empty state and also empties the floating-point stack.

## Optimization Issues

There are many things to consider when optimizing the performance of MMX code and its data. Among the most important, we have:

- Use the MMX instruction latencies to help schedule instructions in the pipeline. The latency of an instruction is the number of clock cycles required to get a result after the instruction is issued. For example, knowing an instruction requires three clock cycles may allow you to schedule one or two other 1-cycle instructions to do other work while waiting for the result.
- Because the Pentium is a superscalar architecture, certain pairs of instructions are allowed to be executed together. Pairs of MMX instructions, or mixtures of MMX and integer instructions, can be chosen carefully to keep the pipeline full.
- Mix floating-point and MMX instructions carefully, as both groups of instructions share the same floating-point/MMX registers.
- Take care to align data values properly in memory. For example, never store a word, double-word, or quadword beginning at an odd address (for example, 1003H). This requires extra clock cycles for the processor to perform a read or write. Double-words should be aligned on an even 4-byte address (such as 1004H, 1008H, or 100CH). Quadwords should be aligned on even 8-byte addresses (such as 1008H, 1010H, or 1018H).
- Investigate the use of **software pipelining** to improve performance. Recall that in a hardware pipeline, multiple instructions exist within the pipeline, in various stages of completion. In software pipelining, we use tricks to schedule the flow of instructions through the pipeline so that there are multiple results being computed simultaneously. This is accomplished by interleaving two or more groups of loop code with itself, spacing out the instructions associated with each loop pass according to their latencies. So, each pass through the software-pipelined code produces the equivalent of multiple passes through the original loop code. The pipeline must be loaded with the correct Prologue code to setup the initial pass of the interleaved code, and must finish with more special Epilogue code to complete the final interleaved pass. Sound complicated? It is, and software pipelining may not be performed automatically by an optimizing compiler, so plenty of research and effort may be required to implement this technique.

- Depending on the amount of data that must be processed, it may be necessary to manage the cache to get the desired performance, including preloading the cache with appropriate data to get things started. Constants or blocks of data that are used frequently must be kept in the cache for best performance.
- When using a high-level language, such as C/C++, design the data structures used by the application with attention to how the data values will align in memory.

Having a good understanding of all of these techniques is a great advantage when working with MMX code and data. When tuning your MMX code, take advantage of the performance monitoring events built-in to the Pentium, such as the ability to monitor pipeline stalls and flushes, memory read/write operations, cache hits, and others.

## Improvements and Competitors

Table F.6 shows the evolution of multimedia technology in the Intel family.

With the release of the Pentium III came SSE (streaming SIMD extensions). Recall from Chapter 15 that SSE added the following improvements;

- Additional MMX instructions.
- Cache management instructions.
- A set of 128-bit XMM registers (XMM0 through XMM7), capable of storing a 128-bit integer or four 32-bit floating-point numbers.

SSE2, which arrived with the Pentium 4, allows 64-bit double-precision floating-point numbers and has new instructions for use with the 128-bit XMM registers.

One of Intel's most successful competitors is AMD (Advanced Micro Devices). Not only has AMD emulated the 80x86 architecture, including MMX technology, but they have also added their own 3DNow! multimedia instructions and hardware, which work with integer and floating-point numbers. Table F.7 shows the multimedia evolution of the AMD processors.

Even the PowerPC (Motorola and IBM) has jumped into the multimedia arena. The fourth generation (G4) PowerPC came with an important new hardware component: The AltiVec vector processor engine. To speed up multimedia, encryption, compression, and many other data-intensive applications, the AltiVec engine processes data in 128-bit chunks using a set of 162 SIMD instructions.

## Sample Programs

The companion CD contains several programs designed to help you explore MMX technology. These programs were assembled using MASM 6.11 d, which contains support for MMX instructions (via the .MMX directive).

**Table F.6** Intel processor and multimedia evolution

| <i>Processor</i> | <i>Multimedia Technology</i> |
|------------------|------------------------------|
| Pentium          | MMX                          |
| Pentium II       |                              |
| Celeron          |                              |
| Pentium III      | MMX, SSE                     |
| Celeron II       |                              |
| Pentium 4        | MMX, SSE, SSE2               |

**Table F.7** AMD processor and multimedia evolution

| <i>Processor</i> | <i>Multimedia Technology</i> |
|------------------|------------------------------|
| K6               | MMX                          |
| K6-2             | MMX, 3DNow!                  |
| K6-3             | MMX, 3DNow!                  |
| Athlon           | MMX, 3DNow!                  |
| Duron            | MMX, 3DNow!                  |
| Athlon 4         | MMX, SSE, 3DNow!             |

The first program is called SHOWMMX and is used to display the contents of the MMX registers. Here is a sample execution, showing the state of the MMX registers just after a Windows XP system finished booting.

```
C:\MASM611\BIN> showmmx
MMX Registers
MM0: 00 00 00 1C 00 00 00 70
MM1: 7C 90 E0 27 00 00 00 00
MM2: 00 13 F5 4C 00 00 00 17
MM3: 00 15 94 98 00 00 00 01
MM4: 05 05 03 03 03 03 00 02
MM5: 00 00 00 00 00 00 00 00
MM6: 00 00 00 00 00 13 F9 D8
MM7: 00 00 00 00 00 00 00 00
```

```
C:\MASM611\BIN>
```

The second program is called RUNMMX. This program simply executes the PADDW to verify its operation. However, the program executes PADDW 10 billion times! On a 3.0 GHz Pentium 4 system this takes around 6 seconds. The results look like this:

```
C:\MASM611\BIN> runmmx
Before executing PADDW M7,M6 for 10,000,000,000 passes.
MMX Registers
MM0: 00 00 00 00 00 00 00 00
MM1: 00 00 00 00 00 00 00 00
MM2: 00 00 00 00 00 00 00 00
MM3: 00 00 00 00 00 00 00 00
MM4: 00 00 00 00 00 00 00 00
MM5: 00 00 00 00 00 00 00 00
MM6: 12 34 00 00 11 11 22 22
MM7: 11 22 33 44 55 66 77 88
After executing PADDW M7,M6 for 10,000,000,000 passes.
MMX Registers
MM0: 00 00 00 00 00 00 00 00
MM1: 00 00 00 00 00 00 00 00
MM2: 00 00 00 00 00 00 00 00
MM3: 00 00 00 00 00 00 00 00
MM4: 00 00 00 00 00 00 00 00
MM5: 00 00 00 00 00 00 00 00
MM6: 12 34 00 00 11 11 22 22
MM7: 61 22 33 44 79 66 BF 88
```

```
C:\MASM611\BIN>
```

First, notice that all MMX registers remain unchanged after execution except for MM7 which is the destination register. Second, because PADDW is adding four different groups of words together, the final result in MM7 actually contains four different 16-bit sums. In particular, note that the third word in MM6 is 0000 and the third word in MM7 is 3344, both before and after execution. This clearly shows that the addition being performed in the second group of words (1111 and 5566 initially) has no affect on the third group.

Ten billion passes were needed on the 3.0 GHz Pentium 4 machine to get a total execution time of 6 seconds. What is really interesting now is to see what the NONMMX program does. This program does not use the MMX instructions, just the plain ADD instruction available in real mode. Four ADD instructions are used to perform the simulated

64-bit parallel addition. While NONMMX was being developed, the processor registers became quickly used up. CX was needed for the loop counter, leaving AX, BX, and DX to simulate the operation of MM7, and SI, DI, and BP to simulate the operation of M6. The results are as follows:

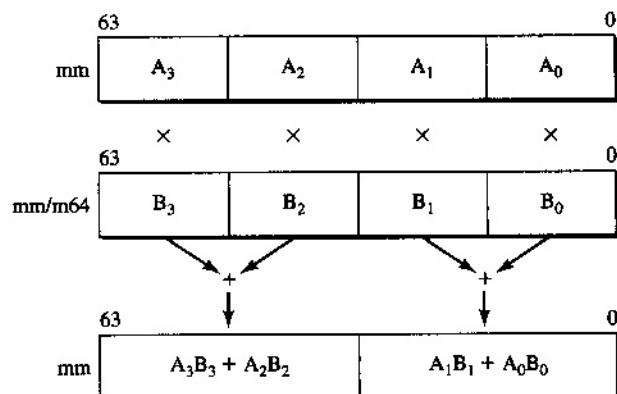
```
C:\MASM611\BIN> nonmmx
Before executing ADD for 10,000,000,000 passes.
AX:BX:DX = 1122:3344:5566
After executing ADD for 10,000,000,000 passes.
AX:BX:DX = 6122:3344:7966

C:\MASM611\BIN>
```

The final results are identical to those of the RUNMMX program. This indicates the same additions were performed correctly. NONMMX took just slightly longer than RUNMMX. That does not mean, however, that the performance improvement is not significant when using MMX instructions, since ADD is one of the fastest and easiest integer operations. A better indicator of how MMX instructions speed up things would be to try the same technique as RUNMMX and NONMMX for a more complicated MMX instruction, such as one that packs or unpacks, or performs saturated addition, or multiply and accumulate. For example, the PMADDWD instruction performs four 16-bit by 16-bit multiplies, giving four 32-bit products, then adds pairs of the 32-bit products together to get a final pair of 32-bit sums. This process is called multiply-and-accumulate and is an important part of many DSP algorithms. The PMADDWD instruction is illustrated in Figure F.11. The program MACMMX executes PMADDWD 10 billion times, taking 16 seconds on the 3.0 GHz Pentium 4 test system.

```
C:\MASM611\BIN>macmmx
Before executing PMADDWD M7,M6 for 10,000,000,000 passes.
MMX Registers
MM0: 00 00 00 00 00 00 00 00
MM1: 00 00 00 00 00 00 00 00
MM2: 00 00 00 00 00 00 00 00
MM3: 00 00 00 00 00 00 00 00
MM4: 00 00 00 00 00 00 00 00
MM5: 00 00 00 00 00 00 00 00
MM6: 12 34 56 78 9A BC DE F0
MM7: 11 22 33 44 55 66 77 88
```

**FIGURE F.11** Operation of PMADDWD mm,mm/m64



After executing PMADDWD M7,M6 for 10,000,000,000 passes.

MMX Registers

```
MM0: 00 00 00 00 00 00 00 00
MM1: 00 00 00 00 00 00 00 00
MM2: 00 00 00 00 00 00 00 00
MM3: 00 00 00 00 00 00 00 00
MM4: 00 00 00 00 00 00 00 00
MM5: 00 00 00 00 00 00 00 00
MM6: 12 34 56 78 9A BC DE F0
MM7: 26 A4 49 64 FC 0A D8 D4
```

C:\MASM611\BIN>

The MULADD program simulates the operation of PMADDWD using integer MUL and ADD instructions. No attempt is made to generate identical numerical results, just the same number of 32-bit products and 32-bit sums. MULADD's execution is as follows:

C:\MASM611\BIN>muladd

Before executing ADD for 10,000,000,000 passes.

AX:BX:DX = 1122:3344:5566

After executing ADD for 10,000,000,000 passes.

AX:BX:DX = FCC4:3344:157C

C:\MASM611\BIN>

MULADD takes 238 seconds to execute on the same system, over fourteen times longer than MACMMX. That is good evidence that the MMX instructions really do provide a significant performance improvement.

---

# Solutions and Answers to Selected Odd-Numbered Study Questions

---

## Chapter 1: Microprocessor-Based Systems

- 1.1: Microwave oven, digital television, heart monitors, FAX machines, UPC scanners, cash registers, automobiles, cameras, sporting equipment, answering machines.
- 1.3: In all three cases, timing signals are required to ensure that data is transferred at the correct time. Specifically, the Serial I/O section uses timing signals to generate the required baud rate clock. The Memory section uses timing signals to control access to dynamic RAMs and to insert wait states as necessary (to allow for chip access times). The Interrupt section uses timing signals to latch interrupt requests and to service certain types of interrupts (for real-time clocks/calendars) at regular intervals.
- 1.7: A cash register without record keeping would require less RAM, because all transactions do not have to be stored.
- 1.9: Doors: any opening/closing. Windows: any opening, closing, breaking. Elevators: all switches (up, down, open, close, floor number). Timing interrupts to keep the elevator from getting stuck on any floor and to close the door after a sufficient delay.
- 1.11: Downloading of main program, status information, serial number of part being assembled, commands from host computer.
- 1.13: The connectors on the plug-in card allow access to all of the required processor signals.
- 1.15: One CPU to run the game program, one CPU to control graphics, one CPU to generate sounds and perform user I/O.
- 1.17: Image processing, system modeling, database search, 3D graphics, and code breaking.
- 1.19: The new CPU must execute all instructions from earlier models. Unused bit patterns from the previous instruction set can be implemented as additional instructions on the new machine. Hardware must be added to decode and execute additional instructions.
- 1.21: The AND instruction takes the longest, because it must both read and write the memory location pointed to by SI.
- 1.23: DEBUG statements assume that input numbers are hexadecimal.



## Chapter 2: An Introduction to the 80x86 Microprocessor Family

- 2.1: AX, BX, CX, DX, SI, DI, and BP. AX is used for multiplication and division. CX is used for loops. SI and DI are used for string operations.
- 2.3: (a)  $03E00 + 1F20 = 05D20$   
(b)  $04000 + 3FFE = 07FFE$   
(c)  $24000 + FFFF = 33FFF$
- 2.5: No. The U pipeline is used for FPU operations as well as integer instructions.
- 2.7: A segment is a 64KB block of memory beginning on any 16-byte boundary (paragraph).
- 2.9: Intel byte swapping refers to the way 16-bit numbers are stored in memory. The lower 8 bits are stored in location N, the upper 8 bits in location N+1.
- 2.11: The processor allows for eleven different addressing modes.
- 2.13: Register addressing is fast because the physical register hardware is contained within the CPU.
- 2.15: `MOV AX,1000H` places the immediate value 1000H into register AX. `MOV AX,[1000H]` reads the word stored at location 1000H and places its value into AX.
- 2.17: An interrupt is an event that temporarily changes execution to service a request.
- 2.19: The processor supports 256 interrupts.
- 2.21: The 80286 supports a 16MB physical memory space, protected mode operation, virtual memory, and multitasking.
- 2.23: A page fault occurs when the processor attempts to access a memory page that has not been loaded into memory.
- 2.25: Physical memory has a greater effect on the number of page faults. The more physical memory, the smaller the chance that a page will not have been previously loaded.
- 2.27: Applications may include imaging, world-wide databases, real-time simulation of weather, chemical reactions, and astronomical experiments.
- 2.29: The 486 uses an 8KB cache memory, has a redesigned internal architecture resulting in fewer clock cycles per instruction, and contains an on-chip coprocessor.
- 2.31: The Pentium differs from other 80x86 CPUs in that it was designed with RISC techniques in mind, such as branch prediction and dual integer pipelines.

## Chapter 3: 80x86 Instructions, Part 1

- 3.1: `ORG` is used to set the initial program counter for the program being assembled. `DB` is used to define (reserve) a byte of storage. `DW` is used to define a word of storage. `END` is used to tell the assembler that it has reached the end of the source file.
- 3.3: Object (.obj) and list (.lst) files.
- 3.5: 16-bit numbers are stored in two consecutive memory locations with the lower byte going into the first location and the upper byte into the second. Thus, the two bytes are swapped.
- 3.7: A linker is used to combine multiple object files into a single object file.
- 3.9: (a) register, (b) immediate, (c) indexed, (d) based, (e) indexed, (f) relative, (g) implied.
- 3.11: Upon completion, AX contains the original contents of BX. BX contains the original contents of CX, and CX contains the original contents of AX.
- 3.13: (a) 9C0A, (b) B29C, (c) 78B2

- 3.15: LDS loads a string pointer using the DS register; LES uses the ES register. LDS should be used to load source string addresses. LES should be used for destination strings.
- 3.21: The simplified segment directives are .DATA and .CODE.
- 3.23: (a) EAX is a base register.  
(b) EBX is a base register. EAX is an index register.  
(c) EAX is a base register. EBX is an index register.  
(d) ESI is a base register. EDI is an index register.
- 3.25: Scale values of 1, 2, 4, or 8 may be used.
- 3.27: The port address must be placed in register DX, which is used as an operand in IN and OUT.
- 3.29: (a) EBX contains 00003000H.  
(b) EBX contains 00003000H.
- 3.31: The PUSH/POPA instructions can be used to save/restore all general purpose registers via the stack.

## Chapter 4: 80x86 Instructions, Part 2

- 4.1: AH is unchanged. AL contains the sum of AL and AH (46H).
- 4.3: After the INC instruction executes, location 2000 will contain 01 and location 2001 will contain 50.
- 4.5: The zero flag is cleared.
- 4.7: MOV BX, 25  
MUL BX
- 4.9: DX contains 0000. AX contains FDE8.
- 4.11: CBW converts 30H into 0030H, and 98H into FF98H.
- 4.13: DX contains 8365H.
- 4.15: AL contains 6AH.
- 4.17: ROR BYTE PTR[1000H],1
- 4.19: JZ
- 4.21: (a) no, (b) no, (c) yes, (d) no, (e) cannot determine without OF status, (f) cannot determine without OF status.
- 4.25: SP for near call: 27FEH. SP for far call: 27FCH.
- 4.27: First the flags are pushed onto the stack. Then the trace and interrupt-enable flags are cleared. Then the CS and IP registers are pushed. The interrupt vector table is accessed for the address of INT 16H's ISR. Then the CS and IP registers are loaded with their new addresses and execution resumes.
- 4.29: ADD AL, BL  
ADD AL, DL  
MOV CL, AL
- 4.31: Dividing by 8 is the same as multiplying by 0.125.  
MOV BL, 8  
DIV BL
- 4.33: MUL multiplies two unsigned 16-bit numbers. Each can be as large as 65535. IMUL multiplies two signed 16-bit numbers. The largest positive number that can be used is 32767.
- 4.35: AND BX, 23F9H
- 4.39: To pop the stack and return to the correct place in the calling program.
- 4.41: MUL BL  
MOV DX, 0  
MOV CX, 2  
DIV CX

```

4.43: MOV     AL, STATUS
      TEST   AL, 80H
      JZ     ROUT1
      TEST   AL, 0CH
      JZ     ROUT3
      TEST   AL, 20H
      JNZ    ROUT2
      AND    AL, 3
      CMP    AL, 2
      JZ     ROUT4
4.45:      MOV     AL, COUNT
      CLC
      INC     AL
      DAA
      CMP     AL, 60H
      JNZ     DONE
      SUB     AL, AL
DONE:  MOV     COUNT, AL

```

## Chapter 5: Interrupt Processing

- 5.1: The processor's environment is the information contained in all of its internal registers. Saving the environment during interrupt processing ensures that the state of the machine prior to the interrupt will be correctly restored.
- 5.3: INTR is enabled with the STI instruction and disabled with the CLI instruction.
- 5.7: 

```
MOV DI, 94H
MOV [DI], 9AE2H
MOV [DI+2], 3C0H
```
- 5.9: 280H divided by 4 gives 0A0H. The address range is for INT 0A0H.
- 5.11: After 1st instruction: AX = E03F.  
 After 2nd instruction: AX = E040.  
 After 3rd instruction: AX = DF40.  
 After 4th instruction: AX = 20BF.
- 5.13: 2.5 ms divided by 0.2  $\mu$ s gives 12,250. An average of 12,250 instructions can be executed after the power failure.
- 5.15: 1FH.
- 5.17: C0, C8, D0, D8, E0, E8, F0, F8.
- 5.19: Yes, because NMITIME is called every 16.67 ms. It is possible that interrupts would be lost, resulting in inaccurate timekeeping.
- 5.21: ISR20H will select the DIV BL instruction. The resulting value in AX is 03C0.
- 5.23: The new program counter will be 0480:1234.
- 5.25: The results of each instruction can be examined, effectively running the program in slow motion and watching all the results.
- 5.27: INTO will not generate an interrupt. Trace is enabled. Interrupts are enabled.
- 5.29: 

```
T7:  PUSH BX
      PUSH DI
      MOV  BL, 7
      IMUL BL
      CMP  AX, 8400H
      JLE  EXIT
      PUSH AX
```

```

CALL FAR PTR OVERSCAN
POP AX
EXIT: POP DI
POP BX
IRET

```

## Chapter 6: An Introduction to Programming the 80x86

- 6.1: Writing programs in pseudocode is often easier than writing programs in assembly language. For many programmers, a statement like IF CHAR = 'A' THEN ACOUNT = ACOUNT + 1 is easier to come up with than the equivalent assembly language statements.
- 6.3: The DS register must be loaded with the segment address of the source strings, and ES must be loaded with the segment address of the destination strings.
- 6.5: Add an additional byte to the BCD string to save a signed exponent.
- 6.7: Fractional values can be more accurately represented in BCD. Consider the example value 0.7. The binary representation of 0.7 repeats over and over, whereas the BCD representation is 07 with an appropriate exponent.
- 6.9: Add an additional byte for use as a sign byte.
- 6.11: Field terminators are useful for searching purposes. Looking for a field terminator is one way of finding the end of a record, or the beginning of the next record.

## Chapter 7: Advanced Programming Applications

- 7.1: The PUBLIC assembler directive is used to make the value of a symbol available to other link modules. The EXTRN directive is used to access the value of an external symbol.
- 7.3: A nested macro is a macro definition that contains a second macro definition within itself.
- 7.7: DEBUG can display the interrupt vector for INT 21H by using the following command: D 0:84 L 4
- 7.11: DOS manages memory by assigning blocks of paragraphs beginning at the first free segment address.
- 7.13: One way to access mouse data is through INT 33H. Mouse information is available when a mouse is connected and an appropriate mouse driver is loaded.
- 7.15: The size of the resident code (in paragraphs) must be determined prior to the exit to DOS through DOS INT 21H, Function 31H.
- 7.17: The BTEXT (board text) string contains the printable text of the tic-tac-toe board. Nine locations within the BTEXT string are modified during game play with the user/computer moves. These locations are specified in the BPOS (board position) array.
- 7.19: The stack in a C program is used to hold variable values, return addresses, and pointers to other stack areas.

## Chapter 8: Hardware Details of the 8088

- 8.1: Some CPU signals are different (ALE vs.  $QS_0$  for example). Minmode does not allow bus request/grant and does not support a coprocessor.
- 8.5:  $10 \text{ MHz}/3 = 3.333 \text{ MHz}$ .
- 8.9: The initial instruction fetch is from address FFFF0.
- 8.11: HOLD/HLDA or  $\overline{RQ}/\overline{GT}$ .
- 8.13: The processor will automatically be interrupted each time interrupts are enabled.
- 8.15: A 74LS374 or two 74LS75s.

8.17: The address 3A8C4 is represented as follows:

|     |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |    |
|-----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|----|
| A19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | A0 |
| 0   | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0  |

8.19: Demultiplexing takes additional clock cycles.

8.23:  $T_1$

8.25: The bus activity on the entire system will be very busy. The bus activity of each processor will have gaps in processing as the other processor alternately takes over.

|            |            |            |            |        |
|------------|------------|------------|------------|--------|
| System Bus | -- busy -- | -- busy -- | -- busy -- | -- ... |
| CPU #1 Bus | -- busy -- | -- idle -- | -- busy -- | -- ... |
| CPU #2 Bus | -- idle -- | -- busy -- | -- idle -- | -- ... |

8.27: An 8-bit interrupt vector number.

## Chapter 9: Memory System Design

9.1:  $AD_0$  through  $AD_7$  act as a bidirectional data bus. They also supply the lower byte of the address bus and receive an 8-bit vector number during  $\overline{INTA}$ .

9.3: Minmode:  $\overline{IO}/\overline{M}$ ,  $\overline{RD}$ ,  $\overline{WR}$ .  
Maxmode:  $\overline{MRDC}$ ,  $\overline{MWTC}$ .

9.5:  $4(250 \text{ ns}) = 1,000 \text{ ns} = \mu\text{s}$ .

9.7:  $2^{17} = 128\text{KB}$ . Seventeen address lines are needed ( $A_0$  through  $A_{16}$ ).  
 $2^{21} = 2\text{MB}$ . Twenty-one address lines are needed ( $A_0$  through  $A_{20}$ ).

Note: The 8088 cannot access 2MB of memory.

9.9: Address line  $A_{11}$  is used to select EPROM #1 when low and EPROM #2 when high. Address lines  $A_{12}$  through  $A_{19}$  must be used in the address decoder.

9.13: (a) 2A000–2AFFF  
(b) 54000–57FFF  
(c) 40000–5FFFF  
(d) E0000–EFFFF

9.15: A: 60000–63FFF  
B: 64000–67FFF  
C: 68000–6BFFF  
D: 6C000–6FFFF  
E: 70000–73FFF  
F: 74000–77FFF  
G: 78000–7BFFF  
H: 7C000–7FFFF

9.17: Advantages: less hardware needed, easier to design. Disadvantages: future expansion requires additional hardware, addresses may conflict or select more than one device at a time.

9.23: Each address line provides a row address bit and a column address bit. These two bits can be in four different states.

9.25: The interrupt service routine could maintain a counter (for RAS refreshing). Each time the interrupt occurs, the counter could be written to the DRAM address circuitry to cause the refresh.

9.27: Waiting in hold acknowledge.

## Chapter 10: I/O System Design

10.1: First way:  $\text{IN AL}, 40\text{H}$   
Second way:  $\text{MOV DX}, 40\text{H}$   
 $\text{IN AL}, \text{DX}$

10.5: Use  $\overline{\text{IORC}}$  and  $\overline{\text{IOWC}}$  instead of  $\overline{\text{RD}}$ ,  $\overline{\text{WR}}$ , and  $\overline{\text{IO}/\text{M}}$ .

10.9: (a) C0 - FF

(b) 1 0 x 0 0 x 0 0 <---> 80, 84, A0, A4

(c) 0 C B A 0 1 x x

PA: 04-07

PB: 14-17

PC: 24-27

PD: 34-37

PE: 44-47

PF: 54-57

PG: 64-67

PH: 74-77

(d) 1 x 0 x 1 x 1 x 0 x 1 x 1 1 x x

There are 128 different addresses. Here are a few of them: 8A2C, 8A2D, 8A2E, 8A3C, 8A6C, 8B2C, 8E2C, 9A2C, and CA2C.

(e) 8001, plus 16383 more that have their LSB and MSB set.

(f) x 0 1 0 0 1 x x <---> 24 - 27, A4 - A7

```

10.15: KP:      MOV    AL, 0E0H
              MOV    CX, 6
ALOP:   OUT     9CH, AL
              CALL   DELAY
              SHR     AL, 1
              LOOP   ALOP
              MOV     AL, 0EH
              MOV     CX, 4
BLOP:   OUT     9CH, AL
              CALL   DELAY
              SHL     AL, 1
              LOOP   BLOP
              JMP     KP
DELAY:   PUSH    AX
              PUSH    CX
              IN      AL, 9CH
              MOV     CH, AL
              MOV     CL, 1
WAIT:   NOP
              NOP
              LOOP   WAIT
              POP     CX
              POP     AX
              RET

```

10.17: Assuming no DELAY cycles, 10-byte waveforms require  $10 \times 17 = 170$  cycles. With a clock speed of 5 MHz, this works out to  $34 \mu\text{s}/\text{cycle}$ , or 29.4 KHz. Fifty-byte waveforms require  $50 \times 17 = 850$  cycles, giving a frequency of 5,882 Hz; 100-byte waveforms require  $100 \times 17 = 1,700$  cycles, giving a frequency of 2,941 Hz.

10.19: This cannot be done, because 8A does not begin on a 4-port boundary (such as 88 or 8C).

10.21: Assume 8255 is programmed for Aout, Bin.

```

BINCNT:  MOV     AL, 0
DISPCNT: OUT     48H, AL
              CALL DELAY
              INC     AL
              JMP     DISPCNT

```

```

DELAY:  MOV  BL, AL
        IN   AL, 49H
        MOV  CH, AL
        MOV  CL, 1
WAIT:   NOP
        NOP
        LOOP WAIT
        MOV  AL, BL
        RET

10.25:  11/1,200 = 9.17 ms.
10.27:  (a) MOV AL, 0DAH
        OUT  DX, AL
        (b) MOV AL, 4DH
        OUT  DX, AL
        (c) MOV AL, 48H
        OUT  DX, AL
10.29:  CHAROUT: MOV  AH, AL
TSTAT:  IN   AL, 79H
        AND  AL, 1
        JNZ  TSTAT
        MOV  AL, AH
        CMP  AL, 1
        JC   COUT
        CMP  AL, 1BH
        JNC  COUT
        PUSH AX
        MOV  AL, '^'
        CALL CHAROUT
        POP  AX
        ADD  AL, 40H
        JMP  CHAROUT
COUT:   OUT  78H, AL
        RET

```

## Chapter 11: Interfacing with the 80x86

```

11.1:  The CPU cannot perform other processing while polling.
11.3:  ICW1: 12, ICW2: C0, ICW3: 00 (although not needed).
11.7:  MOV  AL, 12H
        OUT  70H, AL
        MOV  AL, 0C0H
        OUT  71H, AL
11.11: MOV  AL, 0B0H
        OUT  0B3H, AL
        SUB  AL, AL
        OUT  0B2H, AL
        MOV  AL, 30H
        OUT  0B2H, AL
11.13: LATCH0: MOV  AL, 0
        MOV  DX, 0CC83H
        OUT  DX, AL

```

```

        SUB DX,3
        IN  AL,DX
        MOV DL,AL
        IN  AL,DX
        MOV DH,AL
        CMP DX,7
        JNZ EXIT
        CALL TIMEOUT
EXIT:    RET
11.15: 10,000/250K = 40 ms.
11.19: SWAVE: MOV AL,36H
        OUT 3CH,AL
        SUB DX,DX
        MOV AX,2000
        SUB BH,BH
        DIV BX
        OUT 38H,AL
        MOV AL,AH
        OUT 38H,AL
        RET
11.21: 1,020.6 = 1111111100.1001
        Normalized with exponent of 9 = 1.1111111001001
        23-bit mantissa = .11111110010011001100110
        Exponent = 127 + 9 = 136 = 10001000
        Sign bit = 0
        01000100    01111111    00100110    01100110
           44         7F         26         66
11.23: FLD  LEVEL
        FSQRT
        FST  LEVEL
11.25: PI  DD  3.14159265
        C180 DD  180.0
DTR:   FLD  DEG
        FMUL PI
        FDIV C180
        FST  RAD
        RET
11.27: Image processing, matrix calculations, fft (fast Fourier transform), spreadsheets,
        signal processing, graphics (rotation, scaling).
11.29: NEG5 DD  -5.0
EQU:   FLD  A
        FMUL A
        FLD  B
        FMUL B
        FLD  NEG5
        FMUL A
        FMUL B
        FADD
        FADD
        FST  X
        RET

```



## Chapter 12: Building a Working 8088 System

12.3: Use 138 outputs 0 (0000–3FFF), 2 (4000–7FFF), 4 (8000–BFFF), and 6 (C000–FFFF) for EPROM  $\overline{CS}$  inputs.

```
12.9: INIT:  MOV AL, 90H
            OUT 0E3H, AL
            SUB AL, AL
            OUT 0E1H, AL
            MOV AL, 3CH
            OUT 0E2H, AL
            RET
```

```
12.11: DXOUT: MOV AX, DX
            SUB DX, DX
            MOV BX, 10000
            CALL DOUT
            MOV BX, 1000
            CALL DOUT
            MOV BX, 100
            CALL DOUT
            MOV BX, 10
            CALL DOUT
            ADD AL, 30H
            CALL C_OUT
            RET
```

```
        DOUT: DIV BX
            ADD AL, 30H
            CALL C_OUT
            MOV AX, DX
            SUB DX, DX
            RET
```

12.15: Store commands in a data table (i.e., db 'DUMPDISPGO EXEC...'). Compare input text with each four-letter group in the table. If a match is found, look up corresponding routine address in a second data table (i.e., dw D\_UMP, D\_UMP, E\_XEC, E\_XEC,...). Read this address and jump to it.

```
12.17 V_ERIFY: MOV DI, 500H
            NEXT: MOV BL, 0ABH
                CALL TESTLOC
                MOV BL, 65H
                CALL TESTLOC
                INC DI
                CMP DI, 1F80H
                JNZ NEXT
                JMP GET_COM
TESTLOC: MOV ES: [DI], BL
            MOV AL, ES: [DI]
            CMP AL, BL
            JZ OK
            PUSH AX
            PUSH BX
            CALL CRLF
            MOV DX, DI
            CALL H_OUT
```

```

CALL BLANK
POP  BX
MOV  AL,BL
CALL HTOA
CALL BLANK
POP  AX
CALL HTOA
OK:  RET

```

```

12.19: HEXPM: CALL GET_BYT
MOV  DL,AL
CALL GET_BYT
MOV  AL,DH
CALL CRLF
MOV  AL,DH
ADD  DL
CALL HTOA
CALL BLANK
MOV  AL,DL
SUB  AL,DH
CALL HTOA
JMP  GET_COM

```

12.21: It might be useful to include the segment address in the dump, as in 01CE:0240 3E 43 12 06 0F E5 . . . , and printing the ASCII equivalent characters after the 16 bytes would also be a good addition.

12.23: Turn I\_NEX code (up to NUN) into a subroutine. Use SI to pass the index of the R\_LETS and R\_DATA tables. Rewrite I\_NIT so that I\_NEX is called eleven times if no register name is specified. Otherwise, search R\_LETS for the supplied register name and adjust SI accordingly.

12.25: Add PIN2: to the MOV AL, 'I' instruction. Insert these instructions between CALL C\_OUT and JMP GET\_COM:

```

IN  AL,S_CTRL
AND AL,2
JZ  PIN2
IN  AL,S_DATA
AND AL,7FH
CMP AL,0DH
JNZ PIN2

```

```

12.27: P_IE: MOV AL,SINE[SI]  8 + 9
OUT  D_AC,AL                10
ADD  SI,1                    4
LOOP P_IE                    17

```

These 48 cycles are repeated 256 times, giving a total estimate of 12,288 cycles. The period is approximately  $12,288 / 3.333 \text{ MHz} = 3.68 \text{ ms}$ . The frequency is approximately 271 Hz.

## Chapter 13: Hardware Details of the Pentium

13.1:  $A_3$  through  $A_{31}$ , and the eight-byte enable outputs, are used to output a 32-bit address. The byte enable outputs simulate the operation of  $A_0$ ,  $A_1$ , and  $A_2$ .

13.3: Parity checking is performed on address and data lines.

- 13.5: Processor signals  $\overline{D/C}$ ,  $\overline{M/\overline{IO}}$ ,  $\overline{W/R}$ ,  $\overline{CACHE}$ , and  $\overline{KEN}$  define the type of bus cycle currently running. The byte enable outputs specify special cycles in some cases.
- 13.7:  $\overline{INTR}$ ,  $\overline{NMI}$  and  $\overline{SMI}$ .
- 13.9: The cache, in many cases, is able to supply a stored copy of the requested memory data, without the need for a main memory access.
- 13.11: Branch prediction is a technique used to predict the outcome of a conditional branch instruction. When the prediction is correct, instructions are kept flowing smoothly through the instruction pipelines.
- 13.13: The Pentium indicates the current bus cycle type through its  $\overline{D/C}$ ,  $\overline{M/\overline{IO}}$ ,  $\overline{W/R}$ ,  $\overline{CACHE}$ , and  $\overline{KEN}$  signals.
- 13.15: Single transfer: up to eight bytes. Burst transfer: 32 bytes.
- 13.17: Locked bus cycles are used to restrict access to shared variables, such as semaphores. The current bus cycle is locked if the  $\overline{LOCK}$  output is active.
- 13.21: No,  $\text{ADD DATA}[\text{BP}],3$  is not a simple instruction. It contains both a displacement value and an immediate operand.
- 13.23:  $\text{PF}$ ,  $\text{D1}$ ,  $\text{D2}$ ,  $\text{EX}$ , and  $\text{WB}$ .
- 13.25: A pipeline stall is one or more clock cycles in which nothing happens in the instruction pipeline. Stalls are caused by dependencies between instructions and also by long main memory accesses. Having valid copies of requested data in cache reduces the number of stalls by reducing main memory accesses.
- 13.27: The history bits in a BTB entry are used to indicate results of the most recent branches (taken, not taken).
- 13.29: The BTB predicts "not taken" for a new conditional jump.
- 13.31: A cache hit is a cache access that results in a tag match.
- 13.37: Writethrough: writes to the cache, also updates main memory. Writeback: writes to the cache, does not update main memory.
- 13.39: MESI stands for modified exclusive shared invalid, a protocol used to help maintain cache coherency.
- 13.41: An inquire cycle is used to determine if the cache contains a copy of a shared data item.
- 13.43: The U pipeline makes up the first five stages of the FPU pipeline.

## Chapter 14: Protected-Mode Operation

- 14.1: The additional protected-mode registers are the control registers, the debug registers, the GDTR, LDTR, IDTR, and TR.
- 14.3: A segment descriptor controls the base address, length, and access permissions of a segment of memory.
- 14.5: A logical address uses a segment selector and a 32-bit offset to access memory. The segment selector points to one of 8,192 segment descriptors in the GDT or LDT, which supplies a 32-bit linear base address to combine with the 32-bit offset.
- 14.7: A virtual address is translated into a physical address through two levels of page tables, a PDE and a PTE. The upper 20 bits of the virtual address are replaced by 20 bits from the PTE, giving a new, 32-bit physical address.
- 14.9: A page directory contains 1,024 entries. Each entry can access 1,024 PTEs. Each PTE represents 4KB of memory. So, 1,024 times 1,024 times 4KB gives 4 gigabytes, the entire addressing space of the processor.
- 14.11: Dirty bit: indicates if page has been written to. Accessed bit: indicates if a read or write has been performed on a page.
- 14.15: The RPL bits in the segment selector play a role in protected accesses.
- 14.17: The purpose of a call gate is to protect access to the entry point of a procedure.

- 14.21: A user-level task may not access a supervisor page.
- 14.23: The TSS is used to support task switching. The TSS contains a copy of each processor register for the task it is associated with.
- 14.25: The LTR and STR instructions are used with the task register. They may only be executed in protected mode.
- 14.29: The IDTR is the interrupt descriptor table register. The IDT is the interrupt descriptor table. The IDTR is used to locate the base address of the IDT in memory. The IDT stores interrupt descriptors.
- 14.31: Task gates, interrupt gates, and trap gates may be used in the IDT.
- 14.33: The page fault error code provides valuable information that may be used to correct the page fault (swap in a new page, choose a victim).
- 14.37: Divide C0H by eight (the number of bytes in a descriptor). This gives vector number 18H.
- 14.39: 32 bytes are required in the I/O permission bit map to protect ports 0 through FFH.
- 14.43: A virtual-8086 monitor is a protected-mode program that watches over a running virtual-8086 mode task, providing services such as interrupt handling.
- 14.45: No, virtual-8086 mode may not be entered from real mode.

## Chapter 15: The Pentium II and Beyond

- 15.1: Enhanced pipelining, internal level-2 cache, dual-independent bus architecture, and speculative execution.
- 15.3: Here is one possible ordering:

| <i>Clock Cycle</i> | <i>Pipeline #1</i> | <i>Pipeline #2</i> |
|--------------------|--------------------|--------------------|
| 1                  | MOV AL,[SI]        | MOV AL,[DI]        |
| 2                  | busy               | busy               |
| 3                  | busy               | busy               |
| 4                  | MUL DL             | ADD SI,2           |
| 5                  | busy               | DEC DI             |
| 6                  | MOV [BP],AX        | INC BX             |
| 7                  | busy               | idle               |
| 8                  | busy               | idle               |
| 9                  | AND AL,AH          | idle               |
| 10                 | ADC AL,0           | idle               |

Ten clock cycles is a good savings over the sixteen clock cycles required in the original order. Pipeline #2 must remain idle after the INC BX, because the next instruction (AND AL,AH) affects AL, and AX is being used in Pipeline #1 with the MOV [BP],AX instruction.

- 15.5: Micro-ops are the individual components that make up an instruction, such as the data movement operations.
- 15.7: Instructions can execute out of order as long as there are no register or flag dependencies between the instructions.
- 15.13: Hyperthreading is the concurrent execution of two instructions. Hyperthreading is accomplished by interweaving micro-ops from each instruction through a single instruction pipeline.
- 15.15: (1) Hyper-pipelined technology, (2) Rapid Execution Engine, (3) Execution Trace Cache.
- 15.17: EM64T (Extended Memory 64 Technology) allows 64-bit computing (via cooperation from the operating system) on integers, pointers, and registers. EM64T extends the physical address space to 1 Terabyte (1024 GB) and the virtual address space to a flat 64-bit model.

- 15.21: The Celeron D processor contains support for EM64T and the Execute Disable Bit, as well as SSE3.
- 15.23: Simultaneous execution means two or more instructions are executing at the same time (there are two or more instruction pipelines). Concurrent execution means that two or more instructions complete execution during a particular time period (sharing a single instruction pipeline). Hyperthreading provides concurrent execution, while a dual-core processor provides simultaneous execution.
- 15.25: Unrolling the loop four times requires the following code:

```
MOV    CX, 250
TOP:   INC    BX
      MUL    BX
      ADD    BP, AX
      ADC    DI, DX
      INC    BX
      MUL    BX
      ADD    BP, AX
      ADC    DI, DX
      INC    BX
      MUL    BX
      ADD    BP, AX
      ADC    DI, DX
      INC    BX
      MUL    BX
      ADD    BP, AX
      ADC    DI, DX
      LOOP  TOP
```

Because the loop performs 16-bit additions and multiplications, all 16-bit register values could be placed into one or two MMX registers for parallel calculations.

- 15.29: (1) Low-power CPU designs, (2) chipset, (3) wireless networking.



# The Intel® Microprocessor Family: Hardware and Software Principles and Applications

James L. Antonakos

Readers will be able to build and program their own 8088 single-board computer by applying the interfacing concepts and techniques presented in this book. Coverage begins with the software architecture of the 80x86 family, including the software model, instruction set and flags, and addressing modes. Abundant examples illustrate basic programming concepts such as the use of data structures, numeric conversion, string handling, and arithmetic. Hardware details of the entire 80x86 family are then examined, from pin and signal descriptions to memory and input/output system design. Advanced topics, including protected mode, WIN32 and Linux programming, and MMX technology are also introduced.

## Features:

- Robust coverage of hardware and software principles provides readers with a good overall exposure to the 80x86 family.
- Features a wide variety of interfacing applications, including a complete microprocessor-based system that can be easily duplicated.
- Teaches the relationship between the use of C programming and assembly language for interfacing and control.
- Coverage of 16-bit and 32-bit programming, exposure to the Windows WIN32 and Linux execution environments, and the process of structured programming provides readers with a strong programming foundation.
- Accompanying CD contains source and executable files for all example programs, datasheets in PDF format for easy reference, software and hardware lab experiments, and historical examples of programming in the DOS environment.

## About the Author:

James L. Antonakos is a Professor of Computer Studies at Broome Community College in Binghamton, NY.

## Also Available from Thomson Delmar Learning:

*Microprocessor Architecture, Programming, and Systems featuring the 8085* / Routt

Order #1-4180-3241-7

*Fundamentals of Microcontrollers and Applications in Embedded Systems (with the PIC18 Microcontroller Family)* / Gaonkar

Order # 1-4018-7914-4

*The 8051 Microcontroller, 3E* / Ayala

Order #1-4018-6158-X

*The 8086 Microprocessor: Programming & Interfacing the PC* / Ayala

Order #0-3140-1242-7

Visit [www.delmarlearning.com/electronics](http://www.delmarlearning.com/electronics) or [www.electronictech.com](http://www.electronictech.com) for your lifelong learning solutions  
For more learning solutions by Thomson: [www.thomson.com/learning](http://www.thomson.com/learning)

**THOMSON**  
★  
**DELMAR LEARNING**

ISBN-13 978-1-4180-3845-8  
ISBN-10 1-4180-3845-8



9 781418 038458

9 0000

