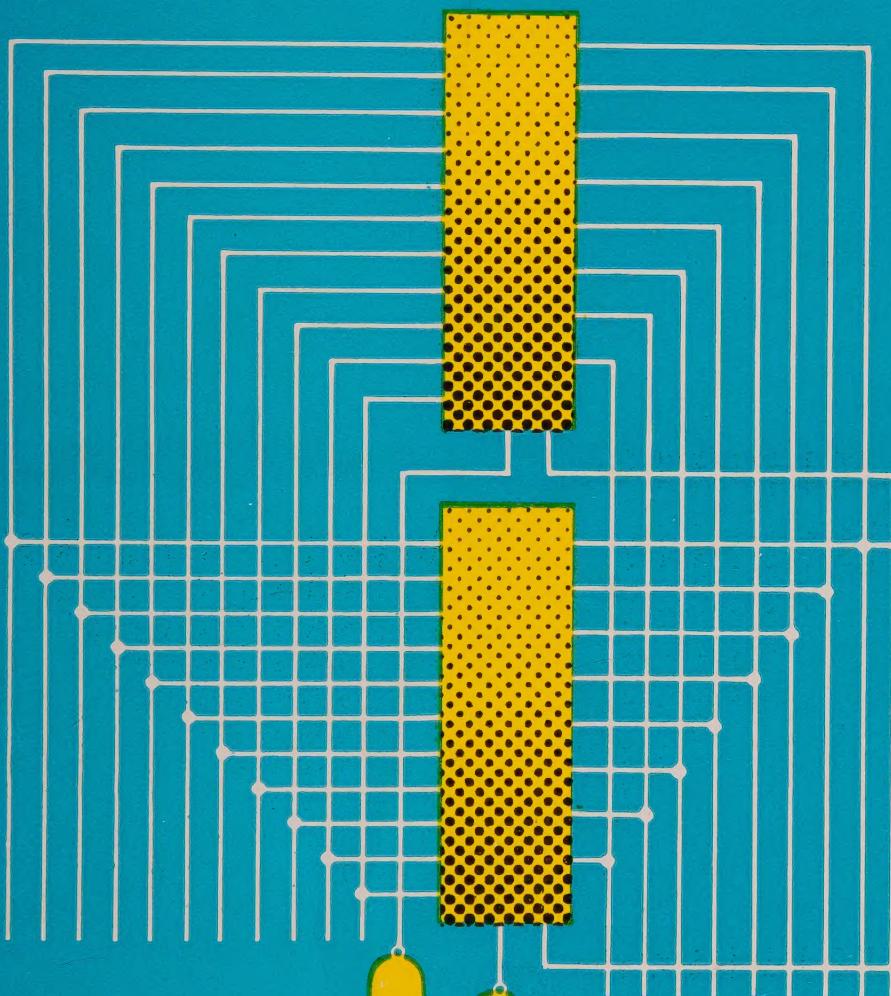


Microcomputer Systems: The 8086/8088 Family

Architecture, Programming, and Design

SECOND EDITION



Yu-cheng Liu
Glenn A. Gibson



Digitized by the Internet Archive
in 2022 with funding from
Kahle/Austin Foundation

<https://archive.org/details/microcomputersys0000liuy>

MICROCOMPUTER SYSTEMS: THE 8086/8088 FAMILY

Architecture, Programming,
and Design, Second Edition

YU-CHENG LIU
GLENN A. GIBSON

both of
Electrical Engineering Department
The University of Texas at El Paso

Prentice-Hall of India Private Limited
New Delhi-110001
1996

This Twelfth Indian Reprint—Rs. 175.00
(Original U.S. Edition—Rs. 2567.00)

**MICROCOMPUTER SYSTEMS: THE 8086/8088 FAMILY:
Architecture, Programming, and Design, 2nd Ed.**
by Yu-Cheng Liu and Glenn A. Gibson

PRENTICE-HALL INTERNATIONAL, INC., Englewood Cliffs.
PRENTICE-HALL INTERNATIONAL (UK) LIMITED, London.
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, Sydney.
PRENTICE-HALL CANADA, INC., Toronto.
PRENTICE-HALL HISPANOAMERICANA, S.A., Mexico.
PRENTICE-HALL OF JAPAN, INC., Tokyo.
SIMON & SCHUSTER ASIA PTE. LTD., Singapore.
EDITORIA PRENTICE-HALL DO BRASIL, LTDA., Rio de Janeiro.

© 1986 by Prentice-Hall, Inc., Englewood Cliffs, N.J., U.S.A. All rights reserved. No part of this book may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publishers.

ISBN-81-203-0409-8

The export rights of this book are vested solely with the publisher.

This Eastern Economy Edition is the authorized, complete and unabridged photo-offset reproduction of the latest American edition specially published and priced for sale only in Bangladesh, Burma, Cambodia, China, Fiji, Hong Kong, India, Indonesia, Laos, Malaysia, Nepal, Pakistan, Philippines, Singapore, South Korea, Sri Lanka, Taiwan, Thailand, and Vietnam.

Reprinted in India by special arrangement with Prentice-Hall, Inc., Englewood Cliffs, N.J., U.S.A.

Twelfth Printing (Second Edition) June, 1996

Published by Prentice-Hall of India Private Limited, M-97, Connaught Circus, New Delhi-110001 and Printed by Mohan Makijani at Rekha Printers Private Limited, New Delhi-110020.

Contents

Preface vii

Introduction 1

- 1-1 Overview of Microcomputer Systems 2
 - 1-1-1 *Hardware* 2
 - 1-1-2 *Software* 4
- 1-2 Data Representation 6
 - 1-2-1 *Binary Format* 6
 - 1-2-2 *BCD Format* 11
 - 1-2-3 *Alphanumeric Codes* 11
- 1-3 Addresses 14
- 1-4 General Operation of a Computer 16
- 1-5 Microprocessors in Digital System Design 20

8086 Architecture 25

- 2-1 CPU Architecture 26
- 2-2 Internal Operation 33
- 2-3 Machine Language Instructions 35
 - 2-3-1 *Addressing Modes* 35
 - 2-3-2 *Instruction Formats* 39
- 2-4 Instruction Execution Timing 47
- 2-5 The 8088 50

3 Assembler Language Programming 53

- 3-1 Assembler Instruction Format 55
- 3-2 Data Transfer Instructions 59
- 3-3 Arithmetic Instructions 63
 - 3-3-1 *Binary Arithmetic* 65
 - 3-3-2 *Packed BCD Arithmetic* 73
 - 3-3-3 *Unpacked BCD Arithmetic* 75
- 3-4 Branch Instructions 78
 - 3-4-1 *Conditional Branch Instructions* 79
 - 3-4-2 *Unconditional Branch Instructions* 82
- 3-5 Loop Instructions 86
- 3-6 NOP and HLT Instructions 90
- 3-7 Flag Manipulation Instructions 92
- 3-8 Logical Instructions 93
- 3-9 Shift and Rotate Instructions 97
- 3-10 Directives and Operators 100
 - 3-10-1 *Data Definition and Storage Allocation* 101
 - 3-10-2 *Structures* 107
 - 3-10-3 *Records* 111
 - 3-10-4 *Assigning Names to Expressions* 112
 - 3-10-5 *Segment Definition* 113
 - 3-10-6 *Program Termination* 116
 - 3-10-7 *Alignment Directives* 117
 - 3-10-8 *Value-Returning Attribute Operators* 118
- 3-11 Assembly Process 120
- 3-12 Translation of Assembler Instructions 128

4 Modular Programming 141

- 4-1 Linking and Relocation 143
 - 4-1-1 *Segment Combination* 145
 - 4-1-2 *Access to External Identifiers* 147
- 4-2 Stacks 151
- 4-3 Procedures 155
 - 4-3-1 *Calls, Returns, and Procedure Definitions* 157
 - 4-3-2 *Saving and Restoring Registers* 161
 - 4-3-3 *Procedure Communication* 162
 - 4-3-4 *Recursive Procedures* 165
- 4-4 Interrupts and Interrupt Routines 169
- 4-5 Macros 174
 - 4-5-1 *ASM-86 Macro Facilities* 174
 - 4-5-2 *Local Labels* 176
 - 4-5-3 *Nested Macros* 177
 - 4-5-4 *Controlled Expansion and Other Functions* 179
- 4-6 Program Design 183
- 4-7 Program Design Example 191

5 Byte and String Manipulation 207

- 5-1 String Instructions 208
- 5-2 REP Prefix 212

- 5-3 Text Editor Example 215
- 5-4 Table Translation 220
- 5-5 Number Format Conversions 223

6

I/O Programming 229

- 6-1 Fundamental I/O Considerations 231
- 6-2 Programmed I/O 236
- 6-3 Interrupt I/O 240
- 6-4 Block Transfers and DMA 251
- 6-5 I/O Design Example 258

7

Introduction to Multiprogramming 272

- 7-1 Process Management and iRMX 86 274
- 7-2 Semaphore Operations 282
- 7-3 Common Procedure Sharing 287
- 7-4 Memory Management 291
- 7-5 Virtual Memory and the 80286 297

8

System Bus Structure 308

- 8-1 Basic 8086/8088 Configurations 310
 - 8-1-1 *Minimum Mode* 314
 - 8-1-2 *Maximum Mode* 319
- 8-2 System Bus Timing 324
- 8-3 Interrupt Priority Management 329
 - 8-3-1 *Interrupt System Based on a Single 8259A* 329
 - 8-3-2 *Interrupt System Based on Multiple 8259As* 337
- 8-4 Bus Standards 339

9

I/O Interfaces 346

- 9-1 Serial Communication Interfaces 349
 - 9-1-1 *Asynchronous Communication* 352
 - 9-1-2 *Synchronous Communication* 354
 - 9-1-3 *Physical Communication Standards* 354
 - 9-1-4 *8251A Programmable Communication Interface* 361
- 9-2 Parallel Communication 369
 - 9-2-1 *8255A Programmable Peripheral Interface* 371
 - 9-2-2 *A/D and D/A Example* 374
- 9-3 Programmable Timers and Event Counters 378
 - 9-3-1 *Intel's 8254 Programmable Interval Timer* 380
 - 9-3-2 *Interval Timer Application to A/D* 383
- 9-4 Keyboard and Display 383
 - 9-4-1 *Keyboard Design* 385
 - 9-4-2 *Display Design* 385
 - 9-4-3 *Keyboard/Display Controller* 387
- 9-5 DMA Controllers 395
- 9-6 Diskette Controllers 402
- 9-7 Maximum Mode and 16-Bit Bus Interface Design. 413

10

- Semiconductor Memory 423
10-1 General Memory Organization 424
10-2 Static RAM Devices 427
10-3 Dynamic RAM Devices 434
10-4 Backup Power for Semiconductor Memories 442
10-5 ROM Devices 444

11

- Multiprocessor Configurations 450
11-1 Queue Status and the LOCK Facility 453
11-2 8086/8088-Based Multiprocessing Systems 456
 11-2-1 *Coprocessor Configurations* 456
 11-2-2 *Closely Coupled Configurations* 460
 11-2-3 *Loosely Coupled Configurations* 463
 11-2-4 *Microcomputer Networks* 477
11-3 The 8087 Numeric Data Processor 477
 11-3-1 *NDP's Data Types* 478
 11-3-2 *Processor Architecture* 482
 11-3-3 *Instruction Set* 485
 11-3-4 *An Example* 493
11-4 The 8089 I/O Processor 496
 11-4-1 *IOP Architecture* 499
 11-4-2 *Communication between CPU and IOP* 502
 11-4-3 *Instruction Set* 510
 11-4-4 *Examples* 515

12

- VLSI Processing and Supporting Devices 519
12-1 The 80130 520
12-2 The 80186 524
 12-2-1 *Instruction Set* 525
 12-2-2 *Internal Architecture* 531
 12-2-3 *Bus Structure* 544

13

- The 80286/80287 551
13-1 Single-level Tasks 555
13-2 Multilevel Tasks 564
13-3 Multiple Tasks 572
13-4 Interrupt System 583
13-5 System and Task Initialization 590
13-6 Interfacing to the 80286 595
13-7 The 80287 602

Appendix: 8086/8088 Instruction Set Summary 605**Index 613**

Preface

This book is designed as a text for a one-semester upper-level college course; however, it could also be used by practicing engineers and technicians. It is assumed that the reader is familiar with at least one high-level language, such as FORTRAN, BASIC, or Pascal, and has a knowledge of elementary logic. Chapter 1 provides a review of general computer concepts; those who are more familiar with computers may elect to scan it rather quickly. There are numerous exercises at the end of each chapter that reinforce the important points of the chapter. These exercises are ordered according to the material in the chapter and are intended to help the reader verify his or her understanding of what has been presented. Some of them are also designed to extend the reader's knowledge in selected areas.

The purpose of the book is to give an in-depth study of both the hardware and software included in microcomputer systems. Although the concepts considered are general in nature, the discussion is based on a particular microprocessor, the Intel 8086/8088, and its associated supporting devices and software. The software material is centered on Intel's ASM-86 assembler; however, not all of the details of this assembler are presented because it was felt that their inclusion would detract from the important points.

It was decided to aim the book at application areas that are above the simple controller area, which is covered by the 8-bit processors; therefore, a 16-bit processor was selected. Of the more popular 16-bit single-chip processors, the 8086/8088 was chosen because it provides a varied instruction set that includes extensive arithmetic operations and string manipulations and has several advanced architectural features designed to support multiprogramming and multiprocessing. Also, Intel has developed associate devices, such as the 8087 numeric data processor and the 8089 I/O processor, that allow certain important multiprocessing functions to

be homogeneously integrated into a system. By examining the 8086 microprocessor, multiprogramming and multiprocessing can be discussed in a unified manner. Other important topics not significantly studied in texts on 8-bit microprocessors, but included here, are those related to system software, string-handling facilities, and modular programming. Consideration of such areas becomes more important as the applications increase in complexity.

After the review of computer systems and the background of microprocessors are given in Chap. 1, the 8086 is introduced by describing its internal architecture and machine language instructions in Chap. 2. Although the 8086 and 8088 execute the same instruction set, there are minor differences in their internal architectures, and these differences are mentioned in the last section of Chap. 2. Chapter 3 covers the fundamentals of assembler language programming. This chapter first introduces the reader to assembler language concepts and then examines the basic 8086 instructions, including the arithmetic, logic, and control transfer instructions. Numerous examples are included to enhance the reader's understanding of the applications of the various instructions. The directives required in single-module programs are also discussed, and the chapter concludes with an explanation of the assembly process. Chapter 4 extends the discussion of assembler language programming to multiple-module program structures and considers advanced programming techniques. It contains a section on program design that examines the development process for properly designing multiple-module programs.

The 8086's facilities for string manipulation are given in Chap. 5. This material also discusses the code conversions needed to write efficient I/O routines. Chapter 6 introduces the reader to I/O programming techniques, including programmed I/O, interrupt-driven I/O, and direct memory access. The interrupt sequence and the use of interrupt vectors are explained through examples. An introduction to buffering is given to provide the reader with an understanding of how I/O is handled by a real-time operating system. Chapter 7 is an introduction to multiprogramming. Although a lengthy discussion of multiprogramming is not practical in a book of this type, this chapter does briefly consider the more important aspects of multiprogramming and how they relate to the design of microcomputer systems. This discussion includes such topics as data sharing, process synchronization, reentrant code, and memory management.

The remainder of the book concentrates on the hardware aspects of microcomputer systems. Chapter 8 considers the interface between the system bus and the microprocessor, interrupt management, and bus activity and timing. Chapter 9 is concerned with the interfaces that join I/O and mass storage devices with the system bus. It considers serial and parallel interfaces, direct memory access controllers, timers, and mass storage controllers. Memory subsystems are discussed in Chap. 10, and Chap. 11 examines the multiprocessing configurations that are compatible with Intel devices. It is at this point that the 8087 numeric data processor and 8089 I/O processor are considered. The last chapter is a survey of Intel's more advanced VLSI devices and includes brief discussions of the 80130, 80186, and 80286. The appendix contains a summary of the 8086/8088 instruction set.

The authors would like to recognize the cooperation they have received from the Electrical Engineering Department at the University of Texas at El Paso and to thank the many people who have helped produce this book. In particular they would like to thank Rich Bruns of Intel Corporation for supplying much of the needed information and for reviewing the book, and Mary Lou Gibson for typing and proofreading the manuscript and solving many of the exercises.

*Yu-cheng Liu
Glenn A. Gibson*

Since the first edition of this book was published, the 80186 and 80286 extensions of the 8086 family have become widely used. These processors are enhanced versions of the 8086, with the 80186 having several on-chip peripheral support hardware features and the 80286 having memory management and multi-tasking capabilities. Due to their significance, Chapter 12 has been expanded to include many of the details of the 80186, and the brief material on the 80286 has been replaced with a lengthy Chapter 13. Although it is impossible to cover everything on the 80286 in one chapter, Chapter 13 examines all of the basics of the 80286's protected virtual address mode and describes how it can be interfaced to both system and local buses and to an 80287. Instructors can now also obtain a solutions manual from Prentice-Hall.

Also by Glenn A. Gibson and Yu-cheng Liu . . .

MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS

Designed for scientists and engineers who wish to acquire an overall knowledge of microcomputers and hardware designers who need to understand microcomputer programming.

1980/479 pp

Also by Glenn A. Gibson and James R. Young . . .

INTRODUCTION TO PROGRAMMING USING FORTRAN 77

An introductory programming text that is directed to a wide audience; however, most of the exercises are engineering/science oriented. It uses the spiral approach and includes material on program design, testing, and debugging.

1982/461 pp

1

Introduction

The applications for microelectronics can be divided into two main categories, control and data processing. Uses of microelectronics in simple controllers are called *low-end applications*, and uses in very complicated controllers, such as robots, avionics, and sophisticated military hardware, and in general purpose data processing are referred to as *high-end applications*. Because it is not cost-effective to use a complex circuit to meet a low-end application, bigger is not always better. However, although improvements can still be made at the low-end of the microelectronic spectrum, the thrust of new technology is at the high end, where unthought-of applications are to be found and satisfied. It is at the high end where demand for smaller and more complicated circuits is never-ending.

Microelectronics devices are labeled according to complexity as follows:

SSI—Fewer than 10 gates.

MSI—Between 10 and 100 gates.

LSI—Between 100 and a few thousand gates.

VLSI—Several thousand gates.

The current microelectronics frontier is in the VLSI range, with the heart of the current technology being centered on the single-chip 16-bit processors and their associated devices. Presently, the three leaders in the 16-bit processor area are the Zilog Z8000, the Motorola M68000, and the Intel 8086. This book is concerned primarily with the Intel 8086 and its associated devices, but much of the discussion applies to other 16-bit microprocessors as well.

The purpose of this chapter is to provide several fundamental definitions and other background information, much of which is considered a review. The first

section gives an overview of computer systems and the next three sections discuss the ways in which data are stored inside a computer, how information within the computer is accessed, and the general operation of a computer as it executes a program. Section 1-5 examines the basic digital system design criteria and considers the evolution of microprocessors by briefly looking at the development of the Intel family of microprocessors.

1-1 OVERVIEW OF MICROCOMPUTER SYSTEMS

A microcomputer system, just as any other computer system, includes two principal components, hardware and software. The *hardware* is, of course, the circuitry, cabinetry, etc., and the *software* is the collection of programs which direct the computer while it performs its tasks.

1-1-1 Hardware

The *architecture* of a computer is the general layout of its major components, their principal features, and how they are connected together. Figure 1-1 depicts the overall architecture of a typical microcomputer system. The components shown in the figure are the central processing unit (CPU), timing circuitry, memory, input/output (I/O) subsystem, bus control logic, and system bus. In a microcomputer the CPU is a microprocessor and is often referred to as the *microprocessor unit* (MPU). Its purpose is to decode the instructions and use them to control the activity within the system. It also performs all arithmetic and logical computations. The timing circuitry, or *clock*, generates one or more trains of evenly spaced pulses and is needed to synchronize the activity within the microprocessor and the bus control logic. The pulse trains output by the timing circuitry normally have the same frequency but are offset in time, i.e., are different phases. Microprocessors require from one to four phases, with the earlier processors requiring more than one phase. For many of the recent microprocessors, the timing circuitry, except for the oscillator, is included in the same integrated circuit (IC) as the processing circuitry.

The memory is used to store both the data and instructions that are currently being used. It is normally broken into several modules, with each module containing several thousand locations. Each location may contain part or all of a datum or instruction and is associated with an identifier called a *memory address* (or, simply, an *address*). The CPU does its work by successively inputting, or *fetching*, instructions from memory and carrying out the tasks dictated by them.

The I/O subsystem may consist of a variety of devices for communicating with the external world and for storing large quantities of information. Card readers, paper tape readers, and analog-to-digital (A/D) converters are examples of input equipment, and line printers, plotters, card punches, paper tape punches, and digital-to-analog (D/A) converters are output devices. Some devices, such as terminals, provide both input and output capabilities. Computer components for permanently storing programs and data are referred to as *mass storage units*. The

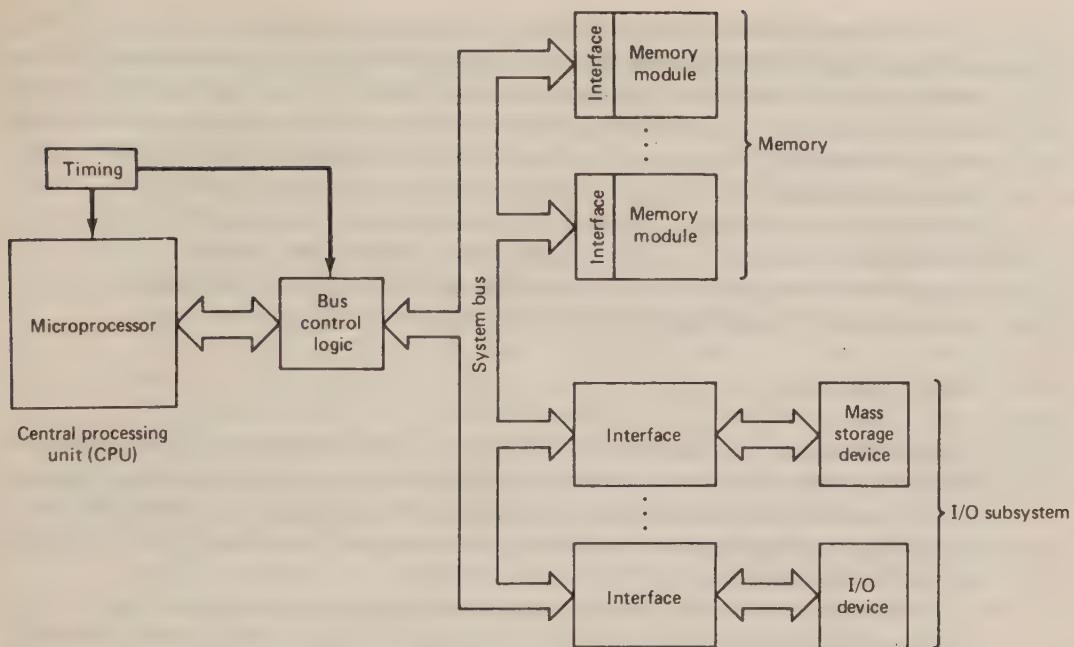


Figure 1-1 Architecture of a typical microprocessor.

more popular types of mass storage equipment are magnetic tape and disk units, but recent technology has made available magnetic bubble memories (MBMs) and charge-coupled devices (CCDs). Although a mass storage unit may be used to store programs as well as data, programs must be transferred to memory before they are executed.

The *system bus* is a set of conductors that connects the CPU to its memory and I/O devices. It is over these conductors, which may be wires in a cable or lines on a printed circuit (PC) board, that all information must travel. Exactly how information is transmitted over the bus is determined by the bus's specifications. Normally, the bus conductors are separated into three groups:

1. The data lines for transmitting the information.
2. The address lines, which indicate where the information is to come from or is to be placed.
3. The control lines, which regulate the activity on the bus.

The signals on the bus must be coordinated with the signals created by the various components connected to the bus. The circuitry needed to connect the bus to a device is called an *interface*, and the *bus control logic* is the interface to the CPU. Most manufacturers provide a variety of IC devices for facilitating the design of interfaces and the bus control logic. Depending on the complexity of the system, the bus control logic may be partially or totally contained in the CPU IC.

Memory interfaces consist primarily of the logic needed to decode the address of the memory location being accessed and to buffer the data onto or off of the bus, and of the circuitry to perform memory reads or writes. I/O interfaces may be quite simple or very complex. All I/O interfaces must be capable of buffering data onto and/or off of the system bus, receiving commands from the CPU, and transmitting status information from their associated devices to the CPU. In addition, those connected to mass storage units must be capable of communicating directly with memory and this requires them to have the ability to control the system bus. The communication between an I/O interface and the data bus is accomplished through registers which are referred to as *I/O ports*.

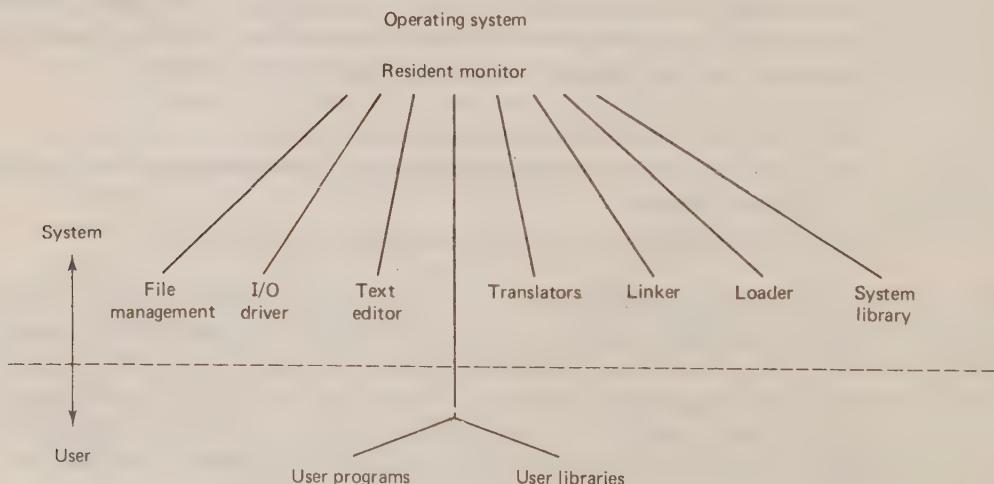
1-1-2 Software

Computer software is generally divided into two broad categories, system software and user software. *System software* is the collection of programs which are needed in the creation, preparation, and execution of other programs. *User software* consists of those programs generated by the various users of the system in their attempts to apply the computer to solving their problems.

How much system software accompanies a particular computer system depends on the application. A computer whose function is static throughout its lifetime (such as a controller used to perform a repetitive task) may execute only one program. On the other hand, a computer that is used for general purpose data processing may require an extensive array of system programs. For the most part, this book will assume the general purpose setting. In such a setting programs normally, but not always, fit into a well-defined hierarchy similar to the one shown in Fig. 1-2.

The *operating system* is a collection of system programs that provide an interface between the user and the machine and enable the machine to be used

Figure 1-2 Software hierarchy.



efficiently. The term "operating system" is not precisely defined and exactly which system programs are included in the operating system depends on the manual or book being referenced. The most important part of the operating system is the *resident monitor*. It is the part of the operating system that is in the computer's memory at all times while the computer is turned on. The resident monitor must be capable of receiving commands from the users and initiating the corresponding actions to be performed by the operating system.

The operating system also includes programs, called *I/O drivers* and *file management routines*, for performing I/O operations and for handling large collections of data that are stored on mass storage devices. Whenever a user program or other system program needs to use an I/O device, it does not normally carry out the operation itself, but instead it requests the operating system to use an I/O driver to perform the task. This gives the operating system better control of the computer and alleviates the need to include I/O subroutines within user programs. The file management routines are used in conjunction with the mass storage I/O drivers and are for accessing, copying, and otherwise manipulating files.

In addition, the system software may encompass a variety of high-level language translators, an assembler, a text editor, and programs for aiding in the preparation of other programs. There are three levels of programming. They are:

1. Machine language.
2. Assembler language.
3. High-level language.

Machine language programs are programs that the computer can understand and execute directly. Assembler language instructions match machine language instructions on a more or less one-for-one basis, but are written using character strings so that they are more easily understood, and high-level language instructions are much closer to the English language and are structured so that they naturally correspond to the way programmers think. Ultimately, an assembler language or high-level language program must be converted into machine language by programs called *translators*. If the program being translated is in assembler language, the translator is referred to as an *assembler*, and if it is in a high-level language the translator is referred to as a *compiler or interpreter*.

A *text editor* is a program for inputting or modifying text (letters, numbers, punctuation marks, etc.) that is to be stored or is stored on a mass storage device. The text may constitute an assembler or high-level language program, a set of data, or a report. (This book was written using a text editor.) Although text editors may be used for several purposes, our interest in them is due to their being used to create programs.

There are two other system programs that are required in preparing programs for execution. It often occurs that the same task must be performed by a variety of programs. Therefore, most operating systems permit the creation of collections of subroutines, called *libraries*, that can be attached to any system or user program. There is normally a system library for general use and, perhaps, the users may

create their own libraries. The preparation program used for joining the program to be executed to library and other previously translated subroutines is referred to as a *linker* or *linkage editor*. The other preparation program is for transferring the program to be executed into memory and is called a *loader*. Sometimes the linking and loading functions are combined into a single program. The complete process for creating and executing programs is discussed in detail in Chap. 3.

1-2 DATA REPRESENTATION

In order to make computers more reliable and easier to build, they are based on devices that can take on only two states, one of which is denoted by 0 and the other by 1. By concatenating several 0s and 1s together, 0-1 combinations can be formed to represent as many different entities as desired. A combination containing a single 0 or 1 is called a *bit*. In general, n bits can be used to distinguish among 2^n distinct entities and each addition of a bit doubles the number of possible combinations.

Computers use strings of bits to represent numbers, letters, punctuation marks, and any other useful pieces of information. Numbers are associated with bit combinations according to *number formats*. The three basic number formats are:

1. Binary (or integer).
2. Floating point (or real).
3. Binary coded decimal (BCD) (or decimal).

(The integer and floating point formats correspond to the integer and real number types used in FORTRAN and other high-level languages.)

An *alphanumeric code* is the assignment of letters and other characters to bit combinations. Because assignments of bit combinations to the decimal digits are also included in alphanumeric codes, such codes can also be used to store and manipulate numbers. (In high-level languages, character strings are represented by alphanumeric codes.) Except for the floating point format, which is considered in Chap. 11, we will now consider number formats and alphanumeric codes in detail.

1-2-1 Binary Format

Numbers are abstract entities that may be denoted using a variety of rules and patterns. Nonnegative integers are normally represented by choosing a number x , called the *base*, and x different symbols, called *digits*, and then using a string of digits such as

$$a_n a_{n-1} \dots a_1 a_0$$

to indicate the number

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

For example, if the base is 10, then 65,308 represents

$$6 \times 10^4 + 5 \times 10^3 + 3 \times 10^2 + 0 \times 10 + 8$$

Although a base of 10 is what is used in our everyday work, the base could be any integer greater than 1. Because computers are made from two-state devices, they use a base of 2, in which case 10110 denotes the number

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2 + 0$$

Bases of 8 and 16 are also commonly encountered when working with computers. The number systems corresponding to the bases of 2, 8, 10, and 16 are called the *binary*, *octal*, *decimal*, and *hexadecimal number systems*, respectively. The symbols for denoting the digits in these number systems are summarized in Fig. 1-3. If it is not clear from the context which number system is intended, then a number is subscripted with its base, e.g., 1110_2 is fourteen and 1110_{10} is one thousand one hundred and ten.

Converting a number in base x to a number in base 10 is a matter of finding the d_i 's in the equation

$$a_n x^n + \cdots + a_1 x + a_0 = d_m \times 10^m + \cdots + d_1 \times 10 + d_0$$

Figure 1-3 Important number systems. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

| Binary (Base 2) | Octal (Base 8) | Decimal (Base 10) | Hexadecimal (Base 16) |
|--------------------|-------------------|----------------------|--------------------------|
| 0 | 0 (000) | 0 (0000) | 0 (0000) |
| 1 | 1 (001) | 1 (0001) | 1 (0001) |
| | 2 (010) | 2 (0010) | 2 (0010) |
| | 3 (011) | 3 (0011) | 3 (0011) |
| | 4 (100) | 4 (0100) | 4 (0100) |
| | 5 (101) | 5 (0101) | 5 (0101) |
| | 6 (110) | 6 (0110) | 6 (0110) |
| | 7 (111) | 7 (0111) | 7 (0111) |
| | | 8 (1000) | 8 (1000) |
| | | 9 (1001) | 9 (1001) |
| | | | A (1010) |
| | | | B (1011) |
| | | | C (1100) |
| | | | D (1101) |
| | | | E (1110) |
| | | | F (1111) |

Note: Binary equivalents are given in parentheses.

from the given a_i s. This can easily be done by expressing x and the a_i s as decimal numbers and then carrying out the necessary decimal arithmetic. For example,

$$\begin{aligned}10111011_2 &= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 \\&\quad + 1 \times 2 + 1 \\&= 128 + 0 + 32 + 16 + 8 + 0 + 2 + 1 \\&= 187_{10}\end{aligned}$$

and

$$51A_{16} = 5 \times 16^2 + 1 \times 16 + 10 = 1306_{10}$$

These conversions can also be made by using *Horner's rule*, which states that

$$a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = ((\cdots((a_nx + a_{n-1})x + \cdots)x + a_1)x + a_0$$

By applying Horner's rule and successive divisions by x , a base 10 number can be converted to a base x number as indicated by the following examples:

$$\begin{array}{r} 38 \\ 2 \underline{) 38} \\ 19 \\ 2 \underline{) 19} \\ 9 \\ 2 \underline{) 9} \\ 4 \\ 2 \underline{) 4} \\ 2 \\ 2 \underline{) 2} \\ 1 \\ 2 \underline{) 1} \\ 0 \end{array} \quad \begin{array}{r} 587 \\ 16 \underline{) 587} \\ 36 \\ 16 \underline{) 36} \\ 2 \\ 16 \underline{) 2} \\ 0 \end{array}$$

Remainders

$$100110_2 = 38_{10}$$

$$24B_{16} = 587_{10}$$

Conversion of a binary number to a hexadecimal number is simply a matter of putting the binary digits in groups of four and assigning each group to its hexadecimal equivalent, e.g.,

$$\begin{array}{c} 0110 \quad 1011 \quad 0111 \\ \hline 1 \quad B \quad 7 \end{array}$$

Conversely, by assigning each hexadecimal digit to its binary equivalent, conversions from hexadecimal to binary can be made as follows:

$$\begin{array}{c} A \quad 1 \quad 9 \\ / \quad | \quad \backslash \\ 1010 \quad 0001 \quad 1001 \end{array}$$

Conversions between the binary and octal number systems can be made similarly, except the grouping is by threes instead of fours.

Although computers work only with binary numbers, manuals and books about computers frequently use the binary/octal or binary/hexadecimal grouping relationships to express the numbers in octal or hexadecimal. The reason is that this allows the binary numbers to be written much more compactly. As an example, the 16-bit binary quantity 1011010100111010 can be written B53A in hexadecimal. This book and the 8086 manuals utilize hexadecimal numbers for this purpose.

The arithmetic operations can be performed in any number system using the same algorithms that are applied in the decimal system. However, because of the unfamiliarity of the addition and multiplication tables for octal and hexadecimal numbers, it is difficult to perform the arithmetic operations on them directly. One may convert the operands to decimal numbers, carry out the operations, and convert the result back to the original number system. Examples of the arithmetic operations in the binary system are given in Fig. 1-4.

Negative integers could be designated by attaching an extra bit to indicate the sign. When this is done a 0 normally represents a plus and a 1 represents a minus, and the resulting format is called the *sign-magnitude format*. Seldom, however, are negative integers denoted by the sign-magnitude format. Computers almost always use the *2's complement format*, which assigns negative integers as follows:

$$-b = 2^n - b$$

where b is the magnitude of the integer and n is the number of bits used to represent it. For $n = 16$ and

$$b = 1234_{10} = 0000010011010010_2 = 04D2_{16}$$

$-b$ in 2's complement format would be

$$2^{16} - 1234_{10} = 1111101100101110_2 = FB2E_{16}$$

If n bits are used for each number, then it can be shown that the integers from

| | | | |
|---------|-----|---------|-----|
| 110101 | 53 | 101101 | 45 |
| +10010 | +18 | -100110 | -38 |
| 1000111 | 71 | 111 | 7 |

(a)

(b)

Figure 1-4 Typical binary arithmetic operations. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

| | | | |
|--------------|------------|------------|----------|
| 10110 | 22 | 10001 | 17 |
| <u>x1011</u> | <u>x11</u> | <u>110</u> | <u>6</u> |
| 10110 | 22 | 110 | 6 |
| 10110 | 22 | 00110 | 42 |
| 10110 | 22 | 110 | 42 |
| 11110010 | | 0 | 0 |

(c)

(d)

-2^{n-1} to $2^{n-1} - 1$ can be unambiguously represented. For $n = 16$, the integers from $-32,768$ to $32,767$ are written

$$100000000000000000 = 8000$$

$$100000000000000001 = 8001$$

.

.

.

$$1111111111111111 = \text{FFFF}$$

$$0000000000000000 = 0000$$

$$0000000000000001 = 0001$$

.

.

.

$$0111111111111111 = 7FFF$$

For computational ease, note that the 2's complement of a number b can be found by interchanging the 0s and 1s, which produces $(2^n - 1) - b$ (called the *1's complement* of b), and adding 1. For $n = 8$ and $b = 46$:

$$2^8 - 1 = 11111111$$

$$b = \underline{00101110}$$

$$\begin{array}{r} (2^8 - 1) - b = 11010001 \\ + 1 \\ \hline 2^8 - b = 11010010 \end{array} \quad \begin{array}{l} \text{1's complement} \\ + 1 \\ \hline \text{2's complement} \end{array}$$

It can also be shown that when addition of signed integers in the 2's complement format is performed in the usual manner, the result will be correctly expressed in 2's complement provided only that the least significant n bits are retained. Subtraction can be performed by adding the 2's complement of the subtrahend. For example, for $n = 8$,

$$\begin{array}{r} 72 = 01001000 \\ - 35 = -00100011 \\ \hline 37 \end{array} \quad \begin{array}{r} 01001000 \\ + 11011101 \\ \hline 100100101 = 37 \end{array}$$

↑
Discarded

and

$$\begin{array}{r} -29 = 11100011 \\ - (-90) = -10100110 \\ \hline 61 \end{array} \quad \begin{array}{r} 11100011 \\ + 01011010 \\ \hline 100111101 = 61 \end{array}$$

↑
Discarded

1-2-2 BCD Format

A *BCD format* is one in which the decimal digits are stored in terms of their 4-bit binary equivalents, i.e., binary coded decimal. There are two basic formats of this type, the packed and unpacked BCD formats. In the packed BCD format a string of decimal digits is stored in a sequence of 4-bit groups. The number 9502 would be stored as

1001 0101 0000 0010

In the unpacked BCD format each digit would be stored in the low-order part of an 8-bit group and what is put into the high-order part is unimportant. In this format 9502 would be stored as

uuuu1001 uuuu0101 uuuu0000 uuuu0010

Unlike the 2's complement, which is used to indicate negative integers in the binary format, unless the system includes special hardware the sign convention to be employed with the BCD format is determined by the programmer (but once a convention is chosen it must be used consistently). Negative integers could be stored using either a sign-magnitude format or the 10's complement. If the sign-magnitude arrangement is selected, the plus and minus signs would be represented by two of the unused 4-bit combinations, e.g., 1100 could indicate a plus and 1101 a minus. The sign may be placed before or after the string of digits.

The *n*-digit 10's complement of an arbitrary integer *d* is defined to be $10^n - d$. For a given *n*, the integers that can be unambiguously defined in 10's complement are from $-5 \times 10^{n-1}$ to $5 \times 10^{n-1} - 1$. For *n* = 8, 32 bits would be required to store an integer in the range -50,000,000 to 49,999,999 if the packed BCD format is assumed. The 10's complement format has the same properties as the 2's complement format and addition will produce the correct 10's complement result, e.g., for *n* = 4

$$\begin{array}{r} 252 = 0252 \\ + (-485) = 9515 \\ \hline -233 = 9767 \end{array}$$

Subtraction is performed by adding the negative of the subtrahend to the minuend.

1-2-3 Alphanumeric Codes

The Extended Binary Coded Decimal Interchange Code (EBCDIC), used by IBM, and the American Standard Code for Information Interchange (ASCII), used by almost all other computer manufacturers, are the two dominant alphanumeric codes. The ASCII code assignments are listed in Fig. 1-5, which gives the hexadecimal equivalents of the assigned bit combinations. Alphanumeric codes are the primary means of performing I/O with the external world. As illustrated in Fig. 1-6, when a key is depressed on a terminal the corresponding ASCII code is generated and sent to the computer. Conversely, if the computer sends a terminal an ASCII-coded string of bits, the terminal must decode the bits and respond accordingly.

Note that not all ASCII characters are printed; some instruct the terminal to space, backspace, line feed, return carriage, and so on. In addition to the printing and control characters, the ASCII code includes several characters, such as end-of-file (EOF) and end-of-transmission (EOT), which serve as markers when transmitting and storing data.

As an example, the character string

DOE,
JOHN P.-50

Figure 1-5 ASCII code. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

| ASCII Char. | Hex. Code | Control Character | ASCII Char. | Hex. Code | Control Character | ASCII Char. | Hex. Code | Control Character |
|-------------|-----------|--------------------|-------------|-----------|-------------------|-------------|-----------|-------------------|
| NUL | 00 | Null | + | 2B | | V | 56 | |
| SOH | 01 | Start heading | , | 2C | | W | 57 | |
| STX | 02 | Start text | - | 2D | | X | 58 | |
| ETX | 03 | End text | . | 2E | | Y | 59 | |
| EOT | 04 | End transmission | / | 2F | | Z | 5A | |
| ENQ | 05 | Inquiry | 0 | 30 | | [| 5B | |
| ACK | 06 | Acknowledgment | 1 | 31 | | \ | 5C | |
| BEL | 07 | Bell | 2 | 32 | |] | 5D | |
| BS | 08 | Backspace | 3 | 33 | | ^ | 5E | |
| HT | 09 | Horizontal tab | 4 | 34 | | _ | 5F | |
| LF | 0A | Line feed | 5 | 35 | | | 60 | |
| VT | 0B | Vertical tab | 6 | 36 | | a | 61 | |
| FF | 0C | Form feed | 7 | 37 | | b | 62 | |
| CR | 0D | Carriage return | 8 | 38 | | c | 63 | |
| SO | 0E | Shift out | 9 | 39 | | d | 64 | |
| SI | 0F | Shift in | : | 3A | | e | 65 | |
| DLE | 10 | Data link escape | : | 3B | | f | 66 | |
| DC1 | 11 | Device control 1 | < | 3C | | g | 67 | |
| DC2 | 12 | Device control 2 | = | 3D | | h | 68 | |
| DC3 | 13 | Device control 3 | > | 3E | | i | 69 | |
| DC4 | 14 | Device control 4 | ? | 3F | | j | 6A | |
| NAK | 15 | Neg. acknowledge | @ | 40 | | k | 6B | |
| SYN | 16 | Synchronous/Idle | A | 41 | | l | 6C | |
| ETB | 17 | End trans. block | B | 42 | | m | 6D | |
| CAN | 18 | Cancel data | C | 43 | | n | 6E | |
| EM | 19 | End of medium | D | 44 | | o | 6F | |
| SUB | 1A | Start special seq. | E | 45 | | p | 70 | |
| ESC | 1B | Escape | F | 46 | | q | 71 | |
| FS | 1C | File separator | G | 47 | | r | 72 | |
| GS | 1D | Group separator | H | 48 | | s | 73 | |
| RS | 1E | Record separator | I | 49 | | t | 74 | |
| US | 1F | Unit separator | J | 4A | | u | 75 | |
| SP | 20 | Space | K | 4B | | v | 76 | |
| ! | 21 | | L | 4C | | w | 77 | |
| " | 22 | | M | 4D | | x | 78 | |
| # | 23 | | N | 4E | | y | 79 | |
| \$ | 24 | | O | 4F | | z | 7A | |
| % | 25 | | P | 50 | | { | 7B | |
| & | 26 | | Q | 51 | | | 7C | |
| , | 27 | | R | 52 | | } | 7D | |
| (| 28 | | S | 53 | | ~ | 7E | |
|) | 29 | | T | 54 | | DEL | 7F | Delete-rubout |
| * | 2A | | U | 55 | | | | |

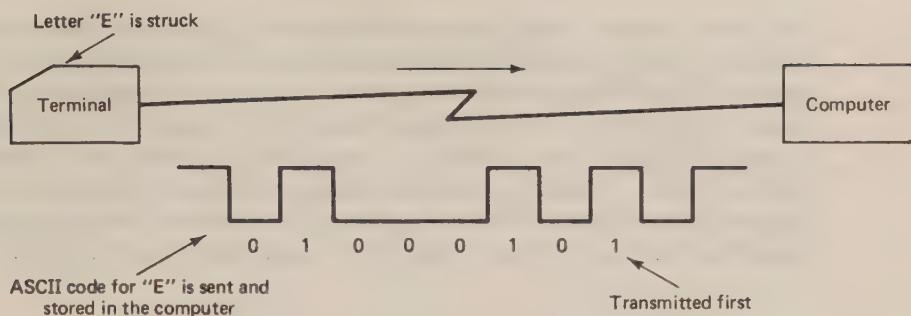


Figure 1-6 ASCII character being transmitted to a computer.

would correspond to the following string of bit combinations (given in hexadecimal):

| Carriage return | Line feed | Space |
|-------------------|-------------|-----------------|
| ↓ | ↓ | ↓ |
| 44 4F 45 2C 0D 0A | 4A 4F 48 4E | 20 50 2E 2D |
| D O E , | J O H N | 35 30 P . - 5 0 |

In particular, note that neither the digit 0 nor the space corresponds to the all 0 combination. The bit combination consisting of all 0s is called the *null character* and causes no action. It is principally used to take up time while a mechanical terminal is performing a carriage return or line feed.

The number of bits that a code needs to represent a character is called its *length*. A code of length n permits the identification of 2^n characters. It is seen from Fig. 1-5 that the ASCII code is a 7-bit code and, hence, it includes 128 characters. In addition to the 7 code bits, a parity bit is normally attached as the most significant bit to each character, making 8 the total number of bits transmitted. It is common practice to retain the extra bit after it is received by the computer, but it is normally set to zero. Thus characters are stored internally in groups of 8 bits.

The numerical sequence of the characters in a code is called the code's *collating sequence*. It is important that the numbers representing the digits are in ascending order because this allows arithmetic on the code numbers to be used directly to determine relative magnitudes. Also, by having the numbers associated with the letters in ascending order, arithmetic can be used to put character strings in alphabetical order.

Numbers are transmitted to and from the computer as sequences of ASCII-coded digits. The number 7902 would be transmitted as

| | | | |
|----|----|----|----|
| 37 | 39 | 30 | 32 |
| 7 | 9 | 0 | 2 |

When the computer receives the number it may store the number without modification, in which case the number will be in unpacked BCD format; it may strip away the high-order bit groups and pack the low-order bit groups together, in which case the number will be in packed BCD format; or it may convert the number to

the binary format. Which of these alternatives is taken depends on the program being executed. Unpacked BCD numbers have the advantage that they do not need to be converted for I/O operations, but have the disadvantage of occupying more storage space. The unsigned integer 7902 needs 32 bits of storage for its unpacked BCD format, 16 bits for its packed BCD format, and 13 bits for its binary format. Also, it takes less computer time to perform arithmetic operations on numbers in the binary format than in a BCD format. Number conversions are considered further in Chap. 5 when code conversions are discussed.

1-3 ADDRESSES

All memory locations and I/O registers are, of course, comprised of bits, but because single bits contain very little information they are grouped together to form *bytes* and *words*. Because characters are normally 7 or 8 bits long and because computers work more naturally with powers of 2, bytes almost always consist of 8 bits. Words may consist of 2, 3, or 4 bytes, depending on the computer and its system bus structure. Since the 16-bit single-chip microprocessors have 16 data lines in their system buses, they use the term "word" to mean 2 bytes (16 bits).

Each byte has an identifying address associated with it and when a byte is to be accessed its address is transmitted to the appropriate interface via the address lines. Addresses are composed of bit combinations and the set of all possible combinations for a given situation is called an *address space*. Some computers have two address spaces while others use a single address space to access all memory locations and I/O registers. If there are separate memory and I/O address spaces, then some of the control lines must be used in conjunction with the address lines to determine which space is being accessed. Because the memory is divided into modules, some of the high-order bits in a memory address are used to select the module and the remaining (low-order) bits are used to identify the byte (or word) within the module. Similarly, an interface is identified by the high-order bits in an I/O address and the register within the interface is selected by the 2 or 3 low-order bits. The overall organization of memory and I/O registers and how they are addressed is summarized in Fig. 1-7.

The number of possible bits in an address determines the size of an address space. If an address is n bits wide, then there are 2^n possible addresses (0 through $2^n - 1$). The number of address lines in the system bus dictates the size of the memory (or possibly the combined memory and I/O) space. A total of n address lines would imply a maximum memory (or overall memory and I/O) capacity of 2^n bytes. Twenty address lines would imply a capacity of

$$2^{20} = (2^{10})^2 \approx (10^3)^2 = 1 \text{ million bytes}$$

When there are 2 bytes in a word there is some question as to which byte address is to be used to identify the word. Also, it is sometimes necessary to comment about a specific bit in a byte or word. Throughout this book (and the Intel manuals) the address of a word is the address of its low-order, or low-address,

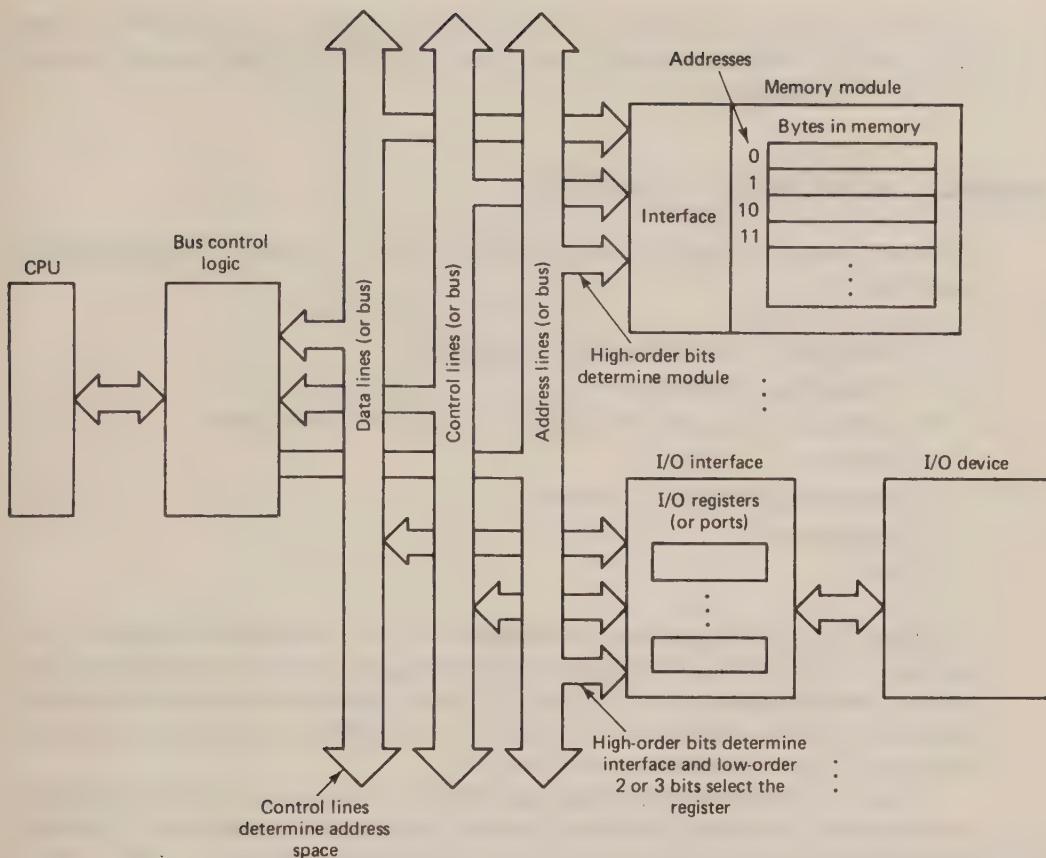
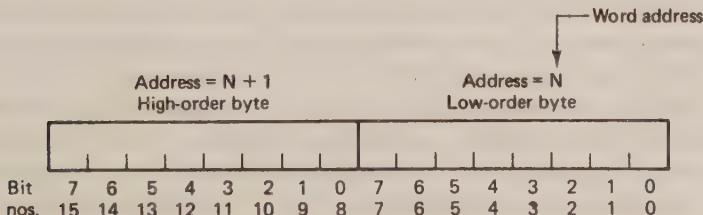


Figure 1-7 Memory and I/O register organization.

byte. The bits are numbered with 0 being assigned to the least significant bit (LSB). In a byte the most significant bit (MSB) is numbered 7 and in a word the MSB is numbered 15. These conventions are summarized in Fig. 1-8.

Some computers require that words begin with addresses that are divisible by the number of bytes in a word and produce an alignment error if this rule is not followed. Others, such as the 8086, permit words to begin at any address;

Figure 1-8 Address and bit numbering conventions.



however, accessing “nonaligned” words may require more than one memory access (e.g., on the 8086, referencing a word that begins with an odd address requires two memory accesses).

1-4 GENERAL OPERATION OF A COMPUTER

From our experience with high-level languages it is seen that a CPU must facilitate working with:

1. Assignments and arithmetic expressions.
2. Unconditional branches.
3. Conditional branches and relational and logical expressions.
4. Looping.
5. Arrays and other data structures.
6. Subroutines.
7. I/O.

A typical CPU architecture that is designed to accommodate these features is given in Fig. 1-9. It includes a control unit for decoding and carrying out the instructions, a set of working registers for helping with the addressing and computational tasks, an arithmetic/logic unit (ALU) for executing the arithmetic and logical operations, and an I/O control section for handling I/O.

Just as with high-level language programs, unless a branch instruction is encountered a machine language program is executed in sequence. A register, called the *instruction register (IR)*, is used to hold the current instruction while it is being decoded and executed, and a register, called the *program counter (PC)*, is used to hold the address of the next instruction. When the present instruction has completed its execution, the address in the PC is placed on the address bus, the memory places the next instruction on the data bus, and the CPU inputs the instruction to the IR. While this instruction is decoded its length in bytes is determined and the PC is incremented by the length so that the PC will point to the next instruction. When the execution of this instruction is completed, the contents of the PC are placed on the address bus and the cycle is repeated.

Unconditional branch instructions permit the normal sequencing to be altered by replacing the contents of the PC, the address of the next instruction, with an address determined by the branch instruction. Conditional branch instructions may or may not replace the contents of the PC depending on the results of the previous instructions, i.e., the current state of the processor as determined by the previous instructions. The current state of the processor is stored in a register called the *processor status word (PSW)*. The PSW contains bits which indicate such things as whether the previous arithmetic operations produced a positive, negative, or zero result. If a “subtract” instruction is followed by a “branch on zero” instruction, then the branch will be taken if the PSW indicates that the subtraction resulted in

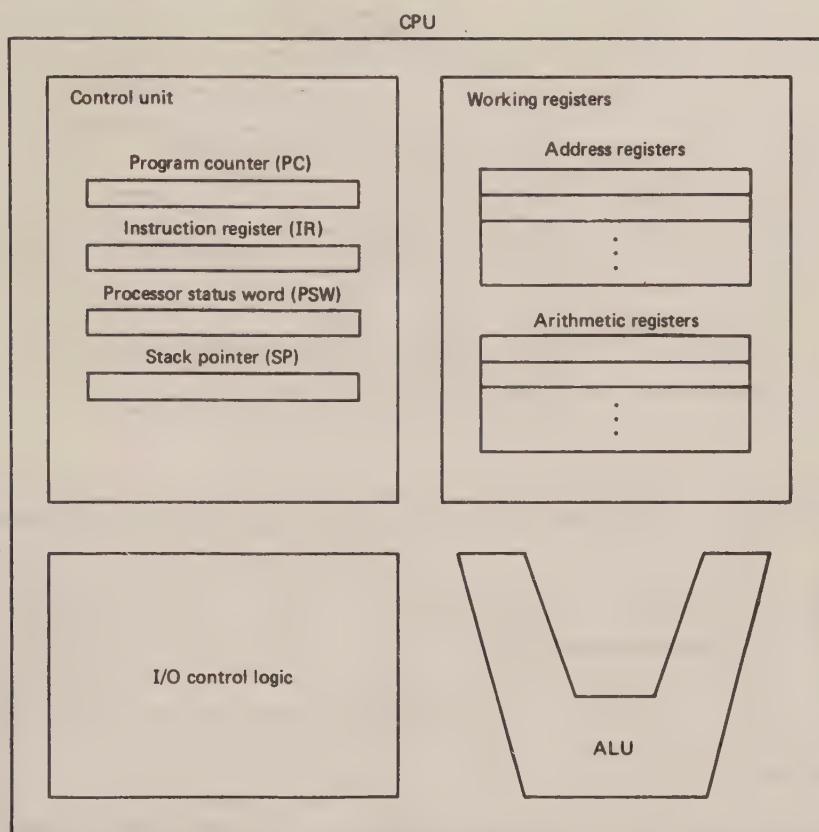


Figure 1-9 Typical CPU architecture. Glenn A. Gibson, James R. Young, INTRODUCTION TO PROGRAMMING USING FORTRAN 77, © 1982, p. 16. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, NJ.

a zero. If the PSW shows that the subtraction produced a nonzero result, the branch will not be taken. When a branch is taken a new sequence of instructions begins at the address to which the branch is made. A flowchart illustrating the sequencing of instructions within a computer is given in Fig. 1-10.

Looping is normally performed using conditional branch instructions, although some microprocessors have instructions that combine counting and/or testing with the conditional branching. Most loops, such as a FORTRAN DO-loop, involve incrementing or decrementing a counter and repeating the loop until the counter reaches a limit. Each time the counter is changed the result is compared with the limit, the PSW is set accordingly, and the branch is taken or not taken depending on the contents of the PSW.

The sequencing involved in a subroutine call is shown in Fig. 1-11 and requires a special form of branching. As with other branching, a subroutine call also causes the contents of the PC to be replaced by the address being branched to, but it must also save the current contents of the PC, which is the *return address*. The return branch instruction must be capable of restoring the return address to the

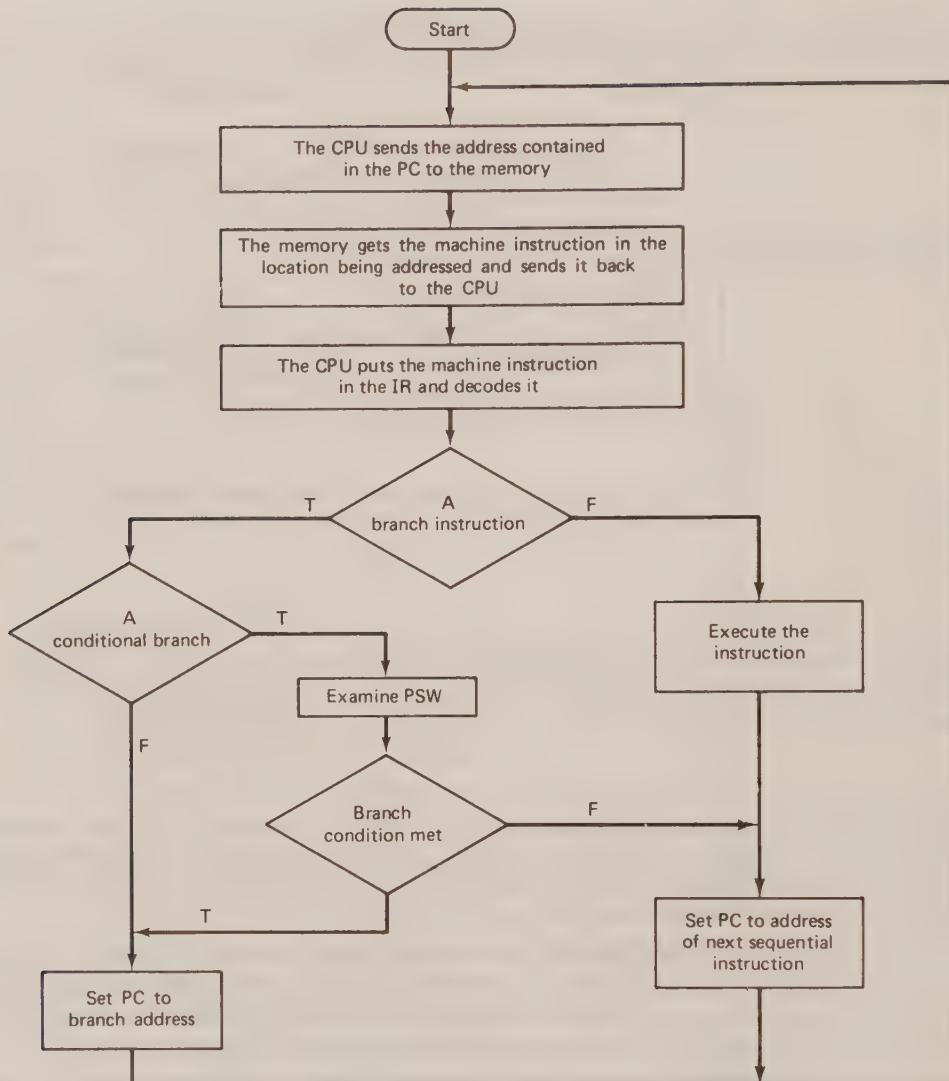


Figure 1-10 Instruction sequencing.

PC so that the main program can continue in sequence once the subroutine is complete. In addition to saving the return address, it is usually necessary to temporarily store other information, such as the contents of the working registers, while the subroutine is executing. This is so because the subroutine might otherwise destroy the original contents of these registers and they may be needed when the return is made to the main program. Normally, this information is stored in a special area in memory called the *stack*. The address of the stack location that

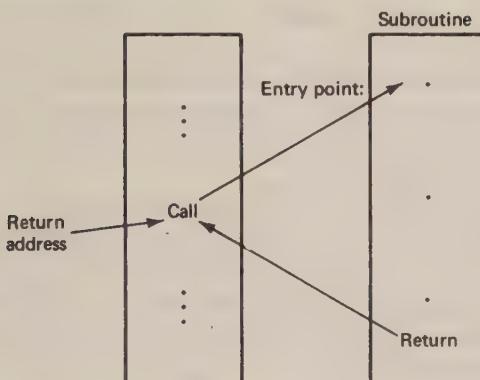


Figure 1-11 Subroutine call.

was last accessed (or, on some computers, that will be accessed next) is kept in the *stack pointer (SP) register*. Stacks and subroutines are discussed in detail in Chap. 4.

The working registers are for temporarily holding information that aids the addressing and computational process. They tend to be divided into two groups, an address group and an arithmetic group, although some registers may be such that they can be used for both purposes.

The address group is used for making the addressing of data more flexible. A datum being operated on by an instruction may be part of the instruction, its address may be part of the instruction, it may be in a register, its address may be in a register, or its address may be the sum of part of the instruction and the contents of one or more registers. Sometimes when a register is used the instruction simply indicates the register that contains the address, but other times the address determination is more complicated. For example, when accessing elements of an array the address of an element is comprised of two parts, a *base address*, which is the address of the first element in the array, and an *offset*. Because one often needs to index through an array, it is helpful if the offset can be easily incremented. Therefore, the address of an array element is frequently computed by adding two registers together, one which contains the base address and is called a *base register* and one which contains the offset and is called an *index register*. A two-dimensional array offers a slightly more complex situation which requires the addition of a base, a column offset, and a displacement. This normally involves the sum of part of the instruction (the displacement), a base register, and an index register. Base registers are also used to relocate programs and blocks of data within memory, a topic that is explored further in Chaps. 2 and 4.

Arithmetic registers are for temporarily holding the operands and results of the arithmetic operations. Because transfers over the system bus are the primary limiting factor with regard to speed, accessing a register is much faster than accessing memory. Therefore, if several operations must be performed on a set of data, it is better to input the data to the arithmetic registers, do the necessary calculations, and return the result to memory than it is to work directly from memory. The

tendency is that the more arithmetic registers a CPU contains, the faster it can execute its computations.

The ALU is for performing arithmetic, logical, shifting, and other operations. The I/O control section contains the CPU's logic that is related to the I/O operations. Exactly which logic is in the CPU and which is part of the external bus control logic depends on the system.

Although there are many variations of the CPU architecture described above, this discussion has summarized the operation and major parts of microprocessors in general. The purpose of this section has been to give the reader an overview that will help him or her study specific microprocessors. A thorough discussion of the architecture and operation of the Intel 8086/8088, the microprocessor on which this book is based, is given in Chap. 2.

1-5 MICROPROCESSORS IN DIGITAL SYSTEM DESIGN

Since 1970 microprocessors and their associated LSI circuits, such as memory and interface devices, have initiated a revolution in digital system design. This is due to the programmable nature of microprocessors and the fact that a single-chip microprocessor can provide the equivalent power of hundreds of SSI and MSI devices. More and more digital systems which had been implemented in hard-wired logic are using microprocessors and other LSI devices. This trend is clearly continuing. As the cost of microprocessors falls, even simple systems are based on a design in which a microprocessor acts as the controller, even though only a small fraction of the microprocessor's power is actually used. The reasons for this tendency can be understood by considering the major criteria which govern digital system design:

Cost—In addition to the IC cost, the system cost includes a per unit cost which can be broken down as follows: IC handling and inventory, IC testing, labor for wiring, soldering, and packaging, and the expense of buying the PC boards, power supplies, cabinetry, and other hardware. The purchase price of the ICs is normally less than 10% of the overall system cost, but the per unit cost is proportional to the number of ICs, not their internal complexity. Therefore, it is more economical to use more expensive LSI and VLSI devices as long as they replace a sufficient number of SSI and MSI devices.

Flexibility—After a system has been produced, changes or improvements may be needed. This could happen because design faults are identified after the system has been field tested or because new functions need to be added. If the system is hard-wired it must be redesigned, modified, and retested, which is both time consuming and costly. On the other hand, a microprocessor is programmable and changes can be made by simply modifying the control program, and normally little or no hardware redesign is necessary.

Reliability—Since the system failure rate is proportional to the number of devices, the system reliability increases as the complexity of the devices goes

up and the number of devices goes down. In addition, fewer devices imply fewer interconnections, and this reduces connection failures, noise, and timing problems.

Development Time—The process of conventional hard-wired system design is sequential in nature. This means that the next phase cannot begin until the previous phase has been completed. On the other hand, the design of a microprocessor-based system can be partitioned into software and hardware tasks which can proceed in parallel.

Speed—A hard-wired logic system has one major advantage over its microprocessor-based counterpart and that is that it can react much faster to its inputs. However, many applications do not need such high speed and the greater flexibility of a programmable system may be the overriding factor. Also, the newer microprocessors are faster and the speed gap is narrowing.

The microprocessor age began when integrated circuit technology had advanced enough to put all of the necessary functions of a CPU into a single IC device. Since that time the advancement of microprocessors has paralleled the advancement in microelectronic technology and the ability to include more and more logic on a single IC surface. Although the cost of a microprocessor increases with its complexity, it is much lower than the cost of the equivalent logic scattered over several less capable ICs. In addition to reducing the number of ICs needed to perform a given function, the total number of pins is reduced, thus the assembly costs are decreased.

As microelectronics fabrication techniques have improved from the LSI level to the VLSI level, microprocessors have advanced from having 4 data lines and 12 address lines, 4 of which were also the 4 data lines, to 16 data lines and 20 to 24 independent address lines. The former were good only for constructing calculators and simple controllers; the latter can be used as CPUs in sophisticated general purpose computers that rival the medium to large computers. The latest 16-bit single-chip microprocessors accommodate up to 24 megabytes of memory and include both multiprogramming and multiprocessing features. To accompany these advanced processors, the IC manufacturers are also improving the variety and complexity of supporting interface, bus logic, and memory devices.

As an example, let us consider the growth of the basic Intel family, which is depicted in Fig. 1-12. This family has expanded from the 4-bit 4004 microprocessor and its associated 4000 series chips to the 16-bit 8086 and its derivatives and supporting devices. The 8008, 8080, and 8085 represent a progression of 8-bit processors, with each new device including more circuitry and being more flexible. The 8088 is an 8-bit version of the 8086 which has fewer data lines but retains all of the processing features of the 8086. The 80186 and 80286 are single-chip extensions of the 8086 that are faster and provide additional processing power, power that is particularly useful when designing complex microcomputer systems. Also shown in the figure are the 8048 and its family, and Intel's advanced multichip 432 system. The Intel family represents an increase in performance by a factor of several thousand.

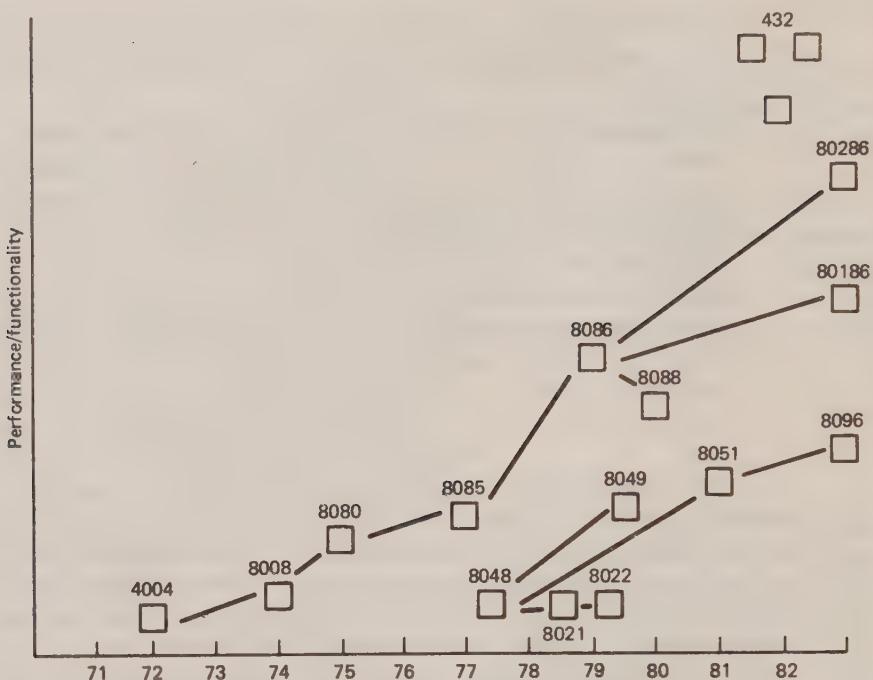


Figure 1-12 Basic Intel family of microprocessors. (Reprinted by permission of Intel Corporation. Copyright 1982.)

BIBLIOGRAPHY

1. Krutz, Ronald L., *Microprocessors and Logic Design* (New York: John Wiley & Sons, Inc., 1980).
2. Wollesen, Donald L., "The Computer Component Process of the 80's," *Computer*, 13, no. 2 (February, 1980), 59-67.
3. Sumney, Larry W., "VLSI with a Vengeance," *IEEE Spectrum*, 17, no. 4 (April, 1980), 24-27.
4. *The 8086 Family User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
5. Morse, Stephen P., Bruce W. Romenel, Stanley Mazor, and William B. Pohlman, "Intel Microprocessors—8008 to 8086," *Computer*, 13, no. 10 (October, 1980), 42-60.
6. Shima, Masatoshi, "Demystifying Microprocessor Design," *IEEE Spectrum*, 16, no. 7 (July, 1979), 22-30.
7. Balde, J. W., "Computer Packaging Workshop Explores Impact of LSI," *Computer*, 13, no. 11 (November, 1980), 100-102.
8. Sequin, C. H., "Instruction in MOS LSI Systems Design," *Computer*, 13, no. 3 (March, 1980), 67-73.
9. Gibson, Glenn A., and Yu-cheng Liu, *Microcomputers for Engineers and Scientists* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980).

10. Blakeslee, Thomas R., *Digital Design with Standard MSI and LSI* (New York: John Wiley & Sons, Inc., 1975).

EXERCISES

- Convert the following decimal numbers to binary numbers:
 (a) 19 (b) 79 (c) 463 (d) 1209 (e) 11,355
- Convert the following binary numbers to decimal numbers:
 (a) 10110 (b) 1011011 (c) 10010010
 (d) 1011010001110110
- Convert each of the following to hexadecimal:
 (a) 35_{10} (b) 298_{10} (c) 852_{10} (d) $162,023_{10}$
 (e) 11010 (f) 11010110 (g) 0110000100111011
- Convert each of the given hexadecimal numbers to its binary and decimal equivalents:
 (a) D5 (b) 9A26 (c) 7BF52A (d) 2A01BF57
- Given that

$$A = 1011 \quad B = 11001 \quad C = 100110 \quad D = 1000010011$$

perform the following computations in binary and check the answers using decimal arithmetic:

- $A + B$
- $C - B - A$
- AB
- D/A
- AC/B
- $D - AB$

- Given that

$$A = 4F5A20 \quad B = 2FF609$$

find

- $A + B$
- $A - B$
- $2B - A$
- $A/4$

(Note: A multiplication by 2^n can be performed by appending n binary zeros to the right, and a division by 2^n can be done by dropping the n rightmost bits—these bits would comprise the remainder.)

- State the approximate value of:
 (a) 2^{24} (b) 2^{32} (c) 2^{64}
 (Hint: $2^{10} \approx 10^3$.)
- Find how many distinct combinations can be represented by a sequence of:
 (a) 12 bits (b) 24 bits (c) 32 bits
- Find the 8-bit 2's complement of:
 (a) 01101001 (b) 11010010
- Assume the 2's complement format and that

$$A = 00110010 \quad B = 01001010 \quad C = 11101001 \quad D = 10111010$$

and find:

- $A + B$
- $A + C$
- $C + B$
- $C + D$
- $A - B$
- $C - A$
- $D - C$
- $A + D - C$

- Find the 4-digit 10's complement of:
 (a) 0232 (b) 9644

12. Assume the 4-digit 10's complement format and that

$$A = 0561 \quad B = 0352 \quad C = 9863 \quad D = 9179$$

and find:

- (a) $A + B$ (b) $A + C$ (c) $C + D$ (d) $A - B$
(e) $C - D$ (f) $B - C$

13. Express the packed BCD equivalents of the following in both binary and hexadecimal:
(a) 3251 (b) 12907

14. Express the following character strings in their ASCII equivalents (give the answers in hexadecimal):

- (a) SAM JONES (b) -75.61 (c) Hello, how are you?
(d) G. A. SMITH
281 DOVE
EL PASO, TEXAS

15. Show that the binary number representing

GREEN, ANNE

is numerically smaller than the one representing

GREEN, MARY

16. If there are 20 address lines and 16 data lines:

- (a) What is the memory address space assuming that the memory and I/O address spaces are separate?
(b) What is the range of 2's complement integers that can be transmitted over the data lines?

17. Repeat Exercise 16 assuming that there are 16 address lines and 8 data lines.

8086 Architecture

The Intel 8086, a 16-bit microprocessor, contains approximately 29,000 transistors and is fabricated using the HMOS technology. Its throughput is a considerable improvement over that of the Intel 8080, its 8-bit predecessor. Although some attempt at compatibility with the 8080 CPU architecture was made, the designers decided not to sacrifice sophistication in order to attain compatibility (Morse et al. [1]). By increasing the number of address pins from 16 to 20, the memory addressing capacity was increased from 64K bytes to $2^{20} \approx 1$ megabyte. The expanded memory capability made multiprogramming feasible and several multiprogramming features have been incorporated into the 8086's design. The 8086 also includes a number of features which enhance its multiprocessing capabilities, thus allowing it to be used with other processing elements such as the 8087 numeric data processor.

A pin assignment diagram for the 8086 is given in Fig. 2-1. The 8086 has 20 address pins, 16 of which are also used as data pins. The use of pins for both addresses and data means that both an address and datum cannot be sent to the system bus at the same time. This multiplexing of addresses and data reduces the number of pins needed, but does slow down the transfer of data. However, because of the timing on the bus, the transfer rate is not decreased as much as one might expect. There are 16 control lines for providing handshaking signals during bus transfers and for permitting at least some external control of the CPU. The 8086 requires only one supply voltage, +5 V, and one clock phase whose frequency can be up to 5 MHz. (There are actually two other versions of the 8086, the 8086-2, which permits a clock frequency of up to 8 MHz, and the 8086-1, which can handle up to 10 MHz.) Rounding out the 40-pin configuration are two grounds, pins 1 and 20.

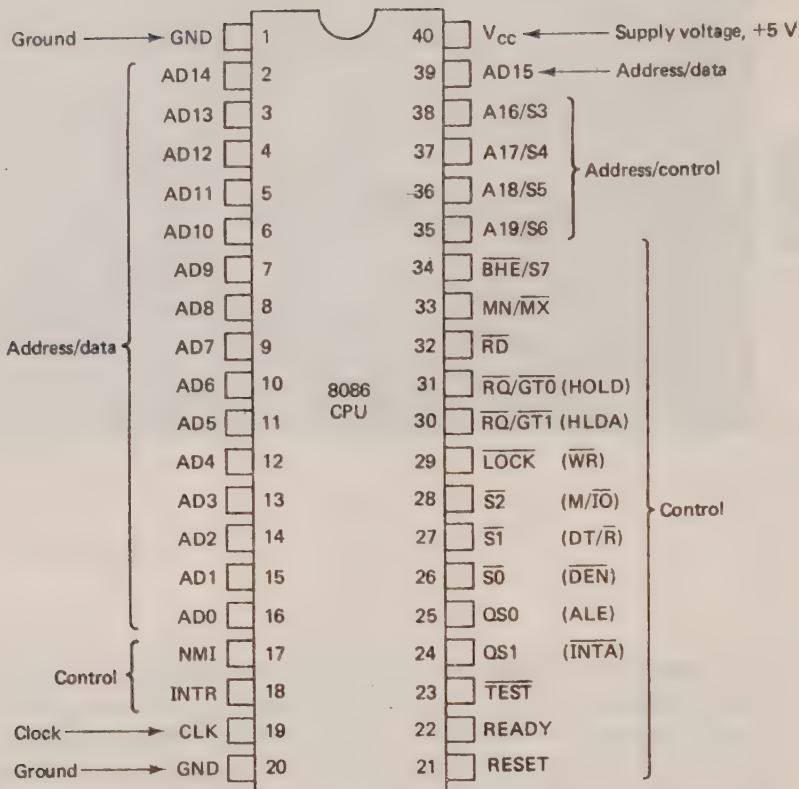
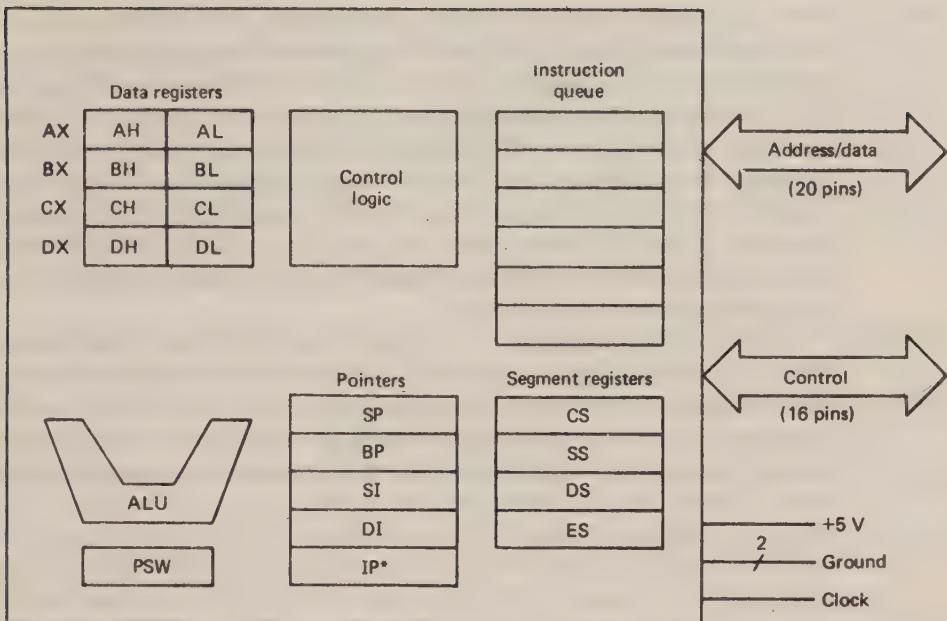


Figure 2-1 8086 pin assignments. (Reprinted by permission of Intel Corporation. Copyright 1981.)

This chapter begins with a description of the 8086's CPU architecture and internal operation. The third section defines the ways in which the 8086 can address data and perform branches and discusses the 8086's machine language instruction formats. Section 2-4 discusses the amounts of time needed to execute the various instructions and Sec. 2-5 describes the Intel 8088, which is a software compatible 8-bit version of the 8086. The 8088 was designed to allow 8080 and 8085 hardware configurations to be upgraded to an 8086 software environment.

2-1 CPU ARCHITECTURE

Figure 2-2 shows the internal architecture of the 8086. Except for the instruction register, which is actually a 6-byte queue, the control unit and working registers described in Sec. 1-4 are divided into three groups according to their functions. There are the data group, which is essentially the set of arithmetic registers; the pointer group, which includes base and index registers, but also contains the pro-



*For the 8086 the program counter is called the instruction pointer (IP).

Figure 2-2 8086's internal configuration.

gram counter and stack pointer; and the segment group, which is a set of special purpose base registers. All of the registers are 16 bits wide.

The data group consists of the AX, BX, CX, and DX registers. These registers can be used to store both operands and results and each of them can be accessed as a whole, or the upper and lower bytes can be accessed separately. For example, either the 2 bytes in AX can be used together, or the upper byte AH or the lower byte AL can be used by itself by specifying AH or AL, respectively. For purposes of converting 8080 software into 8086 software the following correspondences can be drawn:

| 8086 | 8080 |
|------|------|
| AL | A |
| BH | H |
| BL | L |
| CH | B |
| CL | C |
| DH | D |
| DL | E |

In addition to serving as arithmetic registers, the BX, CX, and DX registers play special addressing, counting, and I/O roles:

BX may be used as a base register in address calculations.

CX is used as an implied counter by certain instructions.

DX is used to hold the I/O address during certain I/O operations.

The pointer and index group consists of the IP, SP, BP, SI, and DI registers. The instruction pointer (IP) and SP registers are essentially the program counter and stack pointer registers, but the complete instruction and stack addresses are formed by adding the contents of these registers to the contents of the code segment (CS) and stack segment (SS) registers discussed below. BP is a base register for accessing the stack and may be used with other registers and/or a displacement that is part of the instruction. The SI and DI registers are for indexing. Although they may be used by themselves, they are often used with the BX or BP registers and/or a displacement. Except for the IP, a pointer can be used to hold an operand, but must be accessed as a whole.

To provide flexible base addressing and indexing, a data address may be formed by adding together a combination of the BX or BP register contents, SI or DI register contents, and a displacement. The result of such an address computation is called an *effective address* (EA) or *offset*. [The Intel manuals tend to use the term "effective address" when discussing the machine language and the term "offset" when discussing the assembler language. The word "displacement" is used to indicate a quantity that is added to the contents of a register(s) to form an EA.] The final data address, however, is determined by the EA and the appropriate data segment (DS), extra segment (ES), or stack segment (SS) register.

The segment group consists of the CS, SS, DS, and ES registers. As indicated above, the registers that can be used for addressing, the BX, IP, SP, BP, SI, and DI registers, are only 16 bits wide and, therefore, an effective address has only 16 bits. On the other hand, the address put on the address bus, called the *physical address*, must contain 20 bits. The extra 4 bits are obtained by adding the effective address to the contents of one of the segment registers as shown in Fig. 2-3. The addition is carried out by appending four 0 bits to the right of the number in the segment register before the addition is made; thus a 20-bit result is produced. As an example, if (CS) = 123A and (IP) = 341B, then the next instruction will be fetched from

$$\begin{array}{rl}
 341B & \text{Effective address} \\
 + 123A0 & \text{Beginning segment address} \\
 \hline
 157BB & \text{Physical address of instruction}
 \end{array}$$

[It is standard notation for parentheses around an entity to mean "contents of," e.g., (IP) means the contents of IP. Also, all addresses are given in hexadecimal.]

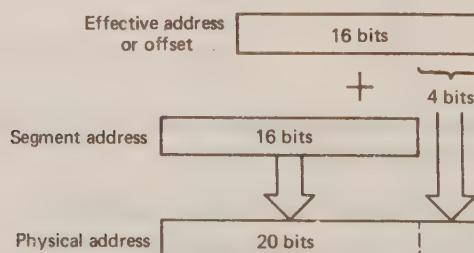


Figure 2-3 Formation of a physical address.

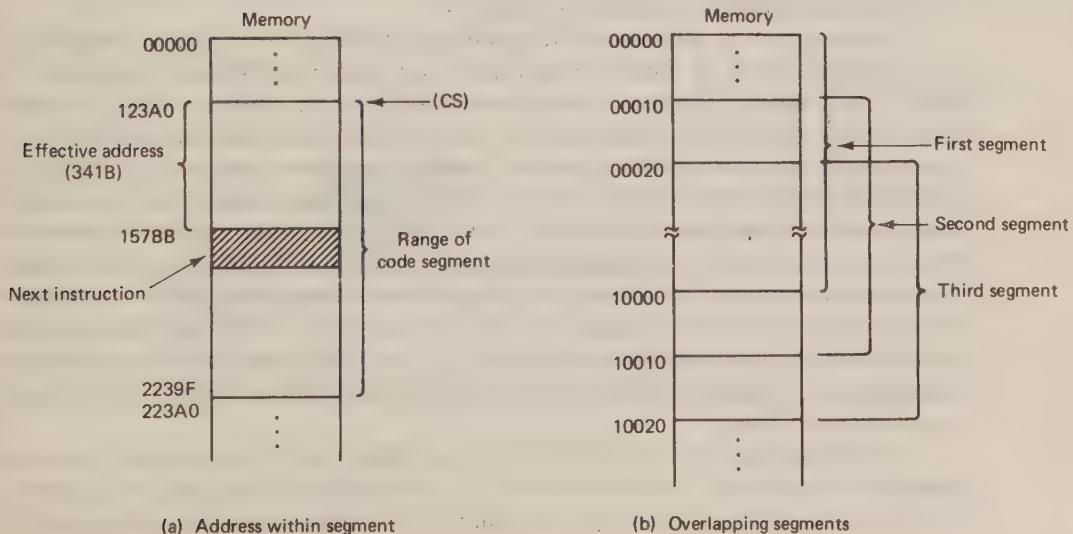
The utilization of the segment registers essentially divides the memory space into overlapping segments, with each segment being 64K bytes long and beginning at a 16-byte, or paragraph, boundary, i.e., beginning at an address that is divisible by 16. We will hereafter refer to the contents of a segment register as the *segment address*, and the segment address multiplied by 16_{10} as the *beginning physical segment address*, or simply, the *beginning segment address*. An illustration of the example above is given in Fig. 2-4(a) and the overall segmentation of memory is shown in Fig. 2-4(b).

The advantages of using segment registers are that they:

1. Allow the memory capacity to be 1 megabyte even though the addresses associated with the individual instructions are only 16 bits wide.
2. Allow the instruction, data, or stack portion of a program to be more than 64K bytes long by using more than one code, data, or stack segment.
3. Facilitate the use of separate memory areas for a program, its data, and the stack.
4. Permit a program and/or its data to be put into different areas of memory each time the program is executed.

Figure 2-5 shows how a program's code and its associated data and stack can be separated in memory. The simpler and conventional approach is to let both the code and the data reside in one contiguous area in memory and put the stack in some fixed area which always begins at, say, address 08000. This is satisfactory if there is only one program in memory at a time, but in a multiprogramming environment there may be several programs in memory simultaneously. For multi-

Figure 2-4 Address computations and memory segmentation.



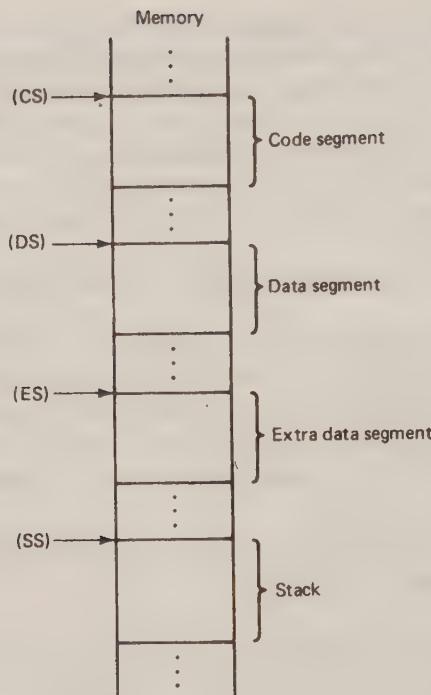


Figure 2-5 Separation of a program's code, its data, and its stack.

programming it is better to keep the contiguous chunks of memory as small as possible and, therefore, it is better to separate the code from the data. Analogously, the space in a pail is more fully utilized by filling it with small stones than with large stones. The option of making the code and data contiguous is always present, because the contents of CS, DS, and ES can always be made equal. Multiprogramming is examined in detail in Chap. 7.

Advantage 4 is also related to multiprogramming. For example, consider a time-sharing system in which several users are executing different programs. If there is not enough memory space, these programs are shuttled back and forth between the memory and mass storage. As they take turns using memory they are dynamically put into different places in memory. A given program may begin at physical address 02000 while it is currently executing, but the next time it is brought into memory it may begin at 1A300. The question is: If the program contains a branch instruction that would currently cause a branch to 02600, would the branch be correctly taken to 1A900 when the program is relocated to 1A300? The answer is yes provided that the contents of CS are changed to 1A300 when the program is relocated. This situation is depicted in Fig. 2-6. Similar statements can be made regarding the relocation of a block of data or a stack. Relocation will be discussed more fully in Chap. 7.

Although segments are 64K bytes long, they can be overlapped to better utilize the memory. If, as shown in Fig. 2-7, a program fills only part of a segment, another code, data, or stack segment could overlap the program's segment and the

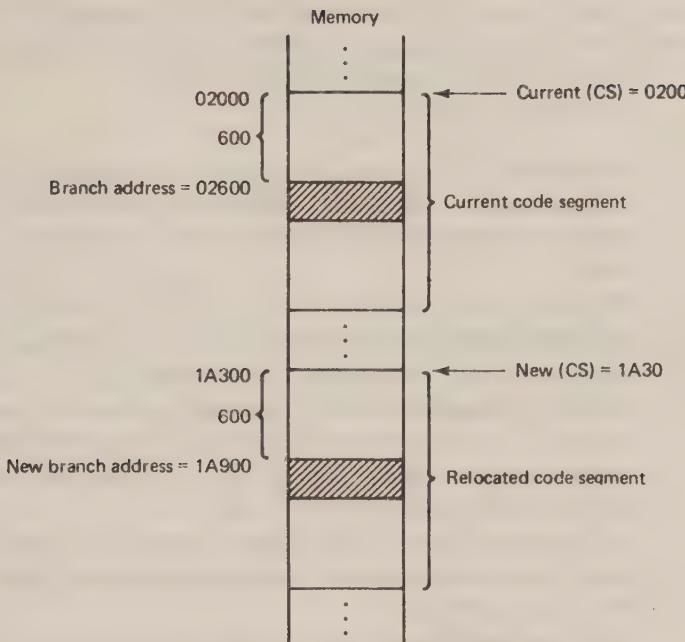


Figure 2-6 Program relocation using the CS register.

beginning address of the second segment could be within 16 bytes of the end of the program. The only wasted space would be the few bytes between the end of the program and the next 16-byte boundary.

The 8086's PSW contains 16 bits, but 7 of them are not used. Each bit in the PSW is called a *flag*. The 8086 flags are divided into the *conditional flags*, which reflect the result of the previous operation involving the ALU, and the *control*

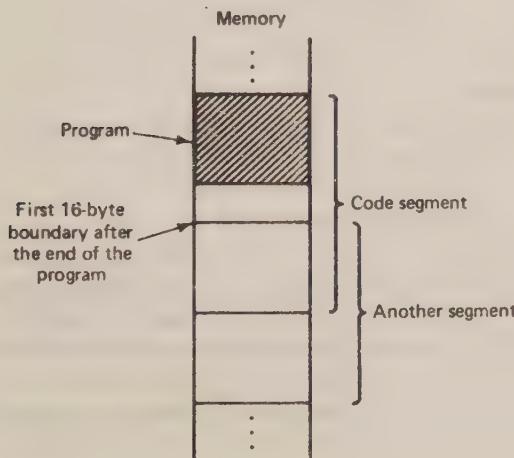


Figure 2-7 Overlapping segments.

flags, which control the execution of special functions. The flags are summarized in Fig. 2-8. The lower byte in the PSW corresponds to the 8-bit PSW in the 8080 and contains all of the condition flags except OF.

The condition flags are:

SF (Sign Flag)—Is equal to the MSB of the result. Since in 2's complement negative numbers have a 1 in the MSB and for nonnegative numbers this bit is 0, this flag indicates whether the previous result was negative or nonnegative.

ZF (Zero Flag)—Is set to 1 if the result is zero and 0 if the result is nonzero.

PF (Parity Flag)—Is set to 1 if the low-order 8 bits of the result contain an even number of 1s; otherwise it is cleared.

CF (Carry Flag)—An addition causes this flag to be set if there is a carry out of the MSB, and a subtraction causes it to be set if a borrow is needed. Other instructions also affect this flag and its value will be discussed when these instructions are defined.

AF (Auxiliary Carry Flag)—Is set if there is a carry out of bit 3 during an addition or a borrow by bit 3 during a subtraction. This flag is used exclusively for BCD arithmetic.

OF (Overflow Flag)—Is set if an overflow occurs, i.e., a result is out of range. More specifically, for addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice versa.

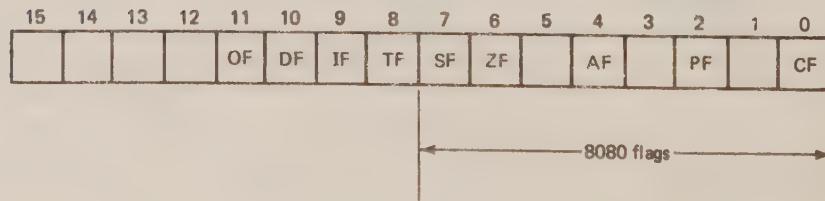
As an example, if the previous instruction performed the addition

$$\begin{array}{r}
 0010\ 0011\ 0100\ 0101 \\
 +\ 0011\ 0010\ 0001\ 1001 \\
 \hline
 0101\ 0101\ 0101\ 1110
 \end{array}$$

then following the instruction:

SF = 0 ZF = 0 PF = 0 CF = 0 AF = 0 OF = 0

Figure 2-8 8086's PSW.



If the previous instruction performed the addition

$$\begin{array}{r}
 0101\ 0100\ 0011\ 1001 \\
 +\ 0100\ 0101\ 0110\ 1010 \\
 \hline
 1001\ 1001\ 1010\ 0011
 \end{array}$$

then the flags would be:

SF = 1 ZF = 0 PF = 1 CF = 0 AF = 1 OF = 1

The control flags are:

DF (Direction Flag)—Used by string manipulation instructions. If clear, the string is processed from its beginning with the first element having the lowest address. Otherwise, the string is processed from the high address towards the low address. (See Chap. 5.)

IF (Interrupt Enable Flag)—If set, a certain type of interrupt (a maskable interrupt) can be recognized by the CPU; otherwise, these interrupts are ignored. (See Chaps. 4 and 6.)

TF (Trap Flag)—If set, a trap is executed after each instruction. (See Chap. 4.)

2-2 INTERNAL OPERATION

The general operation of a computer as outlined in Sec. 1-4 consists of:

1. Fetching the next instruction from the address indicated by the PC.
2. Putting it in the instruction register and decoding it while the PC is incremented to point to the next instruction.
3. Executing the instruction and, if a branch is to be taken, resetting the PC to the branch address.
4. Repeating steps 1 through 3.

The operation of the 8086 follows this basic pattern, but there are differences and some of the operations may be overlapped.

The address of the next instruction is determined by the sum of the (IP) and (CS) \times 16_{10} , and the instruction register is a 6-byte first-in/first-out (FIFO) queue that is continually being filled whenever the system bus is not needed for some other operation. This "look ahead" feature can significantly increase the CPU's throughput because much of the time the next instruction is already in the CPU when the present instruction completes its execution. If a branch is taken, then the instruction queue is flushed out and there is no time saving, but on the average this occurs only a small percent of the time. To demonstrate the time available for loading the queue let us consider an 8-bit multiply instruction. This instruction

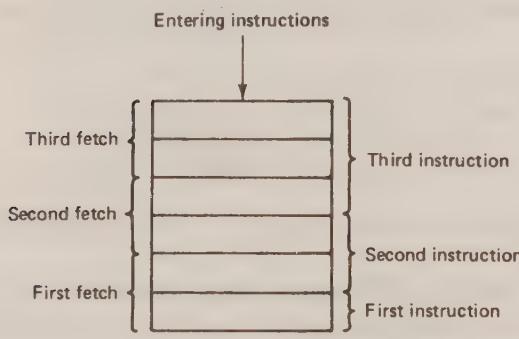
requires at least 71 clock cycles, but only 4 clock cycles are needed to input a word from memory. Therefore, during its execution there would be an ample number of extra clock cycles in which the bus is free to fill the instruction queue.

Although the 8086 can address a word which begins with either an odd address or an even address, for the odd-address case two memory references are required. One memory access is needed for the low-order byte and one for the high-order byte. This means that time is saved if words are stored at only even addresses. Even though the 8086's instructions may be from 1 to 6 bytes long, the 6-byte queue makes it possible to bring instructions in a word at a time using even addresses. The only exception occurs when there is a branch to an odd address. When this happens, the 8086 brings in 1 byte and then continues with even-address words. Figure 2-9(a) shows how the queue is filled by a sequence of the form:

1-byte instruction.

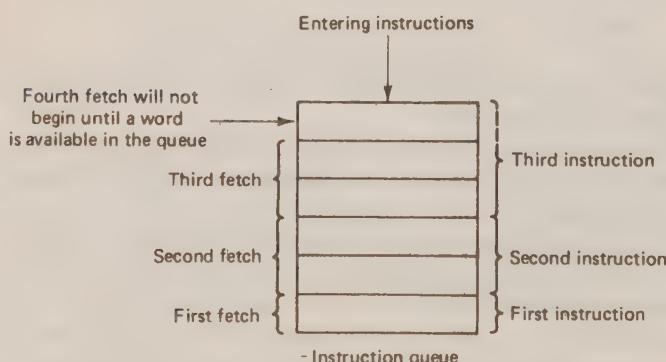
2-byte instruction.

3-byte instruction.



(a) First instruction has an even address

Figure 2-9 Filling the instruction queue after a branch.



(b) First instruction has an odd address

beginning with an even address, and Fig. 2-9(b) shows the same sequence but assumes an odd beginning address. Note that in the latter case the last byte of the third instruction will not be brought into the queue until an empty word in the queue becomes available.

2-3 MACHINE LANGUAGE INSTRUCTIONS

An instruction is divided into groups of bits, or *fields*, with one field, called the *operation code* (or *op code*), indicating what the computer is to do, and the other fields, called the *operands*, indicating the information needed by the instruction in carrying out its task. An operand may contain a datum, at least part of the address of a datum, an indirect pointer to a datum, or other information pertaining to the data to be acted on by the instruction. A general instruction format is shown in Fig. 2-10.

Instructions may contain several operands, but the more operands and the longer these operands are, the more memory space they will occupy and the more time it will take to transfer each instruction into the CPU. In order to minimize the total number of bits in an instruction, most instructions, particularly those for 16-bit computers, are limited to one or two operands with at least one operand in a two-operand instruction involving a register. Because the memory and/or I/O spaces are relatively large, memory and I/O addresses require several bits, but because the number of registers is small, it takes only a few bits to specify a register. Therefore, one means of conserving instruction bits is to use registers as much as possible. The two-operand limitation does reduce the flexibility of many instructions, but normally the extra flexibility is not really needed. For example, an addition instruction, which involves the two numbers being added and the result, is reduced to two operands by putting the sum into the location that contained the addend. This means that the addend is lost, but this is usually not important. If it is important, the addend could be duplicated (by also storing it elsewhere) before the addition is executed.

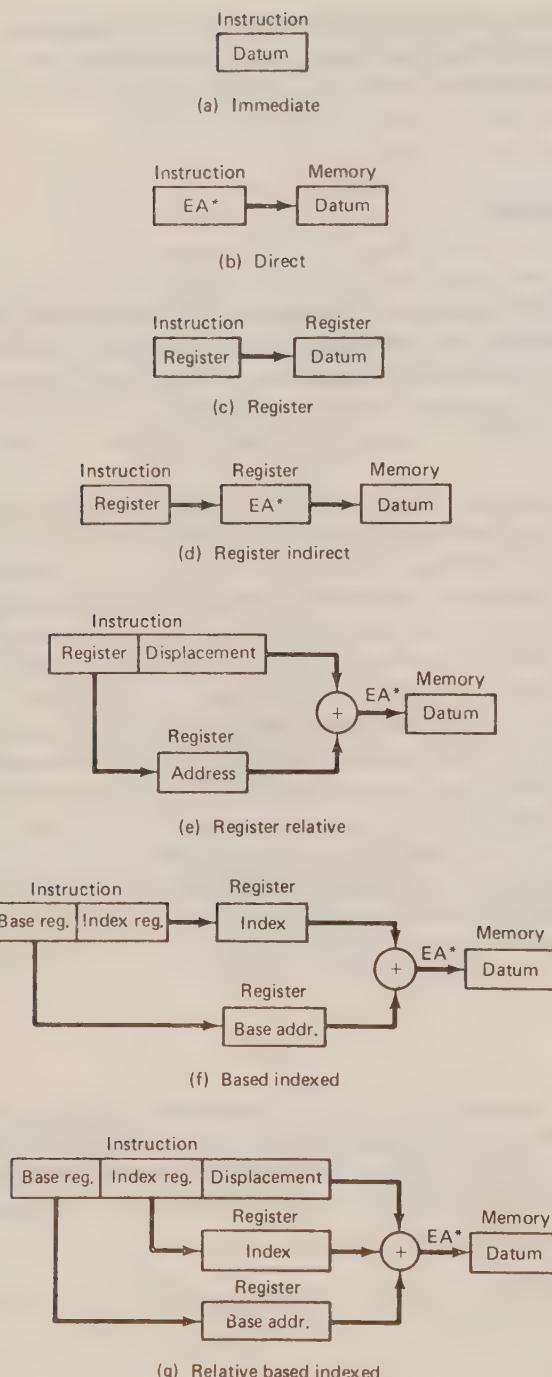
2-3-1 Addressing Modes

The way in which an operand is specified is called its *addressing mode*. The addressing modes for the 8086 instructions are typical and are discussed below. They are broken into two categories, those for data and those for branch addresses. Figure 2-11 graphically shows how the operands are determined for the various data-related addressing modes. These modes are:

Immediate—The datum is either 8 bits or 16 bits long and is part of the instruction.

| | | | |
|---------|---------|-----|---------|
| Op code | Operand | ... | Operand |
|---------|---------|-----|---------|

Figure 2-10 General instruction format.



*EA is added to 16_{10} times the contents of the appropriate segment register.

Figure 2-11 Data-related addressing modes.

Direct—The 16-bit effective address of the datum is part of the instruction.

Register—The datum is in the register that is specified by the instruction. For a 16-bit operand, a register may be AX, BX, CX, DX, SI, DI, SP, or BP, and for an 8-bit operand a register may be AL, AH, BL, BH, CL, CH, DL, or DH.

Register Indirect—The effective address of the datum is in the base register BX or an index register that is specified by the instruction, i.e.,

$$EA = \left\{ \begin{array}{l} (BX) \\ (DI) \\ (SI) \end{array} \right\}$$

Register Relative—The effective address is the sum of an 8- or 16-bit displacement and the contents of a base register or an index register, i.e.,

$$EA = \left\{ \begin{array}{l} (BX) \\ (BP) \\ (SI) \\ (DI) \end{array} \right\} + \left\{ \begin{array}{l} 8\text{-bit displacement} \\ (\text{sign extended}) \\ 16\text{-bit displacement} \end{array} \right\}$$

Based Indexed—The effective address is the sum of a base register and an index register, both of which are specified by the instruction, i.e.,

$$EA = \left\{ \begin{array}{l} (BX) \\ (BP) \end{array} \right\} + \left\{ \begin{array}{l} (SI) \\ (DI) \end{array} \right\}$$

Relative Based Indexed—The effective address is the sum of an 8- or 16-bit displacement and a based indexed address, i.e.,

$$EA = \left\{ \begin{array}{l} (BX) \\ (BP) \end{array} \right\} + \left\{ \begin{array}{l} (SI) \\ (DI) \end{array} \right\} + \left\{ \begin{array}{l} 8\text{-bit displacement} \\ (\text{sign extended}) \\ 16\text{-bit displacement} \end{array} \right\}$$

For example, if

$$(BX) = 0158 \quad (DI) = 10A5 \quad \text{Displacement} = 1B57 \quad (DS) = 2100$$

and DS is used as the segment register, then the effective and physical addresses produced by these quantities and the various addressing modes would be

Direct: $EA = 1B57$

$$\text{Physical address} = 1B57 + 21000 = 22B57$$

Register: No effective address—datum is in specified register.

Register indirect assuming register BX:

$$EA = 0158$$

$$\text{Physical address} = 0158 + 21000 = 21158$$

Register relative assuming register BX:

$$EA = 0158 + 1B57 = 1CAF$$

$$\text{Physical address} = 1CAF + 21000 = 22CAF$$

Based indexed assuming registers BX and DI:

$$EA = 0158 + 10A5 = 11FD$$

$$\text{Physical address} = 11FD + 21000 = 221FD$$

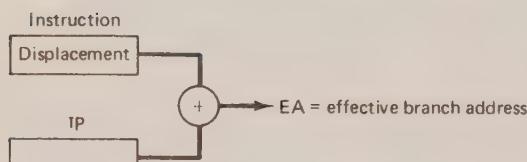
Relative based-indexed assuming BX and DI:

$$EA = 0158 + 10A5 + 1B57 = 2D54$$

$$\text{Physical address} = 2D54 + 21000 = 23D54$$

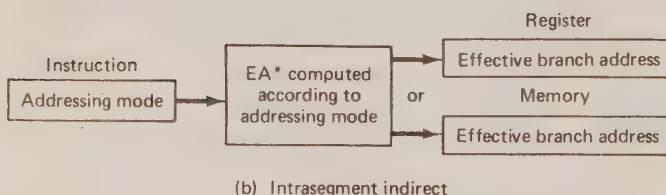
The addressing modes for indicating branch addresses are graphically defined in Fig. 2-12 and are:

Intrasegment Direct—The effective branch address is the sum of an 8- or 16-bit displacement and the current contents of IP. When the displacement is 8 bits long, it is referred to as a short jump. Intrasegment direct addressing is what most computer books refer to as relative addressing because the dis-

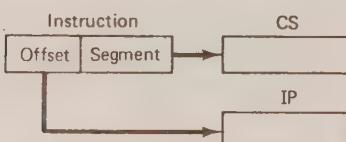


(a) Intrasegment direct

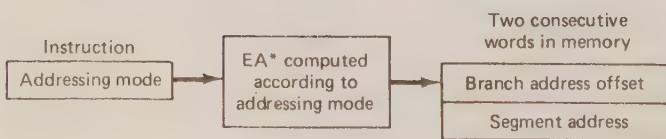
Figure 2-12 Branch-related addressing modes.



(b) Intrasegment indirect



(c) Intersegment direct



(d) Intersegment indirect

*EA is added to 16_{10} times the contents of the appropriate segment register.

placement is computed "relative" to the IP. It may be used with either conditional or unconditional branching, but a conditional branch instruction can have only an 8-bit displacement.

Intrasegment Indirect—The effective branch address is the contents of a register or memory location that is accessed using any of the above data-related addressing modes except the immediate mode. The contents of IP are replaced by the effective branch address. This addressing mode may be used only in unconditional branch instructions.

Intersegment Direct—Replaces the contents of IP with part of the instruction and the contents of CS with another part of the instruction. The purpose of this addressing mode is to provide a means of branching from one code segment to another.

Intersegment Indirect—Replaces the contents of IP and CS with the contents of two consecutive words in memory that are referenced using any of the above data-related addressing modes except the immediate and register modes.

Note that the physical branch address is the new contents of IP plus the contents of CS multiplied by 16_{10} . An intersegment branch must be unconditional.

To demonstrate how indirect branching works with some of the data-related addressing modes, suppose that

$$(BX) = 1256 \quad (SI) = 528F \quad \text{Displacement} = 20A1$$

Then:

With direct addressing, the effective branch address is the contents of:

$$20A1 + (DS) \times 16_{10}$$

With register relative addressing assuming register BX, the effective branch address is the contents of:

$$1256 + 20A1 + (DS) \times 16_{10}$$

With based indexed addressing assuming registers BX and SI, the effective branch address is the contents of:

$$1256 + 528F + (DS) \times 16_{10}$$

2-3-2 Instruction Formats

Several representative 8086 instruction formats are shown in Fig. 2-13. The instructions vary from 1 to 6 bytes in length and a complete summary of them is given in Sec. 3-12. Displacements and immediate data may be either 8 bits or 16 bits long depending on the instruction. The op code and addressing mode designations are in the first 1 or 2 bytes of an instruction. The op code/addressing mode

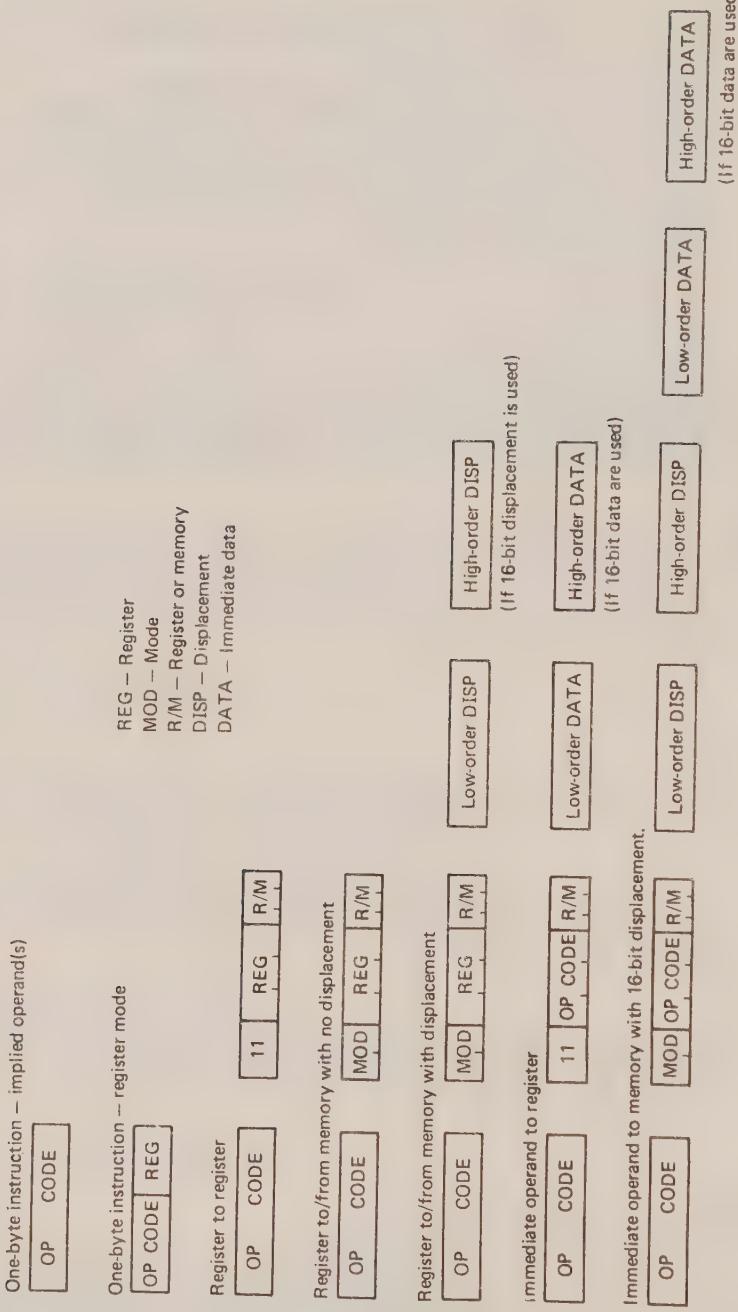


Figure 2-13 Examples of 8086 instruction formats.

byte(s) may be followed by:

- No additional bytes.
- A 2-byte EA (for direct addressing only.)
- A 1- or 2-byte displacement.
- A 1- or 2-byte immediate operand.
- A 1- or 2-byte displacement followed by a 1- or 2-byte immediate operand.
- A 2-byte displacement and a 2-byte segment address (for direct intersegment addressing only).

Which of these possibilities is used is determined by the op code and addressing mode. If a displacement or immediate operand is 2 bytes long, the low-order byte always appears first.

The op code usually occupies the first byte, and only the first byte, of an instruction, but there are a few instructions in which a register designation is in the first byte and a few other instructions in which 3 bits of the op code are in the second byte. Within most of the op codes there are special 1-bit indicators. They are:

W-bit—If an instruction can operate on either a byte or a word, the op code includes a W-bit which indicates whether a byte ($W=0$) or a word ($W=1$) is being accessed.

D-bit—For double-operand instructions (except for instructions with one operand being immediate and string instructions, which are discussed in Chap. 5). One of the operands must be a register specified by a REG field. For these instructions the D-bit is used to indicate whether the register specified by REG is the source operand ($D=0$) or the destination operand ($D=1$).

S-bit—An 8-bit 2's complement number can be extended to a 16-bit 2's complement number by letting all of the bits in the high-order byte equal the MSB in the low-order byte. This is referred to as *sign extension*. The S-bit appears with the W-bit in the immediate to register/memory add, subtract, and compare instructions and is assigned as follows:

8-bit operation—S:W = 00

16-bit operation with a 16-bit immediate operand—S:W = 01

16-bit operation with a sign-extended 8-bit immediate operand—S:W = 11

For small numbers, the latter case would permit the use of a 1-byte immediate operand.

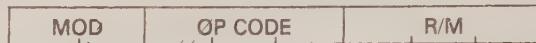
V-bit—Used by shift and rotate instructions to determine the number of shifts (see Chap. 3).

Z-bit—Used by the REP instruction (which is discussed in Chap. 5).

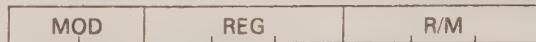
A register designation is 2 bits long if it is for a segment register and 3 bits long if it is for any other type of register. The register addresses (i.e., register bit

combination designations) are given in Fig. 2-14. There is no ambiguity in using, say, 00 to represent the segment register ES and 000 to represent both the register AX and its low-order byte AL. This is because the intended type of register is implied by the op code.

If there are two op code/addressing mode bytes, then the second byte is of one of the following two forms:



or

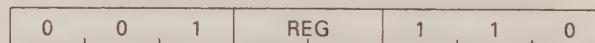


The first of these forms is for single-operand instructions (or instructions involving two operands with one of them being implied by the op code), and the second is for double-operand instructions, in which case REG specifies a register that is the source operand or destination operand depending on the value of the D-bit.

The operand specified by the MOD and R/M fields is determined according to the table in Fig. 2-15. If MOD \neq 11, an effective address is computed according to the entries in the table. Note that MOD = 00 means that there is no displacement except when R/M = 110, in which case direct addressing is implied. MOD = 01 means that the third byte of the instruction contains an 8-bit displacement which is automatically sign-extended to 16 bits before it is used to compute the effective address, MOD = 10 means that the third and fourth bytes of the instruction contain a 16-bit displacement. For MOD = 11, the operand is in the register whose address is designated by the R/M field.

Also indicated in Fig. 2-15 is the segment register to be used for each of the MOD and R/M combinations. Although the effective address of a memory operand is determined by the MOD and R/M fields, the 20-bit physical address is found by adding the effective address to the contents of a segment register multiplied by 16. For the address modes involving the BP register, the segment register SS is added to the effective address to produce the physical address. For all other address modes, the segment register DS is used.

To permit exceptions to the segment register usage given in Fig. 2-15, a special 1-byte instruction called a *segment override prefix* is available. A segment override prefix has the form



| Register Address | Registers W=1 | Registers W=0 | Register Address | Segment Register |
|------------------|---------------|---------------|------------------|------------------|
| 000 | AX | AL | 00 | ES |
| 001 | CX | CL | 01 | CS |
| 010 | DX | DL | 10 | SS |
| 011 | BX | BL | 11 | DS |
| 100 | SP | AH | | |
| 101 | BP | CH | | |
| 110 | SI | DH | | |
| 111 | DI | BH | | |

Figure 2-14 Register addresses.

| R/M \ MOD | 00 | 01 | 10 | 11 W=1 | 11 W=1 |
|------------------|-----------------|--------------------|---------------------|-----------|-----------|
| 000 Segment reg. | (BX)+(SI) DS | (BX)+(SI)+D8 DS | (BX)+(SI)+D16 DS | AL | AX |
| 001 Segment reg. | (BX)+(DI) DS | (BX)+(DI)+D8 DS | (BX)+(DI)+D16 DS | CL | CX |
| 010 Segment reg. | (BP)+(SI) SS | (BP)+(SI)+D8 SS | (BP)+(SI)+D16 SS | DL | DX |
| 011 Segment reg. | (BP)+(DI) SS | (BP)+(DI)+D8 SS | (BP)+(DI)+D16 SS | BL | BX |
| 100 Segment reg. | (SI) DS | (SI)+D8 DS | (SI)+D16 DS | AH | SP |
| 101 Segment reg. | (DI) DS | (DI)+D8 DS | (DI)+D16 DS | CH | BP |
| 110 Segment reg. | D16 DS | (BP)+D8 SS | (BP)+D16 SS | DH | SI |
| 111 Segment reg. | (BX) DS | (BX)+D8 DS | (BX)+D16 DS | BH | DI |

D8: 8-bit displacement (sign extended when executed)
 D16: 16-bit displacement

Figure 2-15 Address modes and default segment registers for various MOD and R/M field combinations.

If an instruction is preceded by a segment override prefix, the segment register REG is used for data reference during the execution of that instruction. For the addressing modes given in the figure, DS may be overridden by CS, SS, or ES; and when BP is used, SS may be overridden by DS, CS, or ES. Specific cases that cannot involve overrides are:

1. The CS register is always used as the segment register when computing the address of the next instruction to be executed.
2. When SP is used, SS is always the segment register. (Stack operations are discussed in Chap. 4.)
3. For string operations the ES register is always used as the segment register for the destination operand. (String operations are discussed in Chap. 5.)

To further examine the 8086's machine instructions, let us consider the addition (ADD) instruction. An ADD causes the contents of the location indicated by the source operand to be added to the contents of the location indicated by the destination operand and the sum to replace the contents of the location indicated by the destination operand. This instruction may take on one of the three forms shown in Fig. 2-16, depending on the addressing mode.

Figure 2-17 shows the machine language code for two ADD instructions, both of which add the contents of register BH to the contents of CL and put the result in CL. In the first instruction D=1 indicates that REG = 001 = CL is where the sum will be stored. MOD=11, so that R/M designates a register, the register

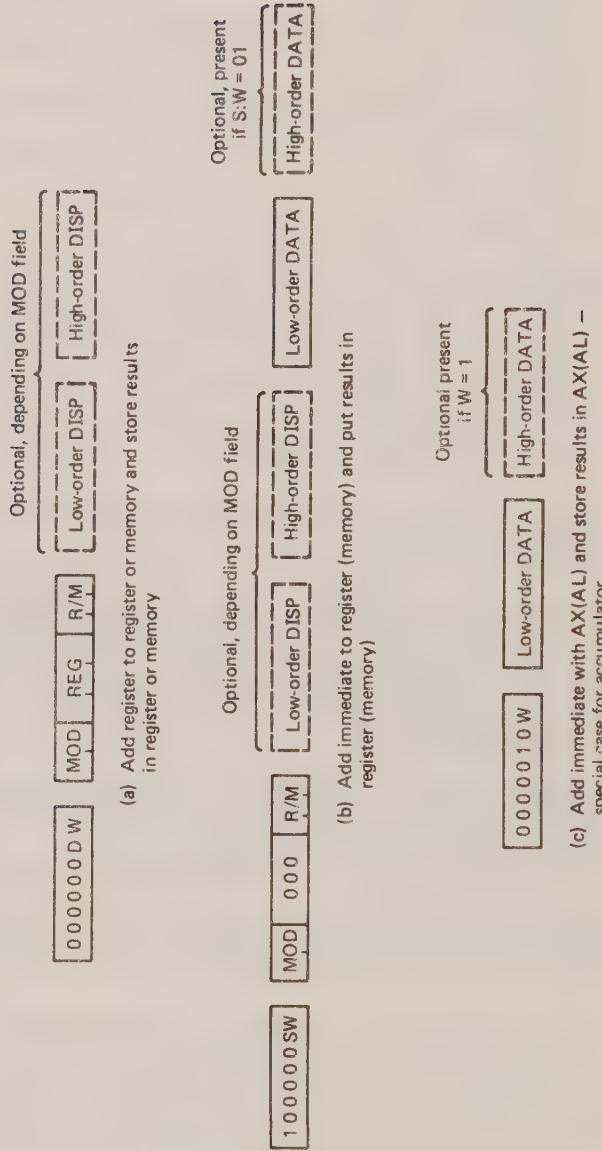


Figure 2-16 Formats for the ADD instruction.

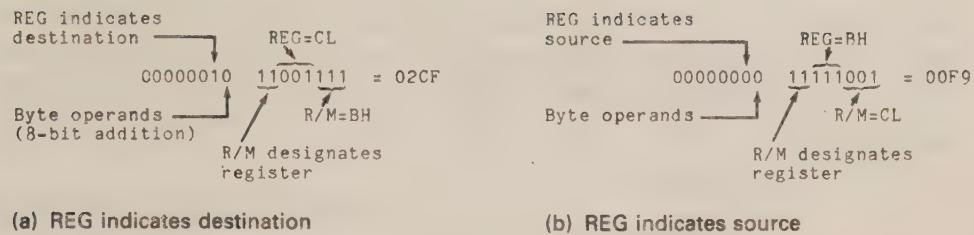


Figure 2-17 Two equivalent instructions for adding the contents of the BH register to those of the CL register.

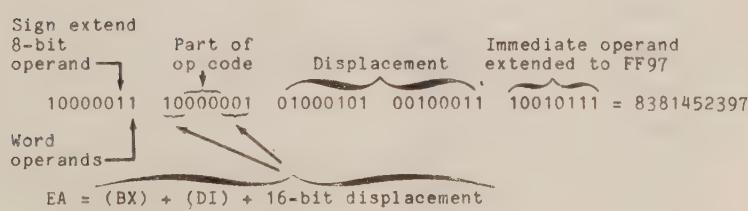
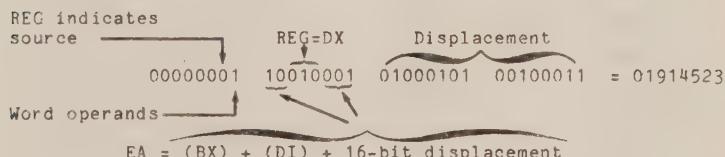
$111 = \text{BH}$. In the second instruction, $D = 0$, which causes $\text{REG} = 111 = \text{BH}$ to be the source. Again $\text{MOD} = 11$ and R/M designates a register, which in this case is $001 = \text{CL}$.

Figure 2-18(a) shows an instruction that uses the relative based indexed addressing mode to add a memory location with a register. From $D = 0$ it is seen that the sum is put in the memory location and $W = 1$ indicates a 16-bit addition. The effective address is found by adding the contents of BX and DI to the 16-bit displacement, which is 2345. If $(\text{BX}) = 0892$ and $(\text{DI}) = 59A3$, then

$$\text{EA} = 0892 + 59A3 + 2345 = 857A$$

Figure 2-18(b) shows a similar instruction except that the source operand is immediate. In this instruction $S = 1$ and $W = 1$, which indicate that the 8-bit immediate operand is sign extended to FF97 before it is added. An equivalent instruction could be constructed of 6 bytes by letting $S:W = 01$ and by including a 16-bit immediate operand containing 97FF.

Figure 2-18 Two examples of the ADD instruction using the relative based indexed addressing mode.



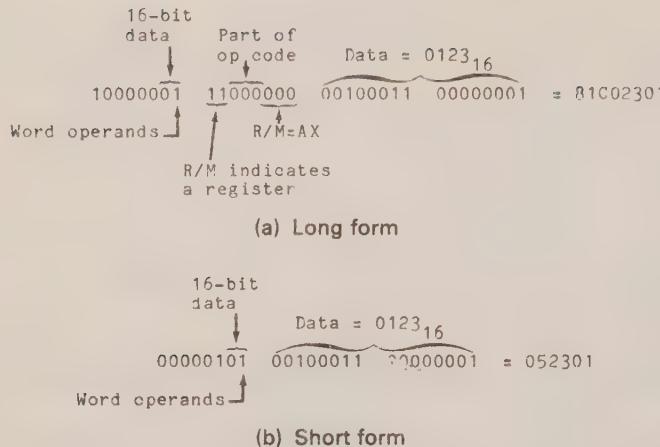


Figure 2-19 Two forms for adding an immediate operand to the AX register.

Last, several instructions have both a long form and a short form. The long and short forms of an instruction for adding an immediate operand to the AX register are given in Fig. 2-19. Because S:W=01 there are 2 bytes of data in the instruction. In the long form AX is explicitly designated by the R/M field and in the short form AX is implied by the op code. Short forms will be discussed more thoroughly in Sec. 3-12.

| Address | Contents | |
|---------|----------|----------------------|
| 00200 | 02 | { First instruction |
| 00201 | CF | |
| 00202 | 01 | |
| 00203 | 91 | { Second instruction |
| 00204 | 45 | |
| 00205 | 23 | |
| 00206 | 83 | |
| 00207 | 81 | |
| 00208 | 45 | { Third instruction |
| 00209 | 23 | |
| 0020A | 97 | |
| 0020B | 05 | |
| 0020C | 23 | { Fourth instruction |
| 0020D | 01 | |
| 0020E | . | |
| . | . | { Fifth instruction |
| . | . | |

Figure 2-20 Sequence of instructions in memory.

Figure 2-20 shows how the instructions in Figs. 2-17(a), 2-18, and 2-19(b) would appear if they were placed in sequence in memory beginning at address 00200. If the contents of CS were 0018, the IP register would increment from 0080 to 008E as these instructions are executed.

2-4 INSTRUCTION EXECUTION TIMING

The execution time of an instruction can be determined by multiplying the number of clock cycles needed to execute the instruction by the clock's period. It can be expressed as the sum of a basic execution time (which varies depending on the instruction and addressing modes) plus the time required to calculate the effective address if a memory operand is involved. The basic execution time assumes that the instruction to be executed has already been prefetched and stored in the instruction queue; otherwise, any additional clock cycles necessary to fetch the instruction must be added. The basic execution times of some sample instructions for the 8086/8088 are shown in Fig. 2-21. A complete listing of execution times is

Figure 2-21 Examples of instruction execution times.

| Instruction | No. of Clock Cycles | No. of Transfers |
|--|---------------------|------------------|
| ADD (addition) or SUB (subtraction) | | |
| Register to register | 3 | 0 |
| Memory to register | 9+EA | 1 |
| Register to memory | 16+EA | 2 |
| Immediate to register | 4 | 0 |
| Immediate to memory | 17+EA | 2 |
| MOV (move) | | |
| Accumulator to memory | 10 | 1 |
| Memory to accumulator | 10 | 1 |
| Register to register | 2 | 0 |
| Memory to register | 8+EA | 1 |
| Register to memory | 9+EA | 1 |
| Immediate to register | 4 | 0 |
| Immediate to memory | 10+EA | 1 |
| Register to segment register | 2 | 0 |
| Memory to segment register | 8+EA | 1 |
| Segment register to register | 2 | 0 |
| Segment register to memory | 9+EA | 1 |
| MUL (unsigned multiply) | | |
| 8-bit register multiplier | 70-77 | 0 |
| 16-bit register multiplier | 118-133 | 0 |
| 8-bit memory multiplier | (76-83)+EA | 1 |
| 16-bit memory multiplier | (124-139)+EA | 1 |
| IMUL (signed multiply) | | |
| 8-bit register multiplier | 80-98 | 0 |
| 16-bit register multiplier | 128-154 | 0 |
| 8-bit memory multiplier | (86-104)+EA | 1 |
| 16-bit memory multiplier | (134-160)+EA | 1 |

| Instruction | No. of Clock Cycles | No. of Transfers |
|--|------------------------------|------------------|
| DIV (unsigned divide) | | |
| 8-bit register divisor | 80-90 | 0 |
| 16-bit register divisor | 144-162 | 0 |
| 8-bit memory divisor | (86-96)+EA | 1 |
| 16-bit memory divisor | (150-168)+EA | 1 |
| IDIV (signed divide) | | |
| 8-bit register divisor | 101-112 | 0 |
| 16-bit register divisor | 165-184 | 0 |
| 8-bit memory divisor | (107-118)+EA | 1 |
| 16-bit memory divisor | (171-190)+EA | 1 |
| Shift and rotate instructions | | |
| Single-bit register | 2 | 0 |
| Variable-bit register | 8+4/bit | 0 |
| Single-bit memory | 15+EA | 2 |
| Variable-bit memory | 20+EA+4/bit | 2 |
| JMP (unconditional branch) | | |
| Short | 15 | 0 |
| Intrasegment direct | 15 | 0 |
| Intersegment direct | 15 | 0 |
| Intrasegment indirect using register mode | 11 | 0 |
| Intrasegment indirect | 18+EA | 1 |
| Intersegment indirect | 24+EA | 2 |
| Conditional branch instructions | | |
| JCXZ | 6 (no branch) 18 (branch) | 0 |
| All other conditional branch instructions | 4 (no branch) 16 (branch) | 0 |

Figure 2-21 (continued)

given in the Appendix. For a memory operand the effective address calculation time (also denoted EA) is determined by the addressing mode as indicated in Figure 2-22.

The third column in Fig. 2-21 indicates the number of memory references required by the instructions. To determine the execution time for an instruction in which a full word memory reference is made, we must also consider the alignment of the operand. If a word operand is located at an odd address, the 8086 must spend two bus cycles, each requiring four clock cycles, in accessing the operand. Therefore, each odd address word reference needs four extra clock cycles. For the 8088, 4 clock cycles should be added for each word transfer because the 8088 can only transfer 1 byte per bus cycle.

Note that some instructions may have several different basic execution times, depending on the addressing modes. The examples in Fig. 2-21 illustrate that for

| EA | No. of Clock Cycles |
|--|---------------------|
| Direct | 6 |
| Register indirect | 5 |
| Register relative | 9 |
| Based indexed (BP)+(DI) or (BX)+(SI) (BP)+(SI) or (BX)+(DI) | 7 8 |
| Based indexed relative (BP)+(DI)+DISP or (BX)+(SI)+DISP (BP)+(SI)+DISP or (BX)+(DI)+DISP | 11 12 |

Figure 2-22 Times needed to calculate the effective address.

a given two-operand instruction, a register-to-register operation is faster than using any other addressing modes, and a register-to-memory operation is slower than a memory-to-register operation because the former requires extra time to store the result into memory. Also, some instructions have variable basic execution times which are data dependent. Typical examples are multiply, divide, shift, and rotate instructions. For conditional branch instructions, two different timing figures are given. The smaller figure is for the case in which the condition is not met, and a branch does not take place, and the larger figure corresponds to when a branch occurs. The latter case includes the flush of the instruction queue and the fetch of the next instruction.

For example, if the clock has a frequency of 5 MHz (its period is 0.2 μ s), then execution times for various forms of the ADD instruction can be computed as follows:

Add register to register (result put in register) requires:

Three clock cycles for either a byte or word operand

$$\text{Time} = 0.6 \mu\text{s}$$

Add memory to register using based indexed relative addressing (result put in register) requires:

$9 + 12 = 21$ cycles for byte or word operation
with word at an even address

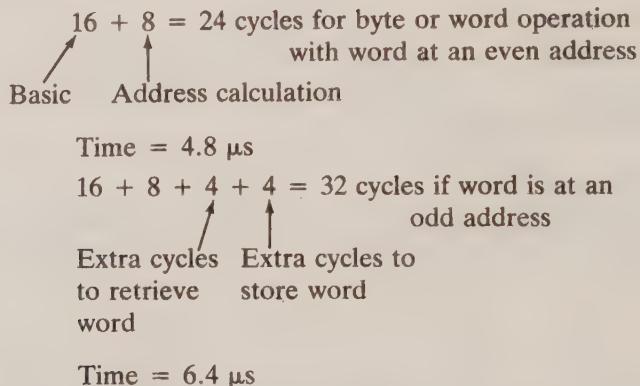
Basic Address calculation

$$\text{Time} = 4.2 \mu\text{s}$$

$9 + 12 + 4 = 25$ cycles if word is at an odd address
Extra cycles

$$\text{Time} = 5.0 \mu\text{s}$$

Add register to memory using based indexed addressing (result to memory) requires:



2-5 THE 8088

The purpose of the 8088 is to provide an 8-bit microprocessor that is fully software compatible with the 8086 (i.e., it has the same instruction set) and can be used in a hardware system that was built for an 8080 or 8085. Like the 8080 and 8085 it has 8 data lines, but its CPU architecture is essentially the same as that of the 8086. Its pin assignments are the same as the 8086 except that the 8088 address pins A15 through A8 are used only for addresses and one of the control pins, the high-byte enable pin, has been changed to a status pin because the 8088 can access only one byte at a time. Since several of the 8088's control pins are different from those of either the 8080 or the 8085, significant changes may need to be made in the bus control logic when putting an 8088 in an 8080- or 8085-based system. One must also take into account the fact that the 8088, like the 8085, differs from the 8080 in that addresses and data are multiplexed. Few or no changes should be required in the memory or I/O subsystem unless the system is to take advantage of the 8088's expanded capacity due to its 20 address lines.

The only internal difference between the 8088 and the 8086 is that the 8088 has only a 4-byte instruction queue instead of a 6-byte queue. The reason for the smaller queue is that the 8088 can only fetch one byte at a time and the longer fetch times mean that the processor cannot fully utilize a 6-byte queue. The prefetching algorithm is different in that instead of waiting for a 2-byte space, the 8088 will fetch an instruction byte when an empty byte in the queue becomes available. The 8088 does retain the segment registers, 20-bit addressing, and the features of the 8086 that support multiprocessing. The ability to manipulate 16-bit operands is also preserved.

BIBLIOGRAPHY

1. Morse, Stephen P., Bruce W. Ravenel, Stanley Mazor, and William P. Pohlman, "Intel Microprocessors—8008 to 8086," *Computer*, no. 10 (October, 1980), 42–60.
2. *MCS-86 User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).

EXERCISES

1. If a physical branch address is 5A230 when (CS) = 5200, what will it be if the (CS) are changed to 7800?
2. Given that the EA of a datum is 2359 and the (DS) = 490B, what is the physical address of the datum?
3. Given that

$$(BX) = 637D \quad (SI) = 2A9B \quad \text{Displacement} = C237$$

determine the effective address (if applicable) resulting from these registers and the addressing mode:

- (a) Immediate.
 - (b) Direct.
 - (c) Register using BX.
 - (d) Register indirect using BX.
 - (e) Register relative using BX.
 - (f) Based indexed.
 - (g) Based indexed relative.
4. Given that
- $$\begin{array}{lll} (IP) = 2BC0 & (CS) = 0200 & \text{Displacement} = 5119 \\ (BX) = 1200 & (DS) = 212A & (224A0) = 0600 \quad (275B9) = 098A \end{array}$$
- find the branch address for a branch instruction that uses:
- (a) Intrasegment direct addressing.
 - (b) Intrasegment indirect addressing which uses the BX register and register addressing.
 - (c) Intrasegment indirect addressing which uses the BX register and register relative addressing.
 5. Give the sum and the flag settings for AF, SF, ZF, CF, OF, and PF after hexadecimally adding 62A0 to each of the following:
 - (a) 1234
 - (b) 4321
 - (c) CFA0
 - (d) 9D60
 6. Give the difference and the flag settings for SF, ZF, CF, OF, and PF after subtracting 4AE0 from each of the following:
 - (a) 1234
 - (b) 5D90
 - (c) 9090
 - (d) EA04
 7. Use the examples in Sec. 2-3-2 to find the machine language code for the instruction that:
 - (a) Adds the contents of BX to the contents of DX and puts the result in DX.
 - (b) Uses registers BX and SI and based indexed addressing to add a byte in memory to the (AL) and put the result in AL.
 - (c) Uses register BX and a displacement of B2 and register relative addressing to add

- a memory location to the (CX), and then puts the result back into the memory location.
- (d) Uses a displacement of 0524 and direct addressing to add the number 2A59 to a memory location and puts the sum back into the memory location.
 - (e) Adds the number B5 to the (AL) and puts the sum in AL. Give both a long form and a short form for this instruction.
8. Assume a 5-MHz clock and use Fig. 2-21 to determine the execution time of an instruction which:
- (a) Adds the (BX) to the (CX).
 - (b) Uses register relative addressing to subtract the contents of a word in memory from the (AX) and puts the result in AX. First assume that the word has an even address and then assume that it has an odd address.
 - (c) Uses direct addressing to move a byte from memory to DX.
 - (d) Performs a conditional branch. First assume that the branch is taken and then assume that it is not taken.

3

Assembler Language Programming

Other than machine language code, which consists of the 0-1 combinations that the computer decodes directly, assembler language code is the most primitive form a program can assume. There are two types of statements in an assembler language. There are *instructions*, which are translated into machine instructions by the assembler, and *directives*, which give directions to the assembler during the assembly process but are not translated into machine instructions.

Although each assembler language instruction produces only one machine language instruction, assembler language instructions are easier to write because acronyms, called *mnemonics*, indicate the type of instruction and character strings, called *identifiers*, represent addresses and, perhaps, numbers. A typical assembler instruction,

ADD AX,COST

adds the contents of the memory location associated with the identifier COST to the register AX. The abbreviation ADD is the instruction mnemonic. An example of a directive is

COST DW ?

which causes the assembler to reserve a word and associate the identifier COST with the word, but does not result in a machine language instruction.

Because an assembler is nothing more than a translating program, the format and syntax of the instructions and directives depend on how the assembler is written, not on the computer. The assembler assumed in this book is the ASM-86 assembler designed by Intel. It is representative of 8086 assemblers and its major features are representative of assemblers in general. For reasons of clarity, some of the

seldom used features of the ASM-86 assembler will not be presented. For programming details, one should refer to the assembler language manual for the system being used.

The features that are available in high-level languages must also be available in assembler languages, even though a single high-level language statement may require several assembler statements to implement it. In addition to including ways of performing branches, loops, I/O, arithmetic operations, and assignments, an assembler needs to provide the equivalents of preassignment, storage allocation, naming of constants, commenting, structuring data, calling procedures, statement functions, and global labeling.

Regardless of the level of the language in which programs are written, they involve inputting, processing, and outputting. Complex programs may require an intermixing of these functions as illustrated by the diagram in Fig. 3-1(a), but simple programs tend to perform them in sequence as shown in Fig. 3-1(b). This chapter will concentrate on simple programs with the inputting, processing, and outputting being performed in order. Because I/O programming is not introduced until Chap. 6, the input and output code will not be given in the examples, but its presence, as well as that of other code, will be indicated by comments. It will be seen later that storage definition and allocation directives are normally placed at the beginning of the source code, and this is also indicated in the comments. The purpose of this chapter is to provide all the material needed to write relatively complex single-module input-process-output programs which are complete, except for the exclusion of the I/O code.

This chapter will proceed by first explaining the assembler instruction format and introducing data transfer and the arithmetic instructions. In Sec. 3-3 the arithmetic instructions are used as a basis for developing the complete structure of a program. Sections 3-4 and 3-5 present the branch and looping instructions, which

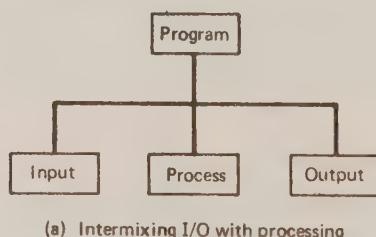
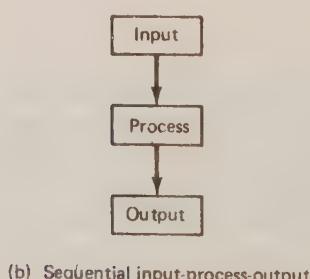


Figure 3-1 General program structures.



allow significantly more complicated programs to be written, and Secs. 3-6 through 3-9 introduce the no operation and halt, flag manipulation, logical, and shift and rotate instructions. Next, directives and the overall structure of a program are considered. Section 3-11 discusses the process of assembling a program and the last section considers the translation of individual assembler language instructions into their machine language formats.

3-1 ASSEMBLER INSTRUCTION FORMAT

The general format of an assembler instruction is

Label: Mnemonic Operand,Operand ;Comments

where the inclusion of spaces is arbitrary, except that at least one space must be inserted if no space would lead to an ambiguity (e.g., between the mnemonic and first operand). Also, there can be no spaces within a mnemonic or identifier and spaces within string constants or comments will be included as space characters. A *label* is an identifier that is assigned the address of the first byte of the instruction in which it appears. The presence of a label in an instruction is optional, but, if present, the label provides a symbolic name that can be used in branch instructions to branch to the instruction. If there is no label, then the colon must be deleted. All instructions must contain a mnemonic. (Mnemonics may be modified by prefixes—these prefixes are introduced as the need arises.) The presence of the operands depends on the instruction. Some instructions have no operands, some have one, and some have two. If there are two operands, they are separated by a comma. The comments field is for commenting the program and may contain any combination of characters. It is optional and if it is deleted the semicolon may also be deleted. A comment may appear on a line by itself provided that the first character on the line is a semicolon. An instruction may be continued to other lines by placing an ampersand at the beginning of each of the continuation lines.

An assembler language instruction must have an operand for each machine language instruction operand and the notation for each operand must be sufficient to indicate the operand's addressing mode. If there are two operands, the destination operand appears first and the source operand second. A typical assembler instruction, one which uses the register relative and register addressing modes to add a memory location to the register AX, is given in Fig. 3-2. When an operand is a word in memory the low-order byte of the word will have the lower address and the high-order byte the higher address, e.g., if the identifier COST is used as a word operand, then COST is associated with the low-order byte and COST + 1 with the high-order byte.

The syntax for the various types of operands is summarized in Fig. 3-3. These forms may be extended by modifiers that are introduced later, but these basic forms are all that are needed at this point. A *variable* is an identifier that is associated with the first byte of a datum. An expression is a concatenation of symbols known

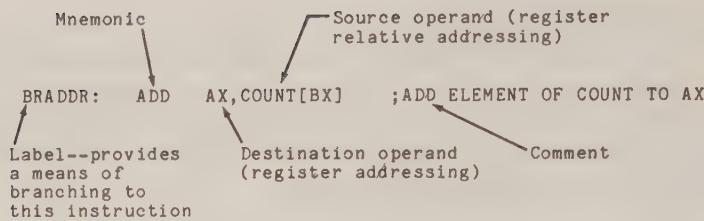


Figure 3-2 Representative assembler language instruction.

as *tokens*, and a constant expression is an expression which can be evaluated to produce a number. The tokens may be variable identifiers or:

Constant—A number whose base is indicated by a suffix as follows:

- B—binary
- D—decimal
- O—octal
- H—hexadecimal

The default is decimal. The first digit in a hexadecimal number must be 0 through 9; therefore, if the most significant digit is a letter (A–F), then it must be prefixed with a 0. Examples are:

$$\begin{aligned}1011_2 &= 1011\text{B} \\223_{10} &= 223\text{D} = 223 \\ \text{B25A}_{16} &= 0\text{B25AH}\end{aligned}$$

String Constant—A character string enclosed in single quotes (').

Arithmetic Operators—The operators "+", "-", "*", and "/".

Logical Operators—The operators "AND", "OR", "NOT", and "XOR" (exclusive OR). The logical operations are performed by putting the operands in binary form and performing the operation on the corresponding pairs of bits.

Subexpressions—An expression that is part of another expression and is delimited from its parent expression by parentheses.

Name—An identifier that represents a constant, string constant, or expression.

(The ASM-86 assembler also permits relational operations, but they are not used in this book.) The order of precedence for operators is the same as it is in high-level languages. An identifier, be it a label, variable, or name, consists of a string of up to 31 characters (actually, it may be longer but the ASM-86 assembler recognizes only the first 31 characters). The characters must be letters, digits, question marks, underscores, or "@" signs except that the first character cannot

| Addressing Mode | Format | Examples |
|------------------------|--|---|
| Data-related | | |
| Immediate | Constant Expression | 10110B |
| | | 529 |
| | | 0A9H |
| | | 'AB' |
| | | YY-XX+5 |
| Direct | Variable \pm Constant Expression | WGT |
| | | CNT-5 |
| | | ARRY+5 |
| Register | Register | AX |
| | | DH |
| Register indirect | [Register] | [BX] |
| Register relative | Variable[Register \pm Constant Exp] or [Register \pm Constant Exp.] | VAR X[BX] MESG[SI+10H] [BX-1] |
| Based indexed | [Base register][Index register] | [BP][DI] |
| Relative based indexed | Var.[Base reg. \pm Const. Exp][Index reg. \pm Const. Exp] or [Base reg. \pm Const. Exp][Index reg. \pm Const. Exp] | E[BX+5][SI-2] DATA[BX][SI] [BP+2][DI-9] |
| Branch-related | | |
| Intrasegment direct | Label \pm Constant Expression | OVFL ERROR+7 |
| Intrasegment indirect | Same as data-related formats except cannot be immediate | |
| Intersegment direct | Label \pm Constant Expression | BRANCH EXIT |
| Intersegment indirect | Same as data related formats except must not be immediate or register | |

Note: For all but the immediate mode the "Constant Expression" may or may not be present.

Figure 3-3 Operand formats for the addressing modes.

be a digit. Identifiers cannot be keywords used by the assembler, such as instruction mnemonics, register names, and special operators. Several complete ADD instructions, along with the machine language instructions that they are translated into, are given in Fig. 3-4.

It should be emphasized that a segment register designation can appear in an operand of only certain instructions. A *segment register* can be explicitly specified in only the MOV, PUSH, and POP instructions. The first of these instructions is discussed in the next section and the other two are defined in Chap. 4. Other instructions, such as the branch instructions, may implicitly reference the segment registers, however. Except for string operations, for a double-operand instruction one of the operands must be a register unless the source operand is immediate. In no case can IP be specified as an operand because only a branch or other control transfer instruction can change the contents of this register.

There are several modifiers, referred to as *attribute operators*, that will be

| Machine Code | Assembler Code |
|---|--|
| 03 C3 | ADD AX,BX ;(BX) ARE ADDED TO AX |
| 81 C3 6D 02 | ADD BX,621 ;ADD 621 TO BX |
| 01 11 | ADD [BX][DI],DX ;ADD (DX) TO THE LOCATION ;WHOSE OFFSET IS THE SUM OF ;(BX) AND (DI) |
| 02 86 2E 02 | ADD AL,NUM[BP+11];ADD TO AL THE CONTENTS OF THE ;LOCATION WHOSE OFFSET IS ;THE SUM OF THE OFFSET OF ;NUM,(BP), AND 11 |
|  <p>Machine code assumes the offset of NUM is 0223 so that $EA = (BP) + 0223 + 000B = (BP) + 022E$</p> | |

Figure 3-4 Typical assembler language instructions.

discussed as the book progresses. One of these operators, the pointer (PTR) operator, needs to be introduced at this point. Many instructions can operate on either bytes or words, depending on whether or not the W-bit is set. Normally, at least one of the operands is such that a byte or a word operation is implied by its notation or by how a memory operand is defined, and the assembler sets or clears the W-bit accordingly, e.g., because of the destination operand the assembler will assume that

ADD AX,[BX]

is a word addition and

ADD AL,[BX]

a byte addition. Also, if appropriate directives are used to define COST to be a word variable and COUNT to be a byte variable, then

INC COST

will be a word operation, and

INC COUNT

will be a byte operation. For some situations, however, it is impossible for the assembler to deduce the operand type. The instruction

INC [BX]

increments the quantity whose address is in BX, but should it increment a byte or a word? One of the purposes of the PTR operator is to specify the length of a quantity in this and other ambiguous situations. It is applied by writing the desired type followed by PTR. For the above INC instruction the PTR operator would be used to modify the operand as follows:

INC BYTE PTR [BX]

if a byte is to be incremented, or

INC WORD PTR [BX]

if a word is to be incremented.

3-2 DATA TRANSFER INSTRUCTIONS

All computers need a variety of instructions solely for moving data, addresses, and immediate operands into register or memory locations. It is the purpose of this section to introduce these instructions for the 8086, but before this is done let us consider the notation that is used throughout this book to define instructions. The symbols and abbreviations to be employed are summarized in Fig. 3-5 (many of them have already been introduced) and are to be used to provide shortened instruction definitions and other descriptive statements. Using this notation the statement

The contents of register AX are replaced by the contents of register AX plus the immediate operand.

Figure 3-5 Glossary of symbols and abbreviations.

| Abbreviation | Meaning | Symbol | Meaning |
|--------------|-----------------------------|--------|----------------------|
| OPR | Operand | ← | Replaced by |
| SRC | Source operand | ↔ | Interchanged with |
| DST | Destination operand | (X) | Contents of X |
| REG | Register | /\ | Logical AND |
| RSRC | Source register | \/ | Logical OR |
| RDST | Destination register | /\̄ | Logical exclusive OR |
| CNT | Count | ̄X | Complement of X |
| DISP | Displacement (general) | | |
| D8 | 8-bit displacement | | |
| D16 | 16-bit displacement | | |
| ADDR | Address | | |
| EA | Effective address | | |
| SEG | Segment | | |
| DATA | Immediate operand (general) | | |
| DATA8 | 8-bit immediate operand | | |
| DATA16 | 16-bit immediate operand | | |
| AX,BX,CX, | 8086 register names | | |
| DX,AL,AH, | | | |
| BL,BH,CL, | | | |
| CH,DL,DH, | | | |
| IP,BP,SP, | | | |
| SI,DI | | | |
| CS,SS,DS | 8086 segment register names | | |
| ES | | | |
| DF,IF,TF | 8086 control flag names | | |
| OF,SF,ZF, | 8086 condition flag names | | |
| AF,PF,CF | | | |

could be written

$$(AX) \leftarrow (AX) + DATA$$

Also, the statement

The contents of BX are replaced by the contents of the memory location whose effective address is the contents of BP plus a displacement.

could be written

$$(BX) \leftarrow ((BP) + DISP)$$

There are four basic 8086 instructions for transferring quantities to and/or from the registers and memory; they are the MOV, LEA, LDS, and LES instructions and are defined in Fig. 3-6. The MOV instruction is for moving a byte or word within the CPU or between the CPU and memory. Depending on the addressing modes it can transfer information from a:

Register to a register.

Immediate operand to a register.

Immediate operand to a memory location.

Memory location to a register.

Register to a memory location.

Register/memory location to a segment register (except CS).

A segment register to a register/memory location.

None of the flags are changed by the execution of a move instruction. Several examples of MOV instructions are given in Fig. 3-7.

Figure 3-6 Data transfer instructions.

| Name | Mnemonic and Format | Description |
|------------------------|---------------------|---|
| Move | MOV DST,SRC | $(DST) \leftarrow (SRC)$ |
| Load effective address | LEA REG,SRC | $(REG) \leftarrow SRC$ |
| Load DS with pointer | LDS REG,SRC | $(REG) \leftarrow (SRC)$ $(DS) \leftarrow (SRC+2)$ |
| Load ES with pointer | LES REG,SRC | $(REG) \leftarrow (SRC)$ $(ES) \leftarrow (SRC+2)$ |
| Exchange | XCHG OPR1,OPR2 | $(OPR1) \leftrightarrow (OPR2)$ |

Flags: None of the flags are affected.

Addressing modes: The destination cannot be immediate and cannot be CS. For the LEA, LDS, and LES instructions REG cannot be a segment register and the source mode cannot be the immediate or register modes. For MOV, unless the source operand is immediate, one of the operands must be a register. For XCHG, at least one of the operands must be a register, but neither operand can be a segment register.

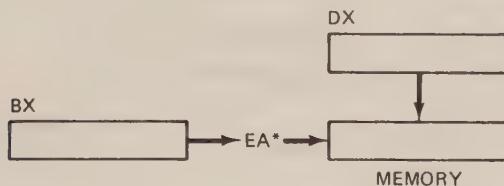
MOV DS, AX ; MOVE (AX) TO DS



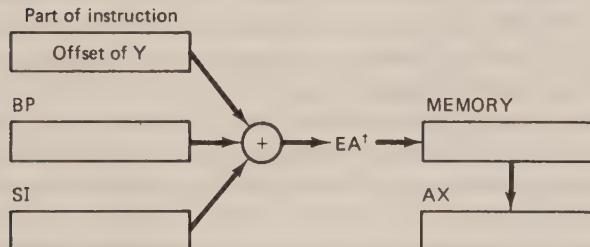
MOV AL, 'E' ; MOVE ASCII 'E' TO AL



MOV [BX], DX ; MOVE (DX) INTO LOCATION INDICATED
; BY (BX)



MOV AX, Y[BP][SI] ; MOVE INTO AX THE CONTENTS OF THE
; LOCATION FOUND BY ADDING (BP), (SI), AND
; THE OFFSET OF Y



^{*}(DS) $\times 16_{10}$ is added to the EA.

[†](SS) $\times 16_{10}$ is added to the EA.

Figure 3-7 Examples of the MOV instruction.

A program sequence for interchanging the contents of the two bytes at the memory locations represented by OPR1 and OPR2 [i.e., (OPR1) \leftrightarrow (OPR2)] is shown in Fig. 3-8(a) and the action taken by the sequence is shown in Fig. 3-8(b). Figure 3-9 shows how the machine code for this segment of assembler code is translated and stored in memory. The indicated instruction addresses assume that the first instruction begins at 0230 and (CS) = 0C30, and the offsets of OPR1 and OPR2 are assumed to be 1200 and 1C20.

A typical LEA instruction is

LEA SI, COL[BX]

Data-related directives, input, and other code
OPR1 and OPR2 are defined as byte variables

| | | |
|-----|---------|----------------|
| MOV | AL,OPR1 | ; (OPR1) TO AL |
| MOV | BL,OPR2 | ; (OPR2) TO BL |
| MOV | OPR2,AL | ; (AL) TO OPR2 |
| MOV | OPR1,BL | ; (BL) TO OPR1 |

Output and other code

(a) Code

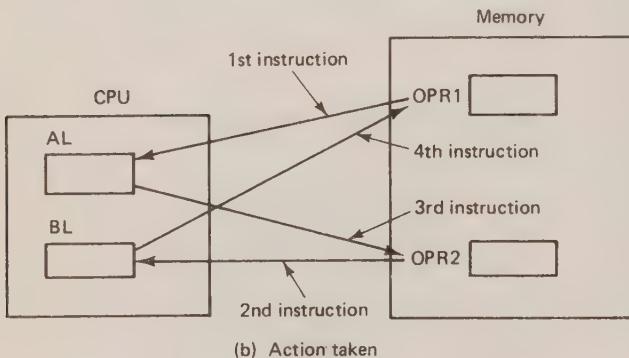


Figure 3-8 Program sequence for interchanging the contents of two locations.

This instruction loads the effective address computed by adding the contents of BX to the offset of COL into SI. The LEA instruction does not affect any of the flags. Its source operand must produce an effective address and it is this address, *not* the contents of this address, that is put into the destination register.

The XCHG instruction is similar to MOV, but instead of duplicating the source in the destination the two quantities are interchanged. Because immediate operands cannot be changed, neither operand in the XCHG instruction can be immediate. In the example shown in Fig. 3-8, the XCHG instruction could be used to replace two of the MOV instructions—see Fig. 3-10. Note that by applying XCHG only three instructions and one register are required. The following sequence of instructions causes the address of ARRAY to be put into BX, 0 to be loaded into SI, and the contents of the word beginning at ARRAY to be transferred to AX:

| | |
|-----|-------------|
| LEA | BX,ARRAY |
| MOV | SI,0 |
| MOV | AX,[BX][SI] |

The LDS and LES instructions are the same except that the former loads the DS register from memory and the latter loads ES from memory. Both instructions also load a second nonsegment register from memory and neither instruction affects the flags. Typical LDS and LES instructions are:

| | |
|-----|--------------------------|
| LDS | SI,STRING_SOURCE_POINTER |
| LES | DI,TABLE[BX] |

where STRING_SOURCE_POINTER and TABLE are double-word variables. These instructions facilitate switching data segments by simultaneously loading a

| Physical Address | Machine Code | |
|------------------|--------------|-------------|
| 0C530 | 8A | |
| 0C531 | 06 | |
| 0C532 | 00 | } |
| 0C533 | 12 | MOV AL,OPR1 |
| 0C534 | 8A | |
| 0C535 | 1E | |
| 0C536 | 20 | } |
| 0C537 | 1C | MOV BL,OPR2 |
| 0C538 | 88 | |
| 0C539 | 06 | |
| 0C53A | 20 | } |
| 0C53B | 1C | MOV OPR2,AL |
| 0C53C | 88 | |
| 0C53D | 1E | |
| 0C53E | 00 | } |
| 0C53F | 12 | MOV OPR1,BL |
| 0C540 | . | |
| . | . | |
| . | . | |
| . | . | |

Figure 3-9 Machine language code for the program sequence given in Fig. 3-8(a).

base or index register and a segment register. The usefulness of their applications will become apparent later.

3-3 ARITHMETIC INSTRUCTIONS

The 8086 instruction set includes instructions for performing arithmetic operations in binary, packed BCD, and unpacked BCD. Figure 3-11 outlines the arithmetic operations that are implemented by 8086 instructions. As indicated in this figure,

Figure 3-10 Interchanging bytes in memory using XCHG.

Data-related directives, input, and other code
OPR1 and OPR2 are defined as byte variables

| | | | |
|------|---------|---|---------------------------|
| MOV | AL,OPR1 | ; | LOAD AL WITH (OPR1) |
| XCHG | AL,OPR2 | ; | EXCHANGE (AL) WITH (OPR2) |
| MOV | OPR1,AL | ; | MOVE (AL) TO (OPR1) |

Output and other code

all four arithmetic operations are available for the binary and unpacked BCD formats, but only instructions that perform addition and subtraction are included for the packed BCD format.

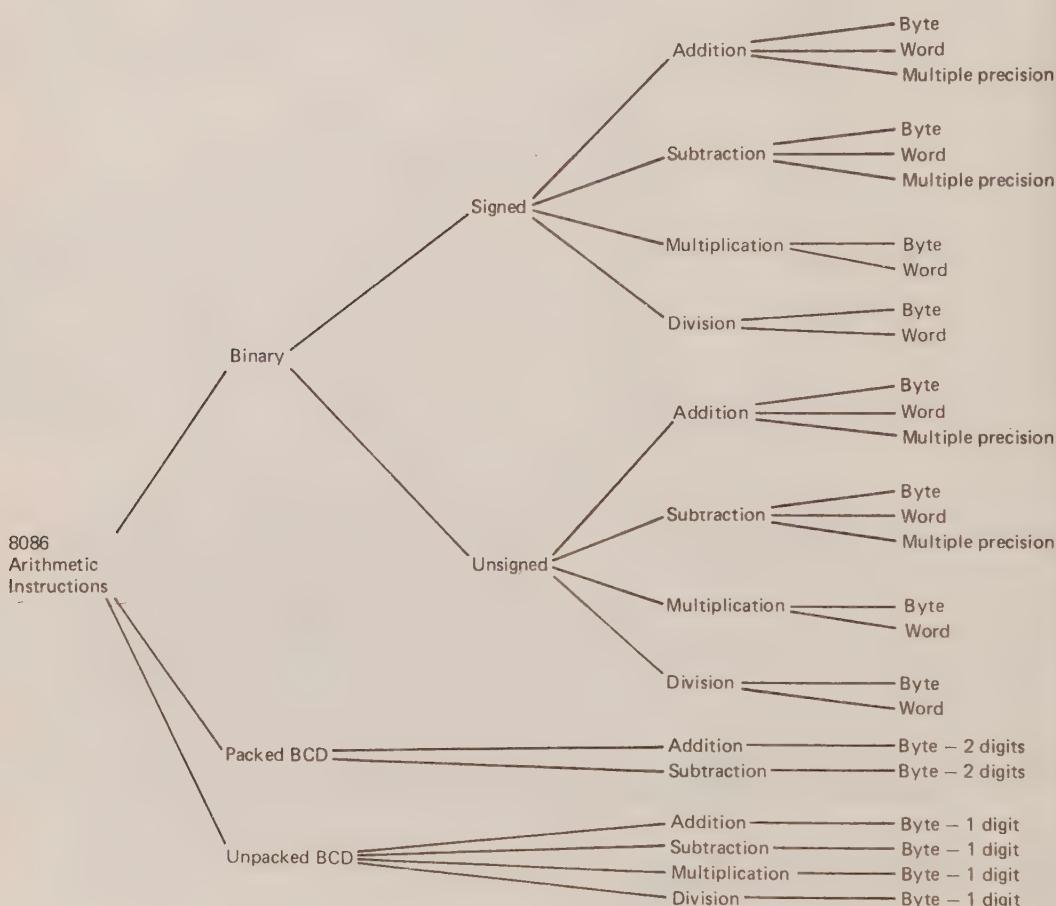


Figure 3-11 Summary of the arithmetic operations that are directly implemented by 8086 instructions.

As mentioned in Sec. 1-2, binary operations can be carried out more quickly than BCD operations; also, a greater range of numbers can be stored in a given number of bits using the binary format. Sixteen bits can store one of 64K binary numbers, only one of 10,000 packed BCD numbers, and only one of 100 unpacked BCD numbers. On the other hand, unpacked BCD numbers can be input and output directly—little or no conversion is necessary. The complete process for converting input ASCII numbers to binary, doing the calculations in binary, re-converting the results to ASCII, and then outputting the results is depicted in Fig. 3-12. A more detailed discussion of this process is given in Sec. 5-5.

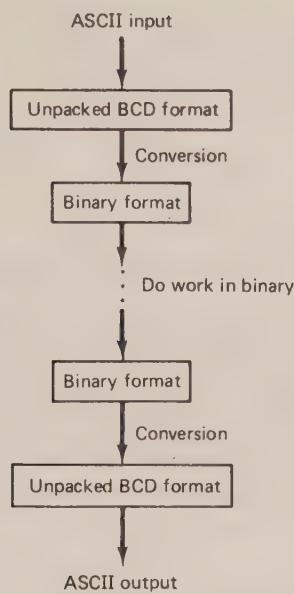


Figure 3-12 Conversion process needed to perform calculations in binary.

3-3-1 Binary Arithmetic

The binary addition and subtraction instructions are defined in Fig. 3-13. All of them affect all of the condition flags. They may use any of the addressing modes for one of the operands, but, except when the source operand is immediate, one of the two operands must be a register. They, as well as the other binary arithmetic instructions, can operate on either bytes or words. A program segment for demonstrating the ADD and SUB instructions is given in Fig. 3-14. It adds the contents of locations X and Y, adds 24 to the sum, subtracts the contents of location Z, and stores the result in W, i.e., it carries out the following calculation and assignment:

$$W \leftarrow X + Y + 24 - Z$$

Figure 3-13 Binary addition and subtraction instructions.

| Name | Mnemonic and Format | Description |
|----------------------|---------------------|---|
| Add | ADD DST, SRC | $(DST) \leftarrow (SRC) + (DST)$ |
| Add with carry | ADC DST, SRC | $(DST) \leftarrow (SRC) + (DST) + (CF)$ |
| Subtract | SUB DST, SRC | $(DST) \leftarrow (DST) - (SRC)$ |
| Subtract with borrow | SBB DST, SRC | $(DST) \leftarrow (DST) - (SRC) - (CF)$ |

Flags: All condition flags are affected.

Addressing modes: Unless the source operand is immediate, one of the operands must be in a register. The other may have any addressing mode.

Data-related directives, input, and other code
 X, Y, W, and Z are defined as word variables

```

MOV     AX,X           ;PUT (X) IN AX
ADD     AX,Y           ;ADD (Y) TO AX
ADD     AX,24          ;ADD 24 TO SUM
SUB    AX,Z            ;SUBTRACT Z FROM X+Y+24
MOV    W,AX            ;STORE RESULT IN W
  
```

Output and other code

Figure 3-14 Single-precision example.

Because a 16-bit word can hold one of only 64K distinct numbers, arithmetic which operates on single words is fairly limited. To extend the range of numbers, words must be concatenated to form groups consisting of multiples of 16 bits. Arithmetic involving one word operands is called *single-precision arithmetic*, involving two words is called *double-precision arithmetic*, and so on. In general, anything higher than single-precision is referred to as *multiple precision*. Double-precision arithmetic, which accommodates

$$2^{32} = 2^2(2^{10})^3 \approx 4 \text{ billion}$$

distinct numbers, is ordinarily adequate when working with integers, and operands consisting of two words are known as *double-word operands*. (Technically, this discussion could apply to bytes, but seldom would multiple-precision byte operations be applied.)

The 8086 binary addition and subtraction can handle at most one word at a time, but, by extending ADD and SUB to ADC and SBB, carries and borrows can automatically be taken into account. Using the ADC and SBB instructions, it is shown below that it is relatively easy to perform multiple-precision arithmetic.

If the 16 bits in a word are treated as a 2's complement number, then the number is said to be *signed*, but if *all* 16 bits are treated as a nonnegative number, then the number is said to be *unsigned*. For example, the 16-bit number C2A3 as a signed quantity is $-15,709_{10}$ but as an unsigned quantity is $49,827_{10}$. Clearly, any number whose MSB is 0 is the same whether it is considered to be signed or unsigned. The logic within the ALU actually executes all additions as if they are unsigned. A 2's complement signed addition is obtained by simply ignoring the carry out of the sixteenth bit. An overflow for an unsigned addition occurs if the sum cannot be expressed with 16 bits, and an overflow for a signed addition occurs if, except for -2^{15} , the magnitude of the sum is too large for 15 bits. However, as noted in Chap. 2, the CPU sets or clears the OF flag as if signed addition were performed. Similar statements can be made for subtraction by recalling that the carry flag is set by a borrow.

To understand how two 16-bit additions can be used to produce a 32-bit sum, consider the following additions:

| Carry = 0 | | Carry = 1 | |
|-----------|------|-----------|------|
| 0015 | 25A0 | 0015 | 65A0 |
| + 0021 | B79E | + 0021 | B79E |
| ————— | DD3E | ————— | 1D3E |

In both cases the correct answer is obtained by adding the lower 16 bits and then adding the upper 16 bits with the carry from the lower 16 bits. Although it is not obvious, this procedure is also valid when negative numbers are permitted and the 2's complement format is used (see Exercise 10). A program sequence making the double-precision computation

$$\text{DPSUM} \leftarrow \text{DP1} + \text{DP2}$$

is given in Fig. 3-15. The sequence assumes that the low-order words of the operands and the sum are in DP1, DP2, and DPSUM, respectively, and the corresponding high-order words are in DP1+2, DP2+2, and DPSUM+2. Note that the low-order words are added using ADD, and the high-order words are added using ADC so that the carry is also included in their sum. Similarly, for higher-order multiple precision the operands should be stored in successive words with the least significant word having the lowest address, the next to the least significant word having the address equal to the lowest address plus two, and so on. Other than the low-order word pair, the word pairs should be added using the ADC instruction.

Data-related directives, input, and other code
DP1, DP2, and DPSUM are defined as word variables

| | | |
|-----|-------------|-------------------------------|
| MOV | AX, DP1 | ;ADD THE LOW-ORDER |
| ADD | AX, DP2 | ;WORDS IN DP1 AND DP2 |
| MOV | DPSUM, AX | ;AND PUT SUM IN DPSUM |
| MOV | AX, DP1+2 | ;ADD THE HIGH-ORDER WORDS |
| ADC | AX, DP2+2 | ;IN DP1+2 AND DP2+2 WITH |
| MOV | DPSUM+2, AX | ;CARRY AND PUT SUM IN DPSUM+2 |

Output and other code

Figure 3-15 Double-precision addition.

For subtraction consider the following 32-bit operations:

$$\begin{array}{r}
 \begin{array}{rr}
 0006 & A92F \\
 - 0004 & 9837 \\
 \hline
 0002 & 10F8
 \end{array}
 &
 \begin{array}{rr}
 0006 & A92F \\
 - 0004 & B837 \\
 \hline
 0002 & F0F8
 \end{array}
 \\[1em]
 \begin{array}{c}
 \text{Borrow } -0 \\
 \hline
 0002
 \end{array}
 &
 \begin{array}{c}
 \text{Borrow } -1 \\
 \hline
 0001
 \end{array}
 \end{array}$$

Once again, the correct answers can be obtained by working with only 16 bits at a time, provided that the borrow from the high-order 16 bits is subtracted from the high-order difference. Recall from the definition of the CF flag that it is set to 1 if the unsigned subtrahend is greater than the unsigned minuend (i.e., a borrow is needed). Although it is not apparent, this rule for setting the CF flag also works when the operands are in 2's complement. The double-precision equivalent of the calculation

$$W \leftarrow X + Y + 24 - Z$$

shown in Fig. 3-14 is given in Fig. 3-16.

Data-related directives, input, and other code
 X, Y, W, and Z are defined as word variables

```

    MOV    AX,X      ; ADD THE DOUBLE PRECISION
    MOV    DX,X+2    ; NUMBER IN X AND X+2
    ADD    AX,Y      ; TO THE DOUBLE PRECISION
    ADC    DX,Y+2    ; NUMBER IN Y AND Y+2
    ADD    AX,24     ; ADD THE DOUBLE PRECISION
    ADC    DX,0      ; NUMBER 24 TO THE RESULT
    SUB   AX,Z      ; SUBTRACT THE DOUBLE PRECISION
    SBB   DX,Z+2    ; NUMBER IN Z AND Z+2 FROM THE SUM
    MOV    W,AX     ; STORE THE OVERALL
    MOV    W+2,DX    ; RESULT IN W AND W+2
  
```

Output and other code

Figure 3-16 Evaluating an expression with double-precision operands.

Signed mixed mode arithmetic involving two different precisions can be performed by extending the sign of the shorter number until the numbers have the same length. The CBW and CWD instructions, defined in Fig. 3-17, are designed to aid the sign-extension process. The CBW instruction extends the sign of the byte in AL to yield an equivalent 2's complement 1-word result in AX, e.g., if the (AL) = B4, then after CBW is executed (AX) will be FFB4. Similarly, CWD extends the sign of the word in AX to DX, thus forming a double word in DX:AX. In both of these instructions the operand is implied by the op code and neither instruction affects the flags. The code needed to add the single-precision number in BX to the double-precision number in DP and DP + 2 and put the result back into DP and DP + 2 is:

```

    MOV    AX,BX
    CWD
    ADD    AX,DP      ;SIGN IS EXTENDED TO DX
    MOV    DP,AX
    ADC    DX,DP+2    ;SIGN OF BX IS ACCOUNTED FOR
    MOV    DP+2,DX
  
```

After the execution of this sequence the old contents of DP and DP + 2 are lost.

Figure 3-18 defines four more binary arithmetic-related instructions. Except for the fact that INC and DEC leave the CF flag unchanged, all of the condition flags are affected by these instructions. The INC, DEC, and NEG instructions

Figure 3-17 Sign-extension instructions.

| Name | Mnemonic and Format | Description |
|-----------------------------|---------------------|-------------------------|
| Convert byte to word | CBW | Extend sign of AL to AH |
| Convert word to double word | CWD | Extend sign of AX to DX |

Flags: None of the flags are affected.

Addressing modes: Operand must be in AL or AX.

| Name | Mnemonic and Format | Description |
|-----------|---------------------|----------------------------|
| Increment | INC OPR | (OPR) \leftarrow (OPR)+1 |
| Decrement | DEC OPR | (OPR) \leftarrow (OPR)-1 |
| Negate | NEG OPR | (OPR) \leftarrow -(OPR) |
| Compare | CMP OPR1, OPR2 | (OPR1)-(OPR2) |

Flags: All condition flags are affected except that INC and DEC do not change the CF flag.

Addressing modes: INC, DEC, and NEG must not use the immediate mode. Unless the OPR2 is immediate, one of the operands for a CMP instruction must be in a register; the other operand may have any addressing mode except that OPR1 cannot be immediate.

Figure 3-18 Single operand binary arithmetic instructions and the compare instruction.

have only one operand; INC adds 1 to the operand, DEC subtracts 1 from the operand, and NEG negates the operand. They may employ any addressing mode other than the immediate mode. INC and DEC are used primarily for counting and indexing and are used extensively in looping. The CMP instruction is identical to SUB except that the result is not stored anywhere. It is used only to set the flags and is normally placed just prior to a conditional jump instruction. It sets the flags according to the relationship between its operands and the branch is taken or not taken depending on the flags, i.e., according to the relative magnitudes of the operands being compared. Applications of the CMP instruction are considered later when jumps are introduced.

The binary multiply and divide instructions are given in Fig. 3-19. The multiply instructions may multiply bytes and produce a word result or multiply words and produce a double-word result. For a signed multiply, if the high-order byte (word) of the product is not simply the sign extension of the low-order byte (word), then both CF and OF are set to 1; otherwise they are both set to 0. For an unsigned multiply, both CF and OF are set to 1 if the high-order byte (or word) of the product is nonzero; otherwise both are cleared. By checking these flags it can be determined whether or not the magnitude of the product was small enough to fit into a single byte (word). The remaining condition flags may be changed, but their contents are undefined, i.e., they are unpredictable. For the signed multiply instruction, IMUL, the sign of the product is determined by the normal sign rules of algebra.

The unsigned multiply instruction, MUL, treats the operands as unsigned numbers and produces an unsigned product. To contrast the IMUL and MUL instructions, suppose that the (AL) = B4, which is -76_{10} as an 8-bit signed integer and 180_{10} as an unsigned integer, and the (BL) = 11, which is 17_{10} both as an 8-bit signed integer and as an unsigned integer. Then

IMUL BL

would cause (AX) = FAF4 = -1292_{10} and (CF) = (OF) = 1 (which implies that

| Name | Mnemonic and Format | Description |
|-------------------|---------------------|---|
| Signed multiply | IMUL SRC | Byte operands $(AX) \leftarrow (AL) * (SRC)$ Word operands $(DX:AX) \leftarrow (AX) * (SRC)$ Product is signed |
| Unsigned multiply | MUL SRC | Same as IMUL except that the operands and product are unsigned |
| Signed divide | IDIV SRC | Byte divisor $(AL) \leftarrow \text{Quotient of } (AX) / (SRC)$ $(AH) \leftarrow \text{Remainder of } (AX) / (SRC)$ Word divisor $(AX) \leftarrow \text{Quotient of } (DX:AX) / (SRC)$ $(DX) \leftarrow \text{Remainder of } (DX:AX) / (SRC)$ Quotient and remainder are signed with the sign of the remainder being the sign of the dividend |
| Unsigned divide | DIV SRC | Same as IDIV except that the operands, quotient, and remainder are unsigned |

Flags: IMUL and MUL set OF and CF to 1 if two bytes (words) are needed for the result; otherwise these flags are reset to 0. The remaining condition flags are undefined. For IDIV and DIV all condition flags are undefined.

Addressing modes: The source operands in these instructions cannot be immediate, but all other addressing modes are permissible. The destination must be AX or DX:AX.

Figure 3-19 Binary multiply and divide instructions.

the result cannot be put in 8 bits). The instruction

MUL BL

would cause the $(AX) = 0BF4 = 3060_{10}$ and the $(CF) = (OF) = 1$ (which again indicates that the product is too large for 8 bits). Note that for both IMUL and MUL, because the space reserved for the product is twice as much as the space occupied by either operand, an overflow is not possible.

The unsigned multiply instruction, MUL, is primarily used for performing multiple-precision multiply operations. To see how a double-precision unsigned multiply is accomplished, consider the two nonnegative double-precision numbers $a2^{16} + b$ and $c2^{16} + d$, where a, b, c , and d represent the coefficients corresponding to the base 2^{16} . Base 2^{16} multiplication is carried out as follows:

$$\begin{array}{r}
 a2^{16} + b \\
 \times \underline{c2^{16} + d} \\
 \hline
 ad2^{32} + bd \\
 ac2^{32} + bc2^{16} \\
 \hline
 ac2^{32} + (ad + bc)2^{16} + bd
 \end{array}$$

Noting that $ad2^{16}$ and $bc2^{16}$ are equivalent to ad and bc followed by sixteen 0 bits and $ac2^{32}$ is ac followed by thirty-two 0 bits, the product can be found by:

1. Computing bd and storing the low-order word as the low-order word of the product.
2. Computing ad , adding the high-order word of bd to the low-order word of ad , and adding the carry to the high-order word of ad .
3. Computing bc and adding it to the result of step 2 using double-precision addition. The carry is stored for use in step 5.
4. Storing the low-order word of the result of step 3 as the next-to-low-order word of the product.
5. Computing ac , adding the high-order word of the result of step 3 to the low-order word of ac , and adding the carries, including the carry from step 3, to the high-order word of ac .
6. Storing the double word resulting from step 5 as the two high-order words of the product.

A program sequence for executing the double-precision calculation

$$\text{DPZ} \leftarrow \text{DPX} * \text{DPY}$$

where DPX and DPY are nonnegative, is given in Fig. 3-20. A multiplication involving negative numbers could be performed by first applying the above algorithm to the magnitudes of the factors and then negating the result if the factors have unlike signs (see Exercise 11).

Figure 3-20 Double-precision multiplication.

Data-related directives, input, and other code
DPX, DPY, and DPZ are defined as word variables

| | | |
|-----|----------|-------------------------------------|
| MOV | AX,DPX | ;MULTIPLY (DPX) |
| MUL | DPY | ;BY (DPY) AND |
| MOV | DPZ,AX | ;STORE LOW-ORDER RESULT IN DPZ |
| MOV | CX,DX | ;AND HIGH-ORDER RESULT IN CX |
| MOV | AX,DPX+2 | ;MULTIPLY (DPX+2) |
| MUL | DPY | ;BY (DPY) AND |
| MOV | BX,DX | ;STORE HIGH-ORDER RESULT IN BX |
| ADD | CX,AX | ;ADD LOW-ORDER RESULT TO CX |
| ADC | BX,0 | ;AND ADD CARRY TO BX |
| MOV | AX,DPX | ;MULTIPLY (DPX) |
| MUL | DPY+2 | ;BY (DPY+2) |
| ADD | CX,AX | ;DOUBLE-PRECISION ADD |
| ADC | BX,DX | ;((DX:AX) AND (BX:CX)) |
| MOV | DPZ+2,CX | ;STORE LOW-ORDER RESULT IN DPZ+2 |
| MOV | CX,0 | ;AND CARRY |
| ADC | CX,0 | ;IN CX |
| MOV | AX,DPX+2 | ;MULTIPLY (DPX+2) |
| MUL | DPY+2 | ;BY (DPY+2) |
| ADD | AX,BX | ;ADD LOW-ORDER OF THIS RESULT TO |
| | | ;HIGH-ORDER OF PREVIOUS RESULT |
| ADC | DX,CX | ;ADD HIGH-ORDER OF THIS RESULT WITH |
| | | ;INTERMEDIATE CARRIES |
| MOV | DPZ+4,AX | ;STORE IN DPZ+4 |
| MOV | DPZ+6,DX | ;AND DPZ+6 |

Output and other code

Neither the IDIV nor DIV instruction leaves meaningful information in any of the condition flags and, as with the multiply instructions, neither permits immediate operands. For division, one must be concerned with both the quotient and remainder. Because the space reserved for the dividend is twice as long as that reserved for the quotient, an overflow can occur, but the OF flag does *not* indicate the overflow, and the quotient and remainder are undefined. (An overflow, however, will cause an interrupt—see Chap. 4.) For the signed division instruction, IDIV, the sign of the quotient is determined by the sign rule of algebra and the remainder is given the same sign as the dividend. (Therefore, regardless of the signs, the product of the divisor and quotient plus the remainder is always equal to the dividend.) To compare the signed division instruction, IDIV, to DIV, assume that (AX) = 0400 and that (BL) = B4 (which is -76_{10} if signed and 180_{10} if unsigned). Then

IDIV BL

would cause

$$\begin{aligned} (\text{AH}) &= 24 = 36_{10} = \text{remainder} \\ (\text{AL}) &= F3 = -13_{10} = \text{quotient} \end{aligned}$$

and

DIV BL

would result in

$$\begin{aligned} (\text{AH}) &= 7C = 124_{10} = \text{remainder} \\ (\text{AL}) &= 05 = 5_{10} = \text{quotient} \end{aligned}$$

Multiple-precision division depends on the lengths of both the divisor and dividend. To develop an algorithm for performing an unsigned division of one word into three words, let us assume that

$$\text{divisor} = a \quad \text{dividend} = b2^{32} + c2^{16} + d$$

where a , b , c , and d are the base 2^{16} coefficients. The normal division algorithm in base 2^{16} is shown below.

$$\begin{array}{r} q_2 2^{32} + q_1 2^{16} + q_0 \\ a) \underline{b 2^{32} + c 2^{16} + d} \\ r_2 2^{32} + \underline{c 2^{16}} \\ \hline r_1 2^{16} + d \\ \hline r_0 \end{array}$$

where q_2 and r_2 are the quotient and remainder resulting from b/a ; q_1 and r_1 are the quotient and remainder resulting from $(r_2 2^{16} + c)/a$; and q_0 and r_0 are the quotient and remainder produced by $(r_1 2^{16} + d)/a$. At the start of the process, b would be put in AX and DX would be zeroed. Because b is only one word long,

q_2 would fit into one word and an overflow could not be caused by the first division. Because r_2 and r_1 , which would reside in DX after the first and second divisions, respectively, would be less than a , an overflow could not result from either of the last two divisions. This process could easily be extended to dividing one word into n words. The reader is asked to implement this algorithm for unsigned integers in Exercise 12. For signed numbers, multiple precision can be handled by dividing the magnitudes and assigning the sign according to the sign rule of algebra—see Exercise 13.

Because DIV permits a divisor whose length is at most one word, it is difficult to use DIV to perform divisions involving divisors containing more than one word. It may be necessary to perform such divisions at the bit level using the shift and rotate instructions discussed later in Sec. 3-9. However, a procedure that uses DIV to divide a double word into a double word is outlined in Exercise 22.

An example program sequence which involves a combination of single-precision binary arithmetic operations is shown in Fig. 3-21. The sequence performs the calculation

```
(AX) ← Quotient of ((V - (X*Y+Z-540))/X)
(DX) ← Remainder
```

3-3-2 Packed BCD Arithmetic

Packed BCD numbers are stored two digits to a byte, in 4-bit groups referred to as *nibbles*. The ALU is capable of performing only binary addition and subtraction, but by adjusting the sum or difference the correct result in packed BCD format can be obtained. The correction rule for addition is:

If the addition of any two digits results in a binary number between 1010 and 1111, which are not valid BCD digits, or there is a carry into the next digit, then 6 (0110) is to be added to the current digit.

Figure 3-21 Example involving all four arithmetic operations.

Data-related directives, input, and other code
X, Y, Z, and V are defined as word variables

| | | |
|------|--------|-------------------------|
| MOV | AX,X | ;MULTIPLY (X) |
| IMUL | Y | ;BY (Y) AND |
| MOV | CX,AX | ;STORE PRODUCT |
| MOV | BX,DX | ;IN BX:CX |
| MOV | AX,Z | ;ADD SIGN-EXTENDED |
| CWD | | ; (Z) TO THE |
| ADD | CX,AX | ;PRODUCT |
| ADC | BX,DX | ;IN BX:CX |
| SUB | CX,540 | ;SUBTRACT 540 |
| SBB | BX,0 | ;FROM BX:CX |
| MOV | AX,V | ;THEN SUBTRACT |
| CWD | | ; (BX:CX) FROM |
| SUB | AX,CX | ;SIGN-EXTENDED |
| SBB | DX,BX | ; (V) AND DIVIDE BY (X) |
| IDIV | X | ;LEAVE QUOTIENT IN AX |
| | | ;AND REMAINDER IN DX |

Output and other code

For example:

$$\begin{array}{r} \text{Carry from } \rightarrow 0 \\ \text{previous digit } 7 \\ + \underline{6} \\ \hline 13 \end{array} \quad \begin{array}{r} 0 \\ 0111 \\ + 0110 \\ \hline 1101 \end{array}$$

Carry to next digit \rightarrow 1 $\begin{array}{r} 110 \\ 0011 \end{array} \leftarrow \text{Add 6 because 1101 is not a valid BCD digit}$

$$\begin{array}{r} \text{Carry from } \rightarrow 1 \\ \text{previous digit } 9 \\ + \underline{9} \\ \hline 19 \end{array} \quad \begin{array}{r} 1 \\ 1001 \\ + 1001 \\ \hline 0011 \end{array}$$

Carry to next digit \rightarrow 1 $\begin{array}{r} 110 \\ 1001 \end{array} \leftarrow \text{Add 6 because of carry to next digit}$

Essentially, the rule is needed to "skip over" the six bit combinations that are unused by the BCD format whenever such a skip is warranted.

When utilizing binary addition to perform a packed BCD addition on the 8086, only byte addition can be used. Because a byte contains two nibbles (digits), the AF flag is needed to indicate the carry out of bit 3, the MSB of the low-order nibble. The CF flag, of course, indicates the carry out of the high-order nibble. By executing a binary addition on two byte operands and following it with an instruction that, after examining the AF and CF flags for ones and the nibbles for invalid combinations, appropriately adds 6s to adjust the sum, the correct answer in packed BCD can be obtained.

A similar rule for subtraction could be developed, which involves borrows instead of carries, and a special instruction for adjusting differences could be used with subtract instructions to yield packed BCD results. The special addition and subtraction instructions are DAA and DAS and are defined in Fig. 3-22. While DAA adds 6s where they are needed, DAS causes 6s to be subtracted. To use these instructions the sum or difference must be in the AL register.

Shown in Fig. 3-23 is a program sequence for carrying out the calculation

$BCD3 \leftarrow BCD1 + BCD2$

Figure 3-22 Packed BCD adjust instructions.

| Name | Mnemonic and Format | Description |
|-----------------------------------|---------------------|--|
| Decimal adjust for addition | DAA | (AL) \leftarrow Sum in AL adjusted to packed BCD format |
| Decimal adjust for subtraction | DAS | (AL) \leftarrow Difference in AL adjusted to packed BCD format |

Flags: OF flag is undefined and all other conditional flags are affected.

Addressing mode: Operates only on the AL register.

| Instruction | Action | Contents of AL CF AF |
|---------------|------------------------------|-------------------------|
| MOV AL,BCD1 | (AL) \leftarrow 34 | 34 -- -- |
| ADD AL,BCD2 | (AL) \leftarrow 34+89- | BD 0 0 |
| DAA | ADJUST | 23 1 * |
| MOV BCD3,AL | (BCD3) \leftarrow 23 | 23 1 * |
| MOV AL,BCD1+1 | AL \leftarrow 18 | 18 1 * |
| ADC AL,BCD2+1 | (AL) \leftarrow 18+27+(CF) | 40 0 1 |
| DAA | ADJUST | 46 0 * |
| MOV BCD3+1,AL | (BCD3+1) \leftarrow 46 | 46 0 * |

*Not important **Figure 3-23** Packed BCD addition.

where BCD1 and BCD2 are defined as byte variables that store four-digit 10's complement numbers with the most significant digits in the high-order bytes of their words. The columns on the right assume that (BCD1) = 1834 and (BCD2) = 2789 and gives the contents of AL, CF, and AF after the execution of each instruction. Figure 3-24 gives a sequence for executing the four-digit 10's complement calculation

$$\text{BCD3} \leftarrow \text{BCD1} - \text{BCD2}$$

Assuming that (BCD1) = 1234 and (BCD2) = 4612, once again the columns on the right give the successive contents of AL, CF, and AF.

There are no instructions for performing packed BCD multiplication or division. The reason is that there are two digits per byte and there are no simple algorithms for handling two digits at a time. Repeated addition (subtraction) could be used if the multiplier (quotient) is known to be small, but, in general, the most practical procedure would be to convert the operands to their binary equivalents and use binary arithmetic.

3-3-3 Unpacked BCD Arithmetic

In unpacked BCD there is only one digit per byte and, because of this, unpacked multiplication and division can be done. Unpacked BCD addition, subtraction, and multiplication are accomplished in much the same way as packed BCD addition and subtraction in that the binary operations act on single bytes and the results are adjusted. For division the adjustment is done before the binary division. For

| Instruction | Action | Contents of AL CF AF |
|---------------|------------------------------|-------------------------|
| MOV AL,BCD1 | (AL) \leftarrow 34 | 34 -- -- |
| SUB AL,BCD2 | (AL) \leftarrow 34-12 | 22 0 0 |
| DAS | ADJUST | 22 0 * |
| MOV BCD3,AL | (BCD3) \leftarrow 22 | 22 0 * |
| MOV AL,BCD1+1 | (AL) \leftarrow 12 | 12 0 * |
| SBB AL,BCD2+1 | (AL) \leftarrow 12-46-(CF) | CC 1 1 |
| DAS | ADJUST | 66 1 * |
| MOV BCD3+1,AL | (BCD3+1) \leftarrow (AL) | 66 1 * |

Figure 3-24 Packed BCD subtraction.

*Not important

Result = 6622 = -3378 in 10's complement

| Name | Mnemonic and Format | Description |
|---|---------------------|--|
| Unpacked BCD adjust for addition | AAA | $(AL) \leftarrow$ Sum in AL adjusted to unpacked BCD format $(AH) \leftarrow (AH) +$ carry from adjustment |
| Unpacked BCD adjust for subtraction | AAS | $(AL) \leftarrow$ Difference in AL adjusted to unpacked BCD format $(AH) \leftarrow (AH) -$ borrow from adjustment |
| Unpacked BCD adjust for multiplication | AAM | $(AX) \leftarrow$ Product in AL adjusted to unpacked BCD format with AH containing high-order digit |
| Unpacked BCD adjust for division | AAD | $(AL) \leftarrow 10^*(AH)+(AL)$ $(AH) \leftarrow 0$ |

Note: The high-order nibble of the operands should be zero.

Flags: For AAA and AAS the flags AF and CF are set if there is an adjustment and the remaining flags are undefined. AAM and AAD set PF, SF, and ZF according to their rules and leave OF, AF, and CF undefined.

Addressing mode: Operand is in AL or AX register.

Figure 3-25 Unpacked BCD adjust instructions.

all four of the arithmetic operations the high-order nibbles of the operands should be zero (even though, for addition and subtraction, erroneous results may not be produced, the high-order nibble of the result may be meaningless). The adjustment instructions for the four arithmetic operations are given in Fig. 3-25. A program sequence for executing

$$(DX) \leftarrow UP1 + UP2 - UP3$$

on two-digit 10's complement numbers is given in Fig. 3-26. Extending unpacked BCD addition and subtraction to numbers with more digits is straightforward and is left to the exercises (see Exercise 18).

Figure 3-26 Example involving unpacked BCD addition and subtraction.

Data-related directives, input, and other code
UP1, UP2, and UP3 are defined as byte variables

```

MOV    AL, UP1      ;ADD LOW-ORDER
ADD    AL, UP2      ;DIGITS AND
AAA
MOV    DL, AL       ;PUT SUM IN DL
MOV    AL, UP1+1    ;ADD HIGH-ORDER
ADC    AL, UP2+1    ;DIGITS WITH CARRY
AAA
XCHG   AL, DL      ;EXCHANGE AL AND DL
SUB    AL, UP3      ;SUBTRACT LOW-ORDER DIGIT
AAS
XCHG   AL, DL      ;EXCHANGE AL AND DL
SBB    AL, UP3+1    ;SUBTRACT HIGH-ORDER DIGIT
AAS
MOV    DH, AL       ;MOVE AL TO DH

```

Output and other code

For unpacked BCD multiplication and division the high-order nibbles of the operands *must* be 0. The AAM instruction replaces (AH) by the quotient of (AL)/10 and the (AL) by the remainder. AAD multiplies (AH) by 10, adds the product to the (AL), and then zeros AH. (It is left to the reader to verify that these are the correct actions—see Exercises 19 and 20.)

The unsigned multiplication of two-digit numbers proceeds according to the algorithm

$$\begin{array}{r}
 a_1 a_0 \\
 \times b_1 b_0 \\
 \hline
 c_{02} c_{01} c_{00} \\
 \hline
 c_{12} c_{11} c_{10} \\
 \hline
 c_3 \ c_2 \ c_1 \ c_0
 \end{array}$$

where A, B, and C are defined as byte variables. An implementation of this algorithm to calculate

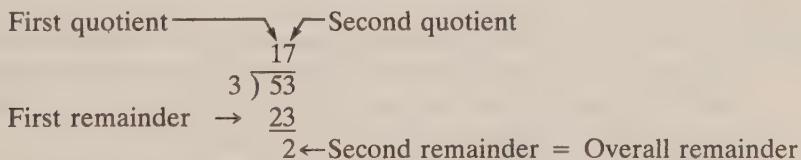
$$C \leftarrow A * B$$

is shown in Fig. 3-27 along with the step-by-step actions and results corresponding to (A) = 34 and (B) = 56.

Division by a single digit follows the normal division pattern and the computation

$$\begin{aligned}
 C &\leftarrow \text{Quotient of } B/A \\
 R &\leftarrow \text{Remainder}
 \end{aligned}$$

where A is a one-digit number and B is a two-digit number, is given in Fig. 3-28. Also shown in the figure are the exact actions and the successive contents of the AX register for the case when the (A) = 3 and the (B) = 53. The process is equivalent to



The division begins by dividing 5 by 3 using DIV. DIV puts the remainder in AH and the quotient in AL. After storing the quotient, which is the most significant digit of the unpacked BCD quotient, and moving the next digit of the dividend to AL, AAD is applied to cause $10 * (\text{AH}) + (\text{AL})$ to be put into AL. Then DIV is executed again to produce the least significant unpacked BCD digit and the overall remainder. By repeatedly applying AAD and DIV and storing the quotient, a dividend containing an arbitrary number of digits could be divided by a single digit. If the divisor contains more than one digit, the algorithm becomes considerably

| Instruction | Action | Contents of AX |
|----------------------|------------------|----------------|
| MOV AL,A | (AL) ← 4 | -- 04 |
| MUL B | (AX) ← 4*6 | 00 18 |
| AAM | ADJUST | 02 04 |
| MOV WORD PTR CO,AX | (CO+1:CO) ← 24 | 02 04 |
| MOV AL,A+1 | (AL) ← 3 | 02 03 |
| MUL B | (AX) ← 3*6 | 00 12 |
| AAM | ADJUST | 01 08 |
| ADD AL,CO+1 | (AL) ← 2+8 | 01 0A |
| AAA | ADJUST | 02 00 |
| MOV WORD PTR CO+1,AX | (CO+2:CO+1) ← 20 | 02 00 |
| MOV AL,A | (AL) ← 4 | 02 04 |
| MUL B+1 | (AX) ← 4*5 | 00 14 |
| AAM | ADJUST | 02 00 |
| MOV WORD PTR C1,AX | (C1+1:C1) ← 20 | 02 00 |
| MOV AL,A+1 | (AL) ← 3 | 02 03 |
| MUL B+1 | AX ← 3*5 | 00 0F |
| AAM | ADJUST | 01 05 |
| ADD AL,C1+1 | (AL) ← 5+2 | 01 07 |
| AAA | ADJUST | 01 07 |
| MOV WORD PTR C1+1,AX | (C1+2:C1+1) ← 17 | 01 07 |
| MOV AL,CO | (AL) ← 4 | 01 04 |
| MOV C,AL | (C) ← 4 | 01 04 |
| MOV AL,CO+1 | (AL) ← 0 | 01 00 |
| ADD AL,C1 | (AL) ← 0+0 | 01 00 |
| AAA | ADJUST | 01 00 |
| MOV C+1,AL | (C+1) ← 0 | 01 00 |
| MOV AL,CO+2 | (AL) ← 2 | 01 02 |
| ADC AL,C1+1 | (AL) ← 2+7+(CF) | 01 09 |
| AAA | ADJUST | 01 09 |
| MOV C+2,AL | (C+2) ← 9 | 01 09 |
| MOV AL,0 | (AL) ← 0 | 01 00 |
| ADC AL,C1+2 | (AL) ← 0+1+(CF) | 01 01 |
| AAA | ADJUST | 01 01 |
| MOV C+3,AL | (C+3) ← 1 | 01 01 |

Figure 3-27 Two-digit unpacked BCD multiplication.

more complicated, and it is suggested that the operands be converted to binary and binary division be employed.

3-4 BRANCH INSTRUCTIONS

A branch instruction transfers control from the normal sequence of instructions by putting the address of the instruction to be branched to in the program counter. If the branch is conditional, then whether or not the contents of the program counter are replaced with the branch address depends on the flags. For the 8086, since a physical address is comprised of both an offset and a segment address, some branch (or jump) instructions modify both the IP and CS registers while others modify only the IP register. (The term "branch" is used in this book because it is prevalent and is found in high-level language as well as assembler language dis-

| Instruction | Action | Contents of AX |
|-------------|-----------|----------------|
| MOV AH,0 | (AH) ← 0 | 00 -- |
| MOV AL,B+1 | (AL) ← 5 | 00 05 |
| DIV A | DIVIDE | 02 01 |
| MOV C+1,AL | (C+1) ← 1 | 02 01 |
| MOV AL,B | (AL) ← 3 | 02 03 |
| AAD | ADJUST | 00 17 |
| DIV A | DIVIDE | 02 07 |
| MOV C,AL | (C) ← 7 | 02 07 |
| MOV R,AH | (R) ← 2 | 02 07 |

Figure 3-28 Unpacked BCD division.

| Distance (in decimal) | D8 (in hexadecimal) | Branch Address |
|-----------------------|---------------------|----------------|
| -128 | 80 | (IP)-128 |
| . | . | . |
| . | . | . |
| -2 | FE | (IP)-2 |
| -1 | FF | (IP)-1 |
| 0 | 00 | (IP) |
| 1 | 01 | (IP)+1 |
| 2 | 02 | (IP)+2 |
| . | . | . |
| . | . | . |
| 127 | 7F | (IP)+127 |

Figure 3-29 Correspondence between branch distances, values of D8, and branch addresses.

cussions; however, Intel manuals tend to use the term “jump” and have chosen to begin their branch instruction mnemonics with the letter “J”. Those instructions that change the contents of the CS register are referred to as intersegment branches and the remaining branch instructions are called intrasegment branches. Only unconditional branch instructions can cause intersegment branches.

3-4-1 Conditional Branch Instructions

All conditional branch instructions have the following 2-byte machine code format:



where the second byte gives an 8-bit signed (2's complement) displacement relative to the address of the next instruction in sequence. The effective branch address is determined by sign extending D8 and adding it to the contents of IP *after* IP has been incremented to point to the next instruction. Therefore, a negative D8 means a backward branch from the next instruction and a positive D8 means a forward branch from the next instruction. The limitation of D8 to 8 bits restricts the distance between the address of the next instruction and the branch address to the range -128 to 127 bytes. Figure 3-29 gives the value of D8 for the various branch distances.

As an example of how the assembler determines the value of D8, consider the following sequence:

| | | | |
|------|--------|-----|-----------|
| 0050 | AGAIN: | INC | CX |
| 0052 | | ADD | AX,[BX] |
| 0054 | | JNS | AGAIN |
| 0056 | NEXT: | MOV | RESULT,CX |

where the column on the left gives the effective address of the first byte of each instruction. Because

| | |
|--------|---------------------------------------|
| 0050 | Effective branch address |
| - 0056 | (IP) when JNS branch decision is made |

the assembler will set the value of D8 to FA.

An assembler language conditional branch instruction has the form:

Operation Operand

where the operand determines the instruction to which the branch is made. The operand may be a label or a label plus or minus an expression that evaluates to a constant. The operation mnemonic begins with "J" (for jump) and its remaining letters indicate the condition under which the branch is made. The conditional branch instructions are summarized in Fig. 3-30. Some of them have alternate

Figure 3-30 Conditional branch instructions.

| Name | Mnemonic and Format | Alternate Mnemonic | Test Condition* |
|---|---------------------|--------------------|------------------------------|
| Branch on zero, or equal | JZ OPR | JE | ZF = 1 |
| Branch on nonzero, or not equal | JNZ OPR | JNE | ZF = 0 |
| Branch on sign set | JS OPR | | SF = 1 |
| Branch on sign clear | JNS OPR | | SF = 0 |
| Branch on overflow | JO OPR | | OF = 1 |
| Branch on no overflow | JNO OPR | | OF = 0 |
| Branch on parity set, or even parity | JP OPR | JPE | PF = 1 |
| Branch on parity clear, or odd parity | JNP OPR | JPO | PF = 0 |
| Branch on below, or not above or equal (unsigned) | JB OPR | JNAE, JC | CF = 1 |
| Branch on not below, or above or equal (unsigned) | JNB OPR | JAE, JNC | CF = 0 |
| Branch on below or equal, or not above (unsigned) | JBE OPR | JNA | CF \vee ZF = 1 |
| Branch on not below or equal, or above (unsigned) | JNBE OPR | JA | CF \vee ZF = 0 |
| Branch on less, or not greater or equal (signed) | JL OPR | JNGE | SF \neq OF = 1 |
| Branch on not less, or greater or equal (signed) | JNL OPR | JGE | SF \neq OF = 0 |
| Branch on less or equal, or not greater (signed) | JLE OPR | JNG | (SF \neq OF) \vee ZF = 1 |
| Branch on not less or equal, or greater (signed) | JNLE OPR | JG | (SF \neq OF) \vee ZF = 0 |

*If the test condition is met (IP) \leftarrow (IP) + sign extended D8; otherwise (IP) are unchanged and the program continues in sequence.

-Flags: No flags are affected.

Addressing mode: Mode is relative to IP. OPR must represent a label that is within -128 to 127 bytes of the instruction following the branch instruction.

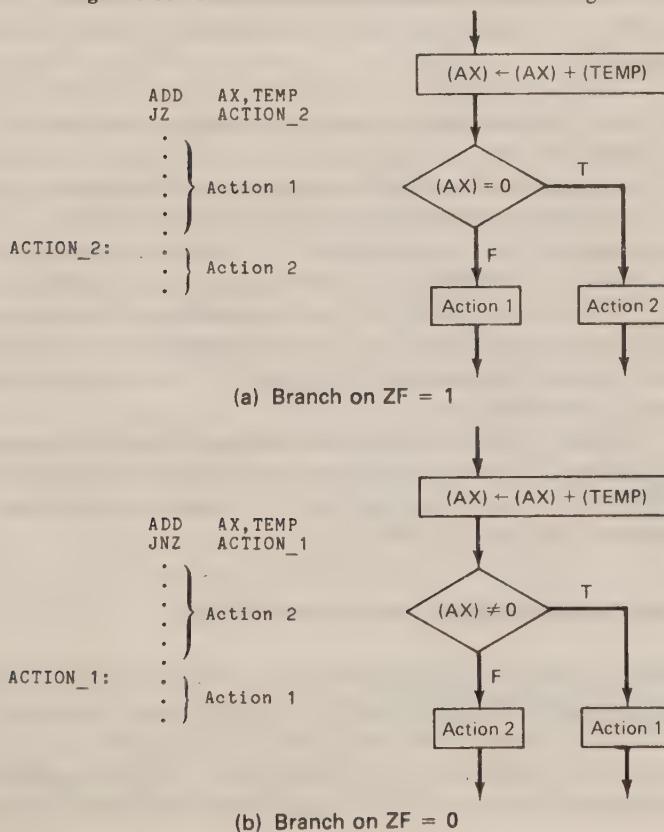
mnemonics, e.g., JNL (branch on not less than) is equivalent to JGE (branch on greater than or equal to).

The first ten instructions in Fig. 3-30 base the branch decision on the state of a single flag. Five of them are designed to branch if the corresponding flag is 1 and to continue in sequence if the flag is 0. Conversely, the other five branch on 0 and do not branch on 1. These ten instructions may be used in any situation which requires that one of two actions be taken depending on the state of one flag. For example, JZ could be applied after an ADD instruction to cause a branch if the sum is 0, or JNZ could be used if the branch is to be taken in the event of a nonzero sum. Two equivalent structures involving JZ and JNZ are depicted in Fig. 3-31.

The last eight instructions given in Fig. 3-30 are intended to be used with CMP instructions which subtract their second operands from their first operands. By comparing two operands and then branching according to the appropriate flag settings, decisions based on the relation operators $<$, $>$, \leq , \geq , \neq , and $=$ can be made. The instructions

| | |
|-----|-----------|
| CMP | TOTAL,100 |
| JGE | EXIT |

Figure 3-31 Conditional branches based on the ZF flag.



Data-related directives, input, and other code
 X , Y , and $RESULT$ are defined as word variables

| | | |
|---------|---------------|-------------------------------|
| MOV | AX,X | ;MOVE (X) TO AX AND BRANCH TO |
| CMP | AX,50 | ;TOO HIGH IF (X) IS |
| JG | TOO HIGH | ;GREATER THAN 50 |
| SUB | AX,Y | ;OTHERWISE SUBTRACT (Y) |
| JO | OVERFLOW | ;AND BRANCH ON OVERFLOW |
| JNS | NONNEG | ;IF NO OVERFLOW |
| NEG | AX | ;TAKE ABSOLUTE VALUE |
| NONNEG: | MOV RESULT,AX | ;AND STORE IT IN RESULT |

Output and other code

Figure 3-32 Sequence invoking several conditional branches.

would result in a branch to EXIT if the contents of TOTAL are greater than or equal to 100. These eight instructions can be broken into two groups of four, with the first four corresponding to unsigned comparisons and the last four being used for signed comparisons. The terms "below" (B) and "above" (A) are associated with the unsigned instructions and "less" (L) and "greater" (G) are used for describing the signed instructions. Also, the JE and JNE instructions (which are equivalent to the JZ and JNZ instructions, respectively) are used in conjunction with the compare instructions to test for equality and inequality. In testing for equality or inequality it makes no difference whether the quantities are signed or unsigned.

The fact that $CF = 1$ is the correct test condition for the unsigned branch on below, JB, instruction is seen by recalling that the CMP instruction performs a subtraction and a subtraction causes the CF flag to be set if and only if an unsigned borrow is needed (see the discussion on binary arithmetic given in Sec. 3-3-1). The test condition for branch on not below is clearly $CF = 0$ since it must be the complement of the test condition for branch on below. It is also easy to see that the conditions for branch on below or equal and branch on not below or equal are $CF \vee ZF = 1$ and $CF \vee ZF = 0$, respectively. The test conditions for the signed instructions are not so easily verified, but can be proven by considering all possible cases for the operands being compared. This is rather laborious and we will not pursue the matter here.

Figure 3-32 gives a program sequence containing several conditional branch instructions. The sequence branches to TOO_HIGH if $X > 50$ and to OVERFLOW if the signed subtraction $X - Y$ causes an overflow. Otherwise, $|X - Y|$ is computed and put in RESULT.

3-4-2 Unconditional Branch Instructions

There are five unconditional branch instructions. Their machine code formats are summarized in Fig. 3-33 and the assembler code instructions are defined in Fig. 3-34. Three of these instructions are for branching to a point in the current segment and two of them are for branching to a different segment. The former type of branching is referred to as an *intrasegment branch* and the latter as an *intersegment*

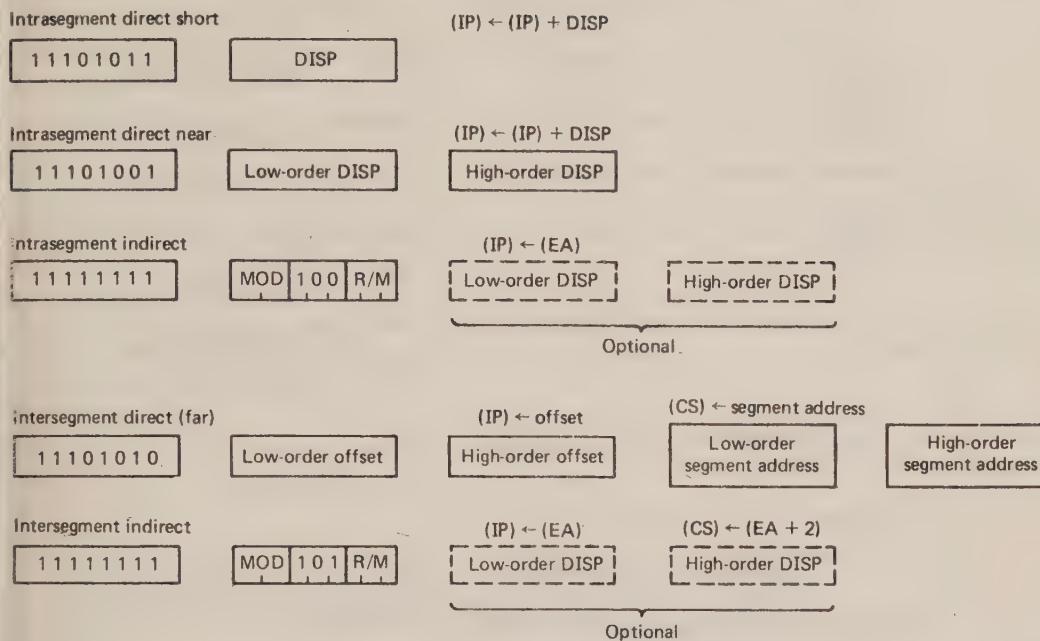


Figure 3-33 Machine code formats for unconditional branch instructions.

branch. All five instructions have the same mnemonic, JMP, and include a single operand.

Intrasegment branches may be made using an 8-bit displacement from the current contents of IP, a 16-bit displacement from (IP), or by obtaining the branch address from the location pointed to by the instruction. The 8-bit displacement address computation is performed in the same manner as with the conditional branch instructions and has the same limitations. In particular, the range of branch addresses is from -128 to 127 bytes relative to the address of the next instruction. The 16-bit displacement address computation is performed similarly, but because the displacement now has 16 bits a branch can be made to anywhere within the current segment. By ignoring carries from the sixteenth bit this computation treats the current segment as a circle. Therefore, although the assembler computes the displacement as a 2's complement number, the range is from $-(IP)$ to $2^{16} - (IP)$, not -2^{15} to $2^{15} - 1$. For example, if as shown in Fig. 3-35, $(IP) = 1000$ and the displacement is EFFF or less, then the displacement would cause a forward branch, but if the displacement were F000 or greater, then a backward branch would be made.

Intersegment branches must replace the contents of the CS register as well as the IP register. They may be direct, in which case the new contents of the IP and CS registers are part of the instruction, or indirect, with the first word of the operand containing the new contents of IP and the next word containing the new contents of CS. For intersegment indirect branches the operand must point to a memory location.

| Name | Mnemonic and Format | Description |
|----------------------------------|---------------------|--|
| Intrasegment direct short branch | JMP SHORT OPR | (IP) \leftarrow (IP) + sign extended D8 determined by OPR |
| Intrasegment direct near branch | JMP NEAR PTR OPR | (IP) \leftarrow (IP) + 16-bit displacement determined by OPR |
| Intrasegment indirect branch | JMP OPR* | (IP) \leftarrow (EA) where EA is determined by OPR |
| Intersegment direct (far) branch | JMP FAR PTR OPR | (IP) \leftarrow Offset of OPR within segment (CS) \leftarrow Segment address of segment containing OPR |
| Intersegment indirect branch | JMP OPR* | (IP) \leftarrow (EA) where EA is determined by OPR (CS) \leftarrow (EA+2) where EA is determined by OPR |

*Type of branch determined by type of operand.

Flags: No flags are affected.

Addressing modes: For intrasegment direct branches the mode is relative and for intersegment direct branches the mode is direct. Indirect branches cannot involve immediate modes and a memory addressing mode must be used in intersegment indirect branches.

Figure 3-34 Unconditional branch instructions.

Because all of the unconditional branch instructions have the same mnemonic, the assembler must have some other means of determining which of the five types of branches is intended. Although there are several ways in which the assembler can determine the type of branch, it is recommended that a convention for indicating the instruction type be established and adhered to so that programs can be more readily understood. The convention assumed in this book is that the intrasegment direct short, intrasegment direct near, and intersegment direct branch instructions will include the attribute operators SHORT, NEAR PTR, and FAR PTR, respectively. If these prefixes are always used for these types of branches, then the lack of an attribute operand would indicate an indirect branch. For an indirect branch, the type of operand always dictates whether the branch is to be intrasegment or intersegment. If it is a word operand, an intrasegment branch is assumed. If it is a double word, which is needed to hold both the offset and segment address, an intersegment branch is assembled. (This means that the type of operand must be known at the time the assembler encounters the branch instruction, a requirement discussed further in Sec. 3-11. Operand types are considered in Sec. 3-10-1.)

To demonstrate both conditional and unconditional branches let us consider the program sequence in Fig. 3-36. This sequence determines the numbers of positive, zero, and negative values in a set of N words beginning at ARRAY. It first counts the number of positive values using register DI and the number of zero values using register SI. It then computes the number of negative values by subtracting (DI) and (SI) from N and leaves the result in AX. If there are any negative values, a near branch is made to NEG_VAL.

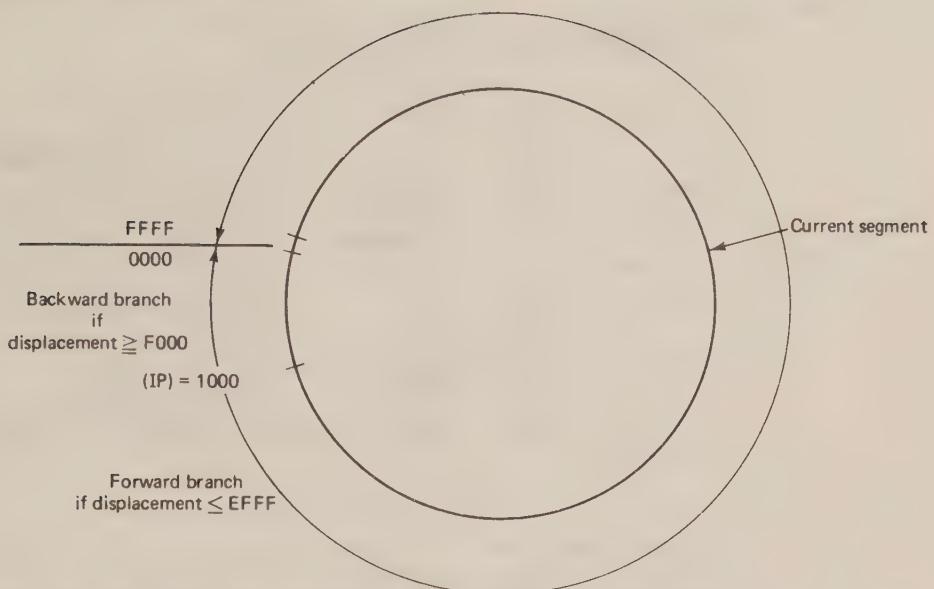


Figure 3-35 Branch address computation using 16-bit displacement.

Because conditional branches have a limited range, a conditional branch instruction must be combined with an unconditional branch instruction to produce a conditional branch to a distant point. This is demonstrated by the last two instructions. The conditional branch

JZ SKIP

causes the unconditional branch

JMP NEAR PTR NEG_VAL

to be skipped if there are no negative values; otherwise, the unconditional branch is made and execution continues at NEG_VAL. If NEG_VAL were in a different code segment, then NEAR should be replaced by FAR.

An intersegment branch is shown in Fig. 3-37. In this example the JMP instruction would cause the offset of CONTINUE within the code segment CODE2 to be put in IP and the segment address of CODE2 to be put in CS.

Indirect branches do not include the branch address as part of the instruction, but must retrieve this address from a memory location or register. An intrasegment indirect branch can use any data-related addressing mode to specify the location of the branch address (except, of course, the immediate mode). An intersegment indirect branch is more limited because both an offset and a segment address are needed; thus the operand must point to a double word in memory and may not point to a register. As an example of an indirect branch, the instruction

JMP TABLE[SI]

would get the branch address from the memory location whose address is that of

Data-related directives, input, and other code
N and ARRAY are defined as word variables

```

        MOV    CX,N      ;LOAD ARRAY SIZE TO CX
        MOV    BX,0      ;CLEAR BX
        MOV    DI,BX    ;USE DI TO COUNT ELEMENTS > 0
        MOV    SI,BX    ;USE SI TO COUNT ELEMENTS = 0
AGAIN:   CMP    ARRAY[BX],0 ;BRANCH TO LESS OR EQ
        JLE    LESS_EQ    ;IF ELEMENT LESS OR EQ 0
        INC    DI        ;OTHERWISE, INCREMENT DI
        JMP    SHORT NEXT ;AND BRANCH TO NEXT
LESS_EQ:  JL     NEXT    ;IF ELEMENT < 0, GO TO NEXT
        INC    SI        ;OTHERWISE, INCREMENT SI
NEXT:/:   ADD    BX,2      ;INCREMENT OFFSET
        DEC    CX        ;DECREMENT LOOP COUNTER
        JNZ    AGAIN    ;AND REPEAT IF NONZERO
        MOV    AX,N      ;SUBTRACT DI AND SI
        SUB    AX,DI    ;FROM N TO GET
        SUB    AX,SI    ;NO. OF ELEMENTS < 0
        JZ     SKIP     ;ARE THERE NEGATIVE VALUES?
        JMP    NEAR PTR NEG_VAL;IF SO, BRANCH TO NEG_VAL
SKIP:    .
        .
        .

```

Output and other code

Figure 3-36 Example involving both conditional and unconditional branch instructions.

TABLE + (SI). This instruction would cause the branch address to be selected from an array of addresses and could be used as part of a multiple-branch decision similar to the FORTRAN computed GO TO.

```

CODE1     SEGMENT
        .
        .
        .
        JMP    FAR PTR CONTINUE
        .
        .

CODE1     ENDS
        .
        .
        .

CODE2     SEGMENT
        .
        .
        .
CONTINUE: MOV    AX,BX
        .
        .

CODE2     ENDS
        .

```

Figure 3-37 Example of an intersegment branch.

3-5 LOOP INSTRUCTIONS

Post-test loops are most often constructed as shown in Fig. 3-38. A counter is set to the number of repetitions and during each repetition it is decremented. When the counter reaches zero the loop is exited. If the CX register is used as the counter

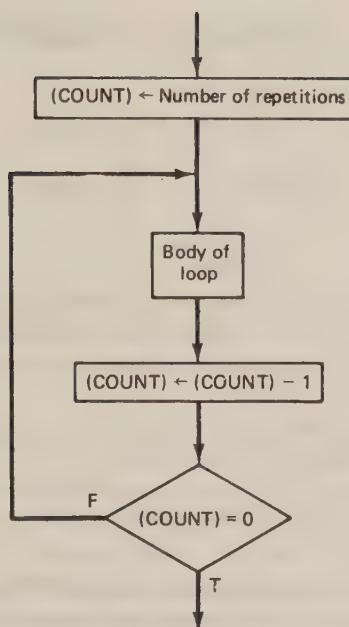


Figure 3-38 Typical structure of a post-test loop.

and N contains the number of repetitions, then a post-test loop could be implemented on the 8086 as follows:

| | |
|-----------|----------------|
| BEGIN: | MOV CX,N |
| . | } Body of loop |
| . | |
| DEC CX | |
| JNZ BEGIN | |

The loop instructions are designed to simplify the decrementing, testing, and branching portion of the loop. In the above case this portion has required two instructions, but in more complicated situations may require more than two instructions.

The loop instructions for the 8086 all have the form



where D8 is a 1-byte displacement from the current contents of IP. Their similarity to the conditional branch instructions is evident and they must obey the same limitations, including the one that restricts the branch address to the range -128 to 127 bytes from the next instruction. The loop instructions are defined in Fig. 3-39. From the definition of the LOOP instruction it is seen that the above post-test loop implementation could be simplified to:

| | |
|------------|----------|
| BEGIN: | MOV CX,N |
| . | . |
| . | |
| LOOP BEGIN | |

| Name | Mnemonic and Format | Alternate Mnemonic | Test Condition* |
|-------------------------------------|---------------------|--------------------|---------------------|
| Loop | LOOP OPR | | (CX) ≠ 0 |
| Loop while zero, or equal | LOOPZ OPR | LOOPE | ZF = 1 and (CX) ≠ 0 |
| Loop while nonzero, or not equal | LOOPNZ OPR | LOOPNE | ZF = 0 and (CX) ≠ 0 |
| Branch on CX | JCXZ OPR | | (CX) = 0 |

*Except for JCXZ which leaves (CX) unchanged, $(CX) \leftarrow (CX)-1$. Then if the test condition is met, $(IP) \leftarrow (IP) + \text{sign extended D8}$; otherwise (IP) are unchanged and the program continues in sequence.

Flags: No flags are affected.

Addressing modes: Mode is relative to IP. OPR must represent a label that is within -128 to 127 bytes of the instruction following the loop instruction.

Figure 3-39 Loop instructions.

Technically, the instructions defined in Fig. 3-39 are branch instructions, but because they are designed for the specific purpose of implementing loops they are referred to as loop instructions. Only the instructions defined in the preceding section are called branch instructions.

A program sequence which uses the LOOP instruction to add the contents of M words beginning at address ARRAY and stores the result in TOTAL is given in Fig. 3-40. As a second example, a program sequence that adds two 16-digit packed BCD numbers is given in Fig. 3-41. Each of the 16-digit numbers is assumed to be in 8 consecutive bytes with the low-order two digits being in the byte with the lowest address. The sum is assumed to be similarly stored in 8 bytes.

The JCXZ instruction is actually just a conditional branch instruction that bases its decision on the CX register instead of on the flags. It has been included with the loop instructions because it is most often used to branch around a loop if the initial count is zero.

The remaining loop instructions, LOOPZ (or LOOPE) and LOOPNZ (or LOOPNE), test two conditions before deciding whether or not the loop is to be

Figure 3-40 Program for adding an array of binary numbers.

Data-related directives, input, and other code
M, ARRAY, and TOTAL are defined as word variables

```

    MOV   CX,M           ;PUT COUNT IN CX
    MOV   AX,0           ;ZERO AX
    MOV   SI,AX          ;AND SI
START_LOOP: ADD  AX,ARRAY[SI] ;ADD NEXT ELEMENT TO AX
    ADD  SI,2            ;INCREMENT INDEX BY TWO
    LOOP  START_LOOP    ;REPEAT LOOP IF CX NONZERO
    MOV   TOTAL,AX        ;STORE RESULT IN TOTAL

```

Output and other code

Data-related directives, input, and other code
 AUGEND, ADDEND, and SUM are defined as byte variables

```

    MOV      CX,8          ;PUT COUNT IN CX
    MOV      SI,0          ;ZERO INDEX
    CMP      AX,AX         ;CLEAR CARRY
    BEGIN:   MOV      AL,AUGEND[SI] ;ADD TWO
              ADC      AL,ADDEND[SI] ;DIGITS AND
              DAA
              MOV      SUM[SI],AL ;RESULT IN SUM
              INC      SI           ;INCREMENT INDEX
              LOOP    BEGIN        ;REPEAT IF CX IS NONZERO
  
```

Output and other code

Figure 3-41 Program for adding two 16-digit packed BCD numbers.

repeated. (The LOOPE and LOOPNE mnemonics are normally used when the loop instructions follow comparisons.) A classical example of testing two conditions before repeating the loop is found in searching an array for a given element. Suppose that ASCII_STR is the variable associated with the beginning of a string containing L characters and the string is to be searched for a space character (whose ASCII code is 20_{16}). The loop is to terminate when a space is found or the entire string has been searched. If a space is not found, a branch is to be made to NOT_FOUND. The code for the needed sequence is given in Fig. 3-42.

Loops may be nested and branches may be made within a loop or nest of loops. To demonstrate a more complicated situation let us consider a program sequence for sorting the numbers in an array into descending order. An algorithm for sorting an array, known as a *bubble sort*, is flowcharted in Fig. 3-43. In the flowchart, N represents the number of elements in the array, A is the name of the array, and I is the index within the array. A bubble sort proceeds by starting at the beginning of an array and puts successive pairs of elements in descending order. After the first pass through the array the smallest element must be at the end; therefore, during the second pass, only the first $N - 1$ elements need to be considered. Similarly, during the third pass only $N - 2$ elements must be considered, and so on. A program sequence for executing a bubble sort is given in Fig. 3-44. A more efficient method of sorting, called an *insertion sort*, is outlined in Exercise 34.

Figure 3-42 Search example using LOOPNE.

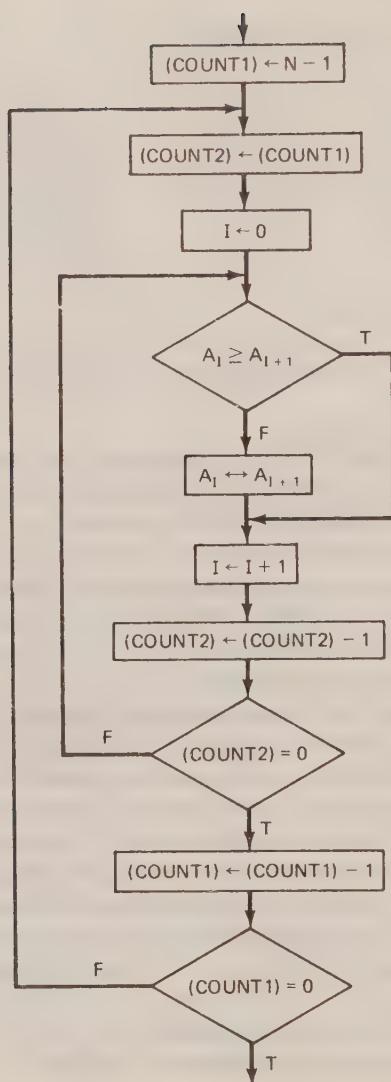
Data-related directives, input, and other code
 L is defined as word variable and ASCII_STR is defined as byte variable

```

    MOV      CX,L          ;PUT ARRAY SIZE IN CX,
    MOV      SI,-1         ;INITIALIZE INDEX, AND
    MOV      AL,20H         ;PUT CODE FOR SPACE IN AL
    NEXT:   INC      SI           ;INCREMENT INDEX
              CMP      AL,ASCII_STR[SI] ;TEST FOR SPACE
              LOOPNE  NEXT        ;LOOP IF NOT SPACE AND COUNT IS NONZERO
              JNZ      NOT_FOUND    ;IF SPACE NOT FOUND, BRANCH TO NOT_FOUND
  
```

Output and other code

Figure 3-43 Flowchart of a bubble sort.



3-6 NOP AND HLT INSTRUCTIONS

An instruction that is quite often used in conjunction with a branch instruction is the no operation, or *no op*, instruction. The no op instruction for the 8086 has the mnemonic NOP and is defined in Fig. 3-45. A no op instruction serves the same purpose as the CONTINUE instruction in FORTRAN and other high-level languages, as a means of providing a label for a “branch to” point that does nothing. The availability of such an instruction is very important, particularly during the

Data-related directives, input, and other code
N and A are defined as word variables

```

        MOV    CX,N      ;SET COUNT1
        DEC    CX       ;TO N-1
LOOP1:   MOV    DI,CX   ;SAVE COUNT1 IN DI
        MOV    BX,0     ;CLEAR BX
        MOV    AX,A[BX] ;LOAD A(I) INTO AX AND
        CMP    AX,A[BX+2];COMPARE WITH A(I+1)
        JGE    CONTINUE;SWAP IF
        XCHG   AX,A[BX+2];A(I) < A(I+1) AND
        MOV    A[BX],AX ;STORE GREATER NUMBER
CONTINUE: ADD    BX,2     ;INCREMENT INDEX
        LOOP   LOOP2    ;IF NOT THE END OF A PASS, REPEAT
        MOV    CX,DI   ;RESTORE COUNT1 FOR THE OUTER LOOP
        LOOP   LOOP1    ;IF NOT THE FINAL PASS, REPEAT

```

Output and other code

Figure 3-44 Program sequence for performing a bubble sort.

development of a program, when modifications are frequently necessary. To attach the "branch to" label to an instruction that takes action as follows

```

        JZ           EXIT
        .
        .
        .
        EXIT:      MOV    AX,SUM[BX]

```

is relatively inflexible because, in order to insert new instructions at the point labeled EXIT, the move instruction must be retyped. If the sequence

```

        JZ           EXIT
        .
        .
        .
        EXIT:      NOP
        MOV    AX,SUM[BX]

```

were used, insertions could be made without disturbing the present code. This is important during the debugging phase when message printout code may need to be temporarily included at key points (which are often "branch to" points) within

Figure 3-45 NOP and HLT instructions.

| Name | Mnemonic and Format | Description |
|--------------|---------------------|--|
| No operation | NOP | Cause no action |
| Halt | HLT | Cause the operation of the computer to cease |

Flags: No flags are affected.

Addressing modes: None.

a program. Finally, just as FORTRAN DO-loops are often terminated with CONTINUE statements, it may be desirable to put NOP instructions at the end of pre-test loops, especially when loops are nested.

In addition, Fig. 3-45 includes the halt, HLT, instruction. It is also sometimes inserted at key points during the development of a program. A HLT instruction causes the computer to cease its operation. By examining the state of the computer at the time it is halted one can get information regarding the recent activity within the computer. HLT instructions are frequently found in diagnostic programs that are designed to help maintain a computer system.

3-7 FLAG MANIPULATION INSTRUCTIONS

As we have seen, many instructions set or clear the flags depending on their results. Sometimes, however, it is necessary to have direct control of the flags. The ability to set, clear, or change the CF flag is desirable and we will see later that direct manipulation of the control flags DF and IF is required. In the example in Fig. 3-41 the carry flag needed to be cleared before entering the loop because the ADC instruction in the loop had to perform all additions, including the addition of the low-order bytes. Although, as in this case, it is possible to clear CF with the instruction

CMP AX,AX

it would be more readable and efficient to have an instruction that explicitly cleared CF. The flag manipulation instructions for the 8086 are summarized in Fig. 3-46.

Figure 3-46 Flag manipulation instructions.

| Name | Mnemonic and Format | Description |
|---------------------|---------------------|---|
| Clear carry | CLC | CF \leftarrow 0 |
| Complement carry | CMC | CF $\leftarrow \neg CF$ |
| Set carry | STC | CF \leftarrow 1 |
| Clear direction | CLD | DF \leftarrow 0 |
| Set direction | STD | DF \leftarrow 1 |
| Clear interrupt | CLI | IF \leftarrow 0 |
| Set interrupt | STI | IF \leftarrow 1 |
| Load AH from flags* | LAHF | (AH) \leftarrow (Low-order byte of PSW) |
| Store AH to flags* | SAHF | (Low-order byte of PSW) \leftarrow (AH) |

* Bits 0, 2, 4, 6, and 7 are transferred according to Fig. 2-8.
Bits 1, 3, and 5 are indeterminate.

Flags: Only the indicated flags are affected.

Addressing modes: None

The STC, CLC, and CMC instructions, which respectively set, clear, and complement the carry flag CF, are typically used in multiple-precision arithmetic. STD and CLD set and clear the direction flag DF, which affects the operation of the string manipulation instructions considered in Chap. 5, and STI and CLI set and clear the interrupt flag IF, which controls the maskable interrupts discussed in Chaps. 4 and 6. The other two instructions defined in Fig. 3-46 are designed to allow the efficient translation into 8086 programs of programs written for the 8080. The LAHF and SAHF instructions transfer the low-order byte of the PSW to and from the AH register. Except for the overflow flag (the flag with no 8080 counterpart), this byte includes all of the condition flags. LAHF and SAHF can be used with the logical instructions to initialize or compare the SF, ZF, AF, PF, and CF flags with a given bit pattern.

3-8 LOGICAL INSTRUCTIONS

The 8086 instructions for performing logical operations are defined in Fig. 3-47. All of the instructions operate bitwise on their operands, which may be one byte or one word in length. That is, for the NOT instruction each bit is complemented, and for the other instructions the logical operation is performed on each of the pairs of corresponding bits. For example if

$$(DL) = 10001010$$

and

NOT DL

is executed, then

$$\text{New (DL)} = 01110101$$

Figure 3-47 Logical instructions.

| Name | Mnemonic and Format | | Description |
|--------------|---------------------|------------|---|
| Not | NOT | OPR | $(OPR) \leftarrow (\overline{OPR})$ |
| Or | OR | DST, SRC | $(DST) \leftarrow (DST) \vee (SRC)$ |
| And | AND | DST, SRC | $(DST) \leftarrow (DST) \wedge (SRC)$ |
| Exclusive Or | XOR | DST, SRC | $(DST) \leftarrow (DST) \oplus (SRC)$ |
| Test | TEST | OPR1, OPR2 | $OPR1 \wedge OPR2$; result not put anywhere, only the flags are affected |

Flags: NOT does not affect the flags. The other four instructions clear CF and OF, leave AF undetermined, and set SF, ZF, and PF according to the usual rules.

Addressing modes: The NOT operand cannot be immediate. For the remaining instructions, unless the source operand is immediate, at least one of the operands must be a register. The other operand may have any addressing mode.

If

$$\begin{aligned} (\text{AL}) &= 10010110 \\ (\text{LOG_DATA}) &= 01011010 \end{aligned}$$

then the instruction

OR AL,LOG_DATA

would produce

$$\text{New (AL)} = 11011110,$$

the instruction

AND AL,LOG_DATA

would produce

$$\text{New (AL)} = 00010010,$$

and

XOR AL,LOG_DATA

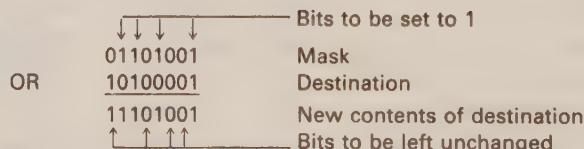
would result in

$$\text{New (AL)} = 11001100$$

The TEST instruction is the same as the AND instruction except that it does not put the result anywhere. Like the CMP instruction, it is used only to set the flags.

The NOT instruction does not affect the flags. All of the other logical instructions clear CF and OF, leave AF undetermined, and set SF, ZF, and PF according to the usual rules. For the double operand instructions, if the source is immediate and the destination is AL or AX, the assembler will assemble a short-form machine instruction (2 or 3 bytes long).

The logical instructions may be employed in a variety of ways, but they are most frequently used to selectively set, change, clear, or test bits in the destination operand according to the bit pattern in the source operand. These actions are particularly important in manipulating bits in I/O registers and I/O data—see Chap. 6. In such applications the source operand is called a *mask* and the operation is called a *masking operation*. Bits are selectively set by applying a logical OR as follows:



Because bits 0, 3, 5, and 6 of the mask are set to 1, these bits in the destination will also be set to 1, but because bits 1, 2, 4, and 7 in the mask are 0 the corresponding bits in the destination will be left unchanged. This masking operation selectively sets bits 0, 3, 5, and 6 and leaves the remaining bits alone. It is left to

the reader to show that the XOR instruction can be similarly used to selectively complement bits.

To selectively clear bits, the bits in the mask that correspond to the bits to be cleared are set to 0, the other bits are set to 1, and the AND instruction is applied. If bits 2, 5, and 7 are to be cleared while the remaining bits are to be left unchanged, the following operation is performed:

| | | |
|-----|-----------------|-----------------------------|
| | ↓ ↓ ↓ | Bits to be cleared |
| | 01011011 | Mask |
| AND | <u>10010101</u> | Destination |
| | 00010001 | New contents of destination |
| | ↑ ↑↑ ↑↑ | Bits to be left unchanged |

For selective testing, if any of the 1-bits in the mask are matched by 1-bits in the destination, then the TEST instruction will result in ZF equal to 0; otherwise, ZF will be 1. Presumably, the TEST instruction would be followed by a JNZ or JZ instruction so that a decision could be made based on the test. A test for all bits in the destination corresponding to 1-bits in the mask being set to 1 could be made as follows:

| | | |
|------------|-----------------|---------------------------|
| | 01101101 | Destination |
| | 10010010 | Complement of destination |
| TEST (AND) | <u>01100100</u> | Mask |
| | 00000000 | Result |
| | ↑↑ | Bits tested |

If any of the tested bits in the original destination had been 0, the result would have been nonzero. If the destination is AL, this test could be implemented with the sequence

```
NOT    AL
TEST   AL,MASK_PAT
JZ     ALL_BITS_SET
```

The XOR instruction can be used to determine whether the bit patterns in its operands exactly match. The instructions

```
XOR   AX,TEST_PAT
JZ    MATCH
```

would result in a branch to MATCH if the bit pattern in TEST_PAT exactly matches the pattern in AX.

Figure 3-48 gives a program sequence that selectively sets bits 0 and 2 of DL, changes bits 1 and 6, clears bits 3 and 4, and then tests bits 1 and 7 and branches to EXIT if they are both set. Figure 3-49 provides a sequence that branches to TASK1, TASK2, or TASK3 according to the bits in AX and the following rules:

1. If bits 1 and 14 or bits 7 and 9 are set, branch to TASK1.

Data-related directives, input, and other code

```

OR      DL,00000101B ;SET BITS 2 AND 0
XOR     DL,01000010B ;COMPLEMENT BITS 6 AND 1
AND     DL,11100111B ;CLEAR BITS 3 AND 4
MOV     AL,DL          ;DUPLICATE RESULT IN AL
NOT     AL              ;AND BRANCH
TEST    AL,10000010B   ;TO EXIT IF BOTH
JZ      EXIT            ;BITS 7 AND 1 ARE SET

```

Output and other code

Figure 3-48 Example of selectively setting, changing, clearing, and testing bits.

2. If no branch is made to TASK1 and either bit 3 or bit 4 is set, branch to TASK2.
3. Otherwise, execute TASK3.

The logical instructions may be used to evaluate logical expressions. The program sequence in Fig. 3-50, which assumes that bits 7, 6, . . . , 1, 0 in AL respectively represent the values of the logic variables X_7, \dots, X_0 , evaluates the Boolean expression

$$f(X_7, \dots, X_0) = X_7\bar{X}_6X_3X_1 + X_6X_4\bar{X}_2X_1X_0 + \bar{X}_7X_5X_3\bar{X}_1$$

and puts the value of f in AH. The first six instructions create the masks for the three terms by ORing AL with bytes that have 1s in the “don’t care” positions and 0s elsewhere. Then the term patterns are compared to the masks and if there is a match AH is set to 1. Otherwise, AH is cleared to 0 by performing an exclusive OR of AH with itself. Using the XOR instruction to clear a register is more efficient than using a MOV immediate instruction because it requires only 2 bytes and is faster, but it also makes the program a little less readable.

Figure 3-49 Using bit settings for program control.

Data-related directives, input and other code

```

NOT     AX          ;BRANCH TO
TEST    AX,4002H   ;TASK1 IF BITS 14 AND 1
JZ      TASK1      ;ARE SET, OR IF
TEST    AX,280H    ;BITS 9 AND 7
JZ      TASK1      ;ARE SET
NOT     AX          ;BRANCH TO TASK2
TEST    AX,18H     ;IF BIT 3 OR BIT 4
JNZ    TASK2      ;IS SET
TASK3:   :          ;OTHERWISE, TASK3
          :          ;IS EXECUTED
          :} Task 3
TASK1:   :          ;
          :} Task 1
TASK2:   :          ;
          :} Task 2
          :

```

Output and other code

Data-related directives, input and other code

```

MOV    BL, 00110101B ;CREATE MASKS FOR
OR     BL, AL   ;EACH OF THE
MOV    BH, 10101000B ;THREE MINTERMS
OR     BH, AL   ;BY SETTING THE DON'T
MOV    CL, 01010101B ;CARE BITS
OR     CL, AL   ;TO 1
XOR    BL, 10111111B ;BRANCH TO
JZ    TRUE   ;TRUE IF ANY
XOR    BH, 11111011B ;ONE OF THE
JZ    TRUE   ;MINTERMS IS 1
XOR    CL, 01111101B
JZ    TRUE
XOR    AH, AH   ;OTHERWISE, AH IS CLEARED
JMP    CONTINUE
TRUE: MOV    AH, 1    ;AH IS SET TO 1
CONTINUE:
        .
        .

```

Output and other code

Figure 3-50 Sequence for evaluating a Boolean expression.

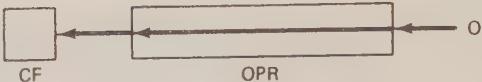
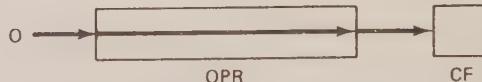
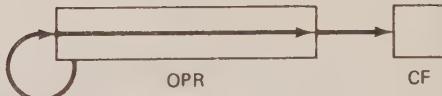
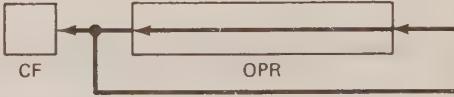
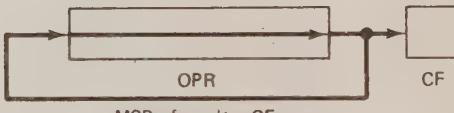
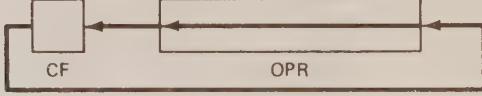
3-9 SHIFT AND ROTATE INSTRUCTIONS

The shift and rotate instructions for the 8086 are defined in Fig. 3-51. These instructions shift all of the bits in the operand to the left or right by the specified count, CNT. For the shift left instructions, 0s are shifted into the right end of the operand and the MSBs are shifted out the left end and lost, except that the least significant of these MSBs is retained in the CF flag. The shift right instructions similarly shift bits to the right; however, SAR (shift arithmetic right) does not automatically insert 0s from the left; it extends the sign of the operand by repeatedly inserting the MSB. The rotate instructions differ from the shift instructions in that the operand is treated like a circle in which the bits shifted out of one end are shifted into the other end. The RCL and RCR instructions include the carry flag in the circle and the ROL and ROR do not, although the carry flag is affected in all cases.

The shift instructions affect all of the condition flags and the rotate instructions affect only the CF and OF flags. However, the AF flag is left with an indeterminate value by the shift instructions. For the shift and rotate instructions, OF is left with a meaningful value only if the count is 1, in which case it is set according to the rule: If the two MSBs of the operand are equal, then OF=0; otherwise, OF=1. For the most part, the contents of CF are most useful even when the other flags are set according to specific rules.

The destination operand, OPR, can have any of the 8086 addressing modes except the immediate mode. CNT can be a 1, a constant expression that evaluates to a 1, or the register designation CL. If it is CL, then the number of positions to be shifted is determined by the contents of CL. The instructions

SHR QUAN[SI],1

| Name | Mnemonic and Format | Description* |
|----------------------------|---------------------|--|
| Shift logical left | SHL OPR,CNT |  |
| Shift arithmetic left | SAL OPR,CNT | Same as SHL |
| Shift logical right | SHR OPR,CNT |  |
| Shift arithmetic right | SAR OPR,CNT |  |
| Rotate left | ROL OPR,CNT |  |
| Rotate right | ROR OPR,CNT |  |
| Rotate left through carry | RCL OPR,CNT |  |
| Rotate right through carry | RCR OPR,CNT |  |

*Number of bit positions shifted is determined by CNT

Flags: CF flag set as indicated. PF, SF, and ZF flags are set by shift instructions but left unchanged by the rotate instructions. OF flag is meaningful only if count is 1. AF flag is affected by shift instruction, but has no meaning.

Addressing modes: OPR can have any mode except immediate; CNT must be 1 or CL.

Figure 3-51 Shift and rotate instructions.

and

SHR QUAN[SI],ONE

are the same if ONE is a name identifier that has been associated with the number 1 (see Sec. 3-10-4). The instructions

MOV CL,6
SAL DATA[BX][DI],CL

would cause the contents of the location pointed to by the EA to be shifted to the left by 6 bit positions. The instruction

SHR AL,3

is invalid because a constant for CNT can only be equal to 1, and

SHR AL,BL

is invalid because only CL can be used as the count register. Figure 3-52 assumes that initially (DL) = 8D, (CL) = 3, and CF = 1, and indicates the results of several shift and rotate instructions.

The machine code format of the shift and rotate instructions is of the form



The w-bit serves the usual purpose of identifying whether a byte or word is to be operated on by the instruction. The v-bit is set to 0 if the shift count is to be 1 and is set to 1 if the CL register contains the shift count. The three center bits in the second byte identify one of the seven possible shift or rotate instructions.

The shift and rotate instructions are used to reformat data, provide program control in certain situations, and serve several other special activities. It should be

| Instruction | | Results | |
|-------------|-------|---------|----------------------|
| SHL | DL,CL | CF = 0 | 01101000 |
| SAL | DL,CL | CF = 0 | 01101000 |
| SHR | DL,CL | | 00010001 CF = 1 |
| SAR | DL,CL | | 11110001 CF = 1 |
| ROL | DL,CL | CF = 0 | 01101100 |
| ROR | DL,CL | | 10110001 CF = 1 |
| RCL | DL,CL | CF = 0 | 01101110 |
| RCR | DL,CL | | 01110001 CF = |

Figure 3-52 Examples of shift and rotate instructions.

Initially:
 (DL) = 10001101
 (CL) = 00000011
 (CF) = 1

Data-related directives, input, and other code
UNPACKED and PACKED are defined as byte variables

```

        MOV    DX,8           ;SET LOOP COUNTER DX TO 8
        MOV    CL,4           ;SET SHIFT COUNT TO 4
        MOV    SI,0           ;ZERO INDICES
        MOV    DI,SI          ;IN SI AND DI
CONVERT:   MOV    AX,WORD PTR UNPACKED[SI] ;BRING 2 DIGITS INTO AX
        SHL    AL,CL          ;AND PACK THEM
        SHR    AX,CL          ;
        MOV    PACKED[DI],AL  ;STORE RESULT
        ADD    SI,2           ;INCREMENT INDICES
        INC    DI              ;IN SI AND DI
        DEC    DX              ;DECREMENT LOOP COUNT
        JNZ    CONVERT         ;REPEAT IF COUNT NONZERO

```

Output and other code

Figure 3-53 Conversion of a 16-digit unpacked BCD number to a packed BCD number.

noted that a shift left by n positions is equivalent to a multiplication by 2^n , e.g.,

$$6 \times 2^2 = 00000110 \times 100 = 00011000$$

A similar shift right is the same as a division by 2^n . If the shift right is logical, then the division is unsigned, and if it is arithmetic, the division is signed.

Figure 3-53 gives the code needed to convert the 16-digit unpacked BCD number beginning at UNPACKED into a packed BCD number and to store it a PACKED. The shift instructions are used to delete the extra bits. The reader is asked to write the code for the inverse conversion in Exercise 35.

The program control example flowcharted in Fig. 3-54(a) is used to demonstrate the rotate instructions. The corresponding code is given in Fig. 3-54(b). The sequence contains a loop that is exited only when (DH), which are initially zeroed, become nonzero. Inside the loop the bits in DL are rotated left by one position with each execution of the loop. One of two program sequences is executed depending on the DL bit that is rotated into the CF flag. Presumably one of these sequences sets DH to a nonzero value under certain conditions. In order to ensure enough space for the tasks, near unconditional branches have been used in some places instead of the conditional branches.

3-10 DIRECTIVES AND OPERATORS

Assembler instructions are translated into machine language instructions and correspond to executable statements in high-level language programs. Just as high-level language programs must have nonexecutable statements to preassign values, reserve storage, assign names to constants, form data structures, and terminate a compilation, assembler language programs must contain directives to perform similar tasks. Because of the 8086's reliance on segment registers, an 8086 assembler must also include directives for indicating to the assembler the assumed contents in the segment registers under the various circumstances as the assembly progresses.

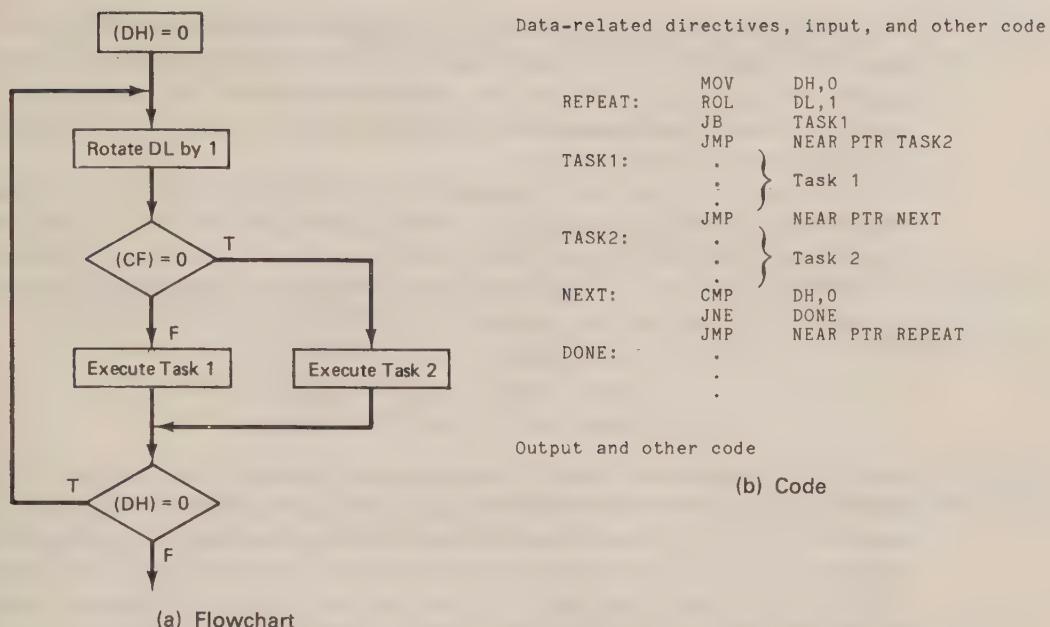


Figure 3-54 Program control example that employs a rotate instruction.

Moreover, since assembler language programming is closer to the actual operations of the computer, most assemblers have directives that give the programmer more control over the exact placement of data and segmentation of the program than is available to the high-level language programmer. This section will introduce the directives that are needed for single-module programs, but those required for joining modules together are left for Chap. 4.

3-10-1 Data Definition and Storage Allocation

Statements that preassign data and reserve storage have the form:

Variable Mnemonic Operand, . . . , Operand ;Comments

where the variable is optional, but if it is present it is assigned the offset of the first byte that is reserved by the directive. Note that unlike the label field, a variable must be terminated by a blank, not a colon. The mnemonic determines the length of each operand and is one of the following:

DB (Define Byte)—Each operand datum occupies one byte.

DW (Define Word)—Each operand datum occupies one word, with its low-order part being in the first byte and its high-order part being in the second byte.

DD (Define Double Word)—Each operand datum is two words long with the low-order word followed by the high-order word.

The operands indicate the data to be preassigned or the amount of space to be reserved. Comments may be used to describe the data assignment or allocation or may be omitted. There must be one or more spaces after the variable and the mnemonic; otherwise, the insertion of spaces is arbitrary (except they cannot appear within mnemonics, identifiers, etc.).

The DB, DW, and DD directives may be used to put data in particular locations or to simply allocate space without preassigning anything to the space. The DW and DD directives may also be used to generate offsets or full addresses of labels and variables. To preassign *data* the operand must be a constant, an expression that evaluates to a constant, or a string constant. For example,

| | | |
|-----------|----|-------------|
| DATA_BYTE | DB | 10,4,10H |
| DATA_WORD | DW | 100,100H,-5 |
| DATA_DW | DD | 3*20,0FFFDH |

will cause a sequence of bytes to be preassigned as shown in Fig. 3-55. The first statement preassigns three constants to three consecutive bytes and associates the variable DATA_BYTE with the first byte. The second statement preassigns three constants to three consecutive words (6 bytes) and associates the variable DATA_WORD with the first byte of the first word. The last statement reserves two double words and preassigns to them the values of the expressions $3*20$ and

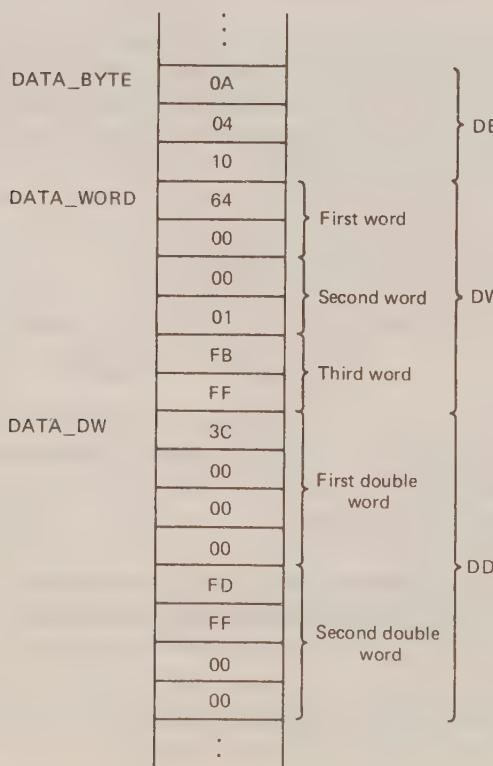


Figure 3-55 Typical preassignment of data using the DB, DW, and DD directives.

FFFFD. It also associates DATA_DW with the first byte of the first double word As another example, an 8-digit number 12345678 can be defined in packed BCD form by

```
PACKED    DB    78H,56H,34H,12H
```

and in unpacked BCD form by

```
UNPACKED   DB    8H,7H,6H,5H,4H,3H,2H,1H
```

The least significant bytes are in the lower addresses represented by PACKED and UNPACKED, respectively.

An ASCII character string can be preassigned by using a string constant as an operand. The statement

```
MESSAGE    DB    'H','E','L','L','O'
```

puts the ASCII codes for H(48), E(45), L(4C), L(4C), and O(4F) in consecutive bytes beginning with the byte whose address is associated with the variable MESSAGE. This statement is equivalent to

```
MESSAGE    DB    'HELLO'
```

Note that the first character in the string goes in the first byte, the second in the second byte, and so on. The DW and DD directives may also preassign character strings, but they are infrequently used for this purpose because the bytes are reversed, i.e.,

```
DB    'AB'
```

would put 'A' in the first byte and 'B' in the second while

```
DW    'AB'
```

would put 'B' in the first and 'A' in the second. Furthermore, string operands in a DW or DD statement cannot exceed two characters in length.

If an operand is a question mark, then no preassignment is made but the appropriate amount of space is reserved (1 byte if it is in a DB statement, 2 bytes if in a DW statement, and 4 bytes if in a DD statement). The statements

```
ABC    DB    0,?,?,?,0  
DEF    DW    ?,52,?
```

assemble the bytes shown in Fig. 3-56.

Preassignment by individually listing the contents of the locations is satisfactory if there are only a few locations to be filled, but if there are several operands, such an approach would be burdensome. Therefore, just as high-level languages allow for duplication factors in their preassignment statements, ASM-86 permits the use of the duplication operator DUP. Several operands or operand patterns can be replaced with a form such as

Exp DUP(Operand, . . . , Operand)

where Exp is an expression that evaluates to a positive integer, which causes the

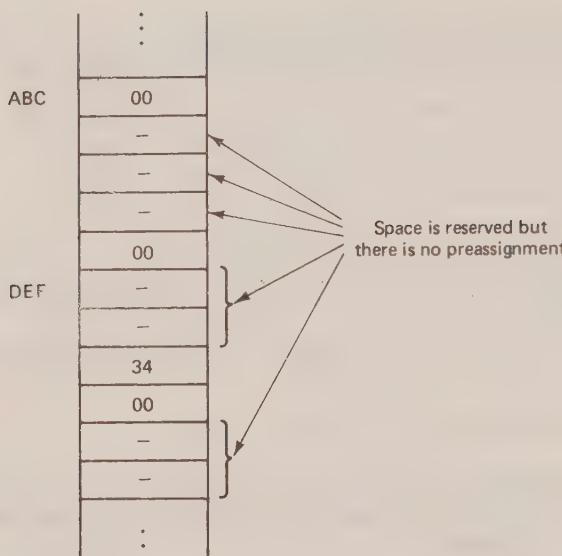


Figure 3-56 Application of the “?” operand.

operand pattern to be repeated the number of times indicated by Exp. The statements

```
ARRAY1    DB    2 DUP(0,1,2,?)
ARRAY2    DB    100 DUP(?)
```

would cause the preassignment and allocation shown in Fig. 3-57(a). The first statement would reserve 8 bytes for ARRAY1 and the second statement would reserve 100 bytes for ARRAY2. The first statement is equivalent to

```
ARRAY1    DB    0,1,2,?,0,1,2,?
```

The DUP operator can also be nested, as in the statement

```
ARRAY3    DB    100 DUP(0,2 DUP(1,2),0,3)
```

which causes data to be preassigned as indicated in Fig. 3-57(b).

It is also possible to preassign the offset or full address of a variable or label by letting an operand in a DW or DD statement be an expression that evaluates to an offset or address, e.g.,

Variable \pm Constant expression

or

Label \pm Constant expression

If the operand appears in a DW statement, then only the offset is stored, but if it appears in a DD statement, then both the offset and the segment address are stored, with the offset being put in the first word. A DB statement cannot be used

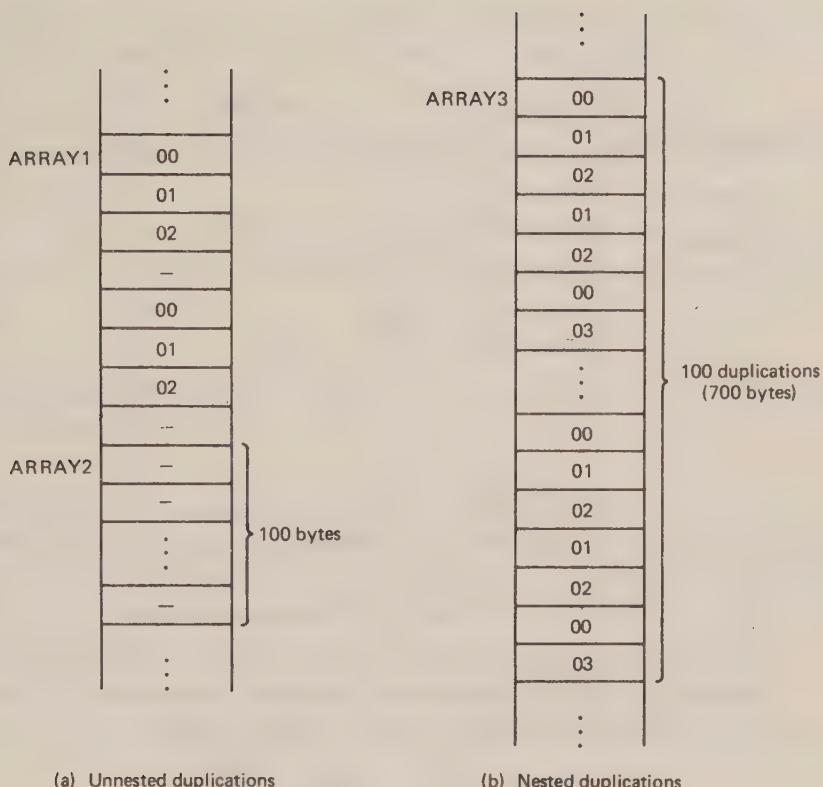


Figure 3-57 Application of the DUP operator.

for storing an address because it reserves only 8 bits. The statements

```
PARAMETER_TABLE DW PAR1
                  DW PAR2
                  DW PAR3
```

would cause the offsets of PAR1, PAR2, and PAR3 to be stored as shown in Fig. 3-58(a). PAR1, PAR2, and PAR3 may be variables or labels. Statements such as

```
INTERSEG_DATA DD DATA1
                  DD DATA2
```

could be used to store both the offsets and segment addresses, as shown in Fig. 3-58(b).

The variable associated with a storage definition statement represents the offset in the current segment of the first byte of the first data item and has a type attribute that indicates the length of each data item in the statement. This length is 1 byte for a DB statement, 2 bytes for a DW statement, and 4 bytes for a DD statement. An expression

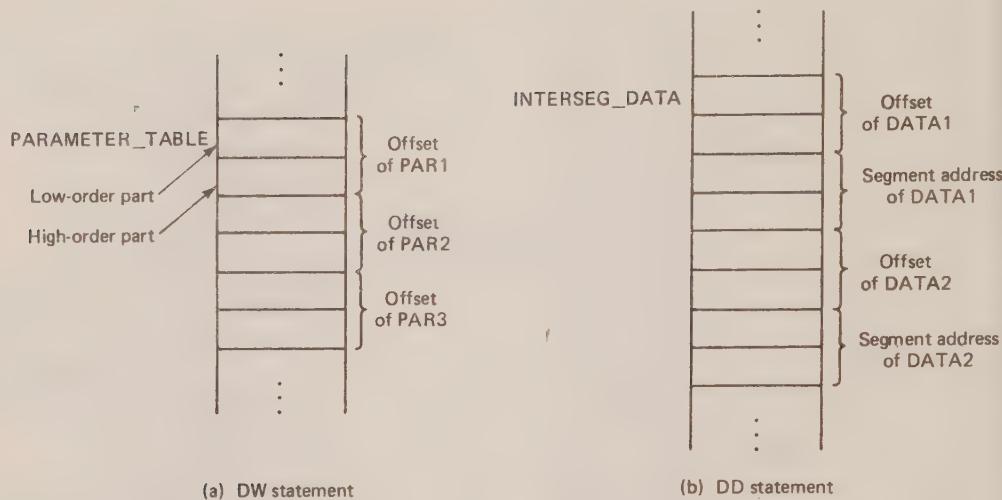


Figure 3-58 Use of DW and DD statements to preassign addresses.

Variable ± Constant expression

is given the same type attribute as its variable. The assembler uses the type attribute to determine whether the machine instruction is to operate on a byte or word (i.e., the w-bit is to be set to 0 or 1). For example, given

```

        MOV    OPER1,0
        MOV    OPER2,0
        .
        .
OPER1    DB    ?,?
OPER2    DW    ?,?

```

the w-bit is set to 0 in the first MOV instruction and to 1 in the second.

Because all variables are typed according to the statement in which they appear, the assembler is able to reduce coding errors by verifying that both operands in a double-operand instruction have the same type attribute. If the first instruction in the sequence

| | | |
|-------|----|-----------|
| OPER1 | DB | 1,2 |
| OPER2 | DW | 1010,2020 |

were allowed, then it would replace the contents of the first byte of OPER2 with (AH). Since this is almost certainly not what is intended, the assembler would not assemble this instruction, but print out an error message. Similarly, the second statement has unlike operands and would generate an error.

There is, however, a way of overriding the implicit type given to an operand, and that is to use a PTR attribute operator in a form such as

Type PTR Variable ± Constant expression

where the type may be BYTE, WORD, or DWORD (for double word). In the above sequence

`MOV WORD PTR OPER1+1,AX`

would allow a word MOV instruction to be assembled without generating an error. Also,

`MOV BYTE PTR OPER2,AL`

would replace the first byte of the first word with the contents of AL. On the other hand, the instruction

`MOV BYTE PTR OPER2+1,AL`

would cause the high-order byte of the first word of OPER2 to be replaced.

Since it may become tedious to use the PTR operator to frequently reference variables that are accessed as two different types, a method has been provided for associating a location with two different variables. The LABEL directive, which has the form

Variable LABEL Type

causes the variable to be typed and assigned to the current offset. The directives

| | | |
|------------|-------|-----------|
| BYTE_ARRAY | LABEL | BYTE |
| WORD_ARRAY | DW | 50 DUP(?) |

would assign both BYTE_ARRAY and WORD_ARRAY to the same location, the first byte of a 100-byte block. The instruction

`MOV WORD_ARRAY+2,0`

would set the third and fourth bytes of the block to 0 and

`MOV BYTE_ARRAY+2,0`

would set the third byte to 0.

3-10-2 Structures

All elements allocated by a single storage definition statement must be of the same type (bytes, words, or doublewords). It is desirable, especially in business data processing applications, for a variable to have several fields, with each field having its own type. For instance, the data structure for a personnel record might be as shown in Fig. 3-59. To permit a program to define and refer to such data structures as a whole, many high-level languages such as Pascal and PL/1 include special nonexecutable statements. Analogously, the ASM-86 assembler includes the STRUC

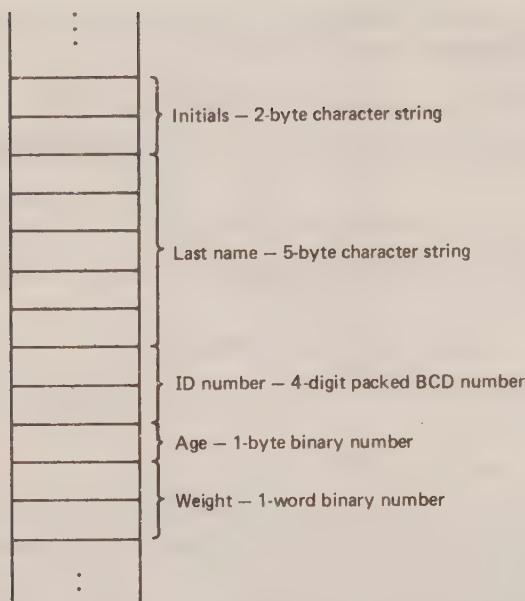


Figure 3-59 Fields in a typical personnel record data structure.

directive to serve these needs and allows instruction operands to be modified so that they can point to a specific field within a structure.

A structure definition gives the pattern of the structure and may have the simplified form

```

Structure name      STRUC
:
    Sequence of DB, DW, and DD directives
:
Structure name      ENDS

```

If a DB, DW, or DD statement includes a variable identifier, it denotes the beginning of a field and is referred to as a *field identifier*. The structure for the personnel record shown in Fig. 3-60 could be defined

```

PERSONNEL_DATA  STRUC
    INITIALS     DB      'XX'
    LAST_NAME    DB      5 DUP(?)
    ID           DB      0,0
    AGE          DB      ?
    WEIGHT       DW      ?
PERSONNEL_DATA  ENDS

```

The structure definition does not reserve storage or directly preassign values; it merely defines a pattern. Therefore, to reserve the necessary space it must be accompanied by a statement for invoking the structure. Invocation statements have the form

Variable Structure name <Preassignment specifications>

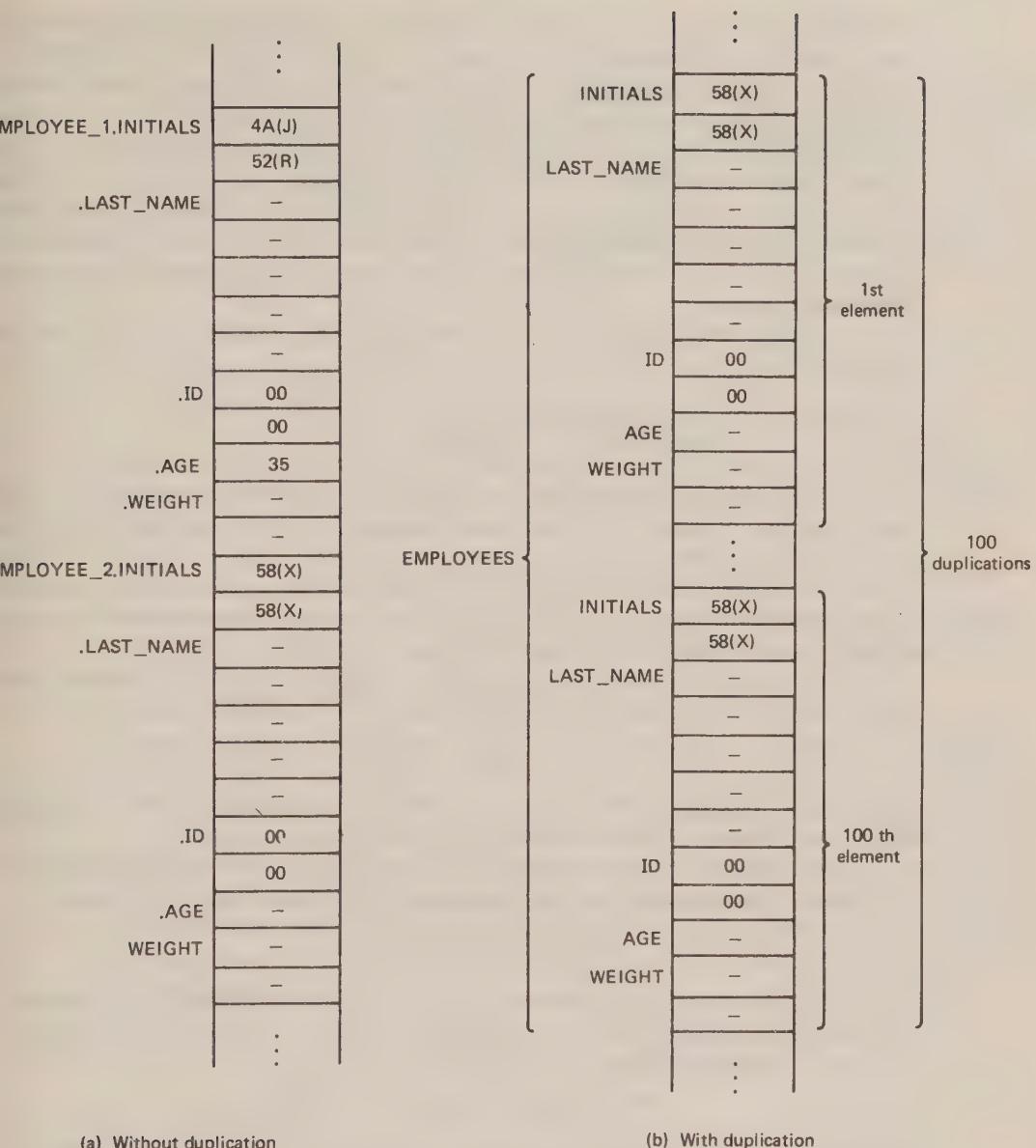


Figure 3-60 Allocation and preassignment of structures.

where the name is the structure name in the STRUC directive and the variable is associated with the beginning of the structure. The variable is also used with the field names to refer to the various fields within the structure.

Although the structure definition may indicate preassigned values, they are only default values and, except when multiple preassignments are made in a single statement, can be overridden by the preassignment specifications. In the above

example, all but the LAST_NAME and ID fields can be overridden. A preassignment specification consists of a sequence of constant expressions, one expression per field, separated by commas. If a sequence begins with one or more commas or if there are consecutive commas, then the absence of one or more expressions is indicated. This means that for the corresponding fields the preassigned values in the structure definition are to be used. There may be more fields than are implied by the expressions and commas, in which case only the structure definition preassignments for the indicated fields are overridden. If there are no expressions or commas (i.e., only `()` appears), then all of the preassignment values in the structure definition are used.

In conjunction with the structure PERSONNEL_DATA one might use the statements

```
EMPLOYEE_1    PERSONNEL_DATA    ('JR', , , 35)
EMPLOYEE_2    PERSONNEL_DATA    ( )
```

which would reserve memory and initialize values as shown in Fig. 3-60(a). The pattern in the structure definition would be used twice, once for allocating space for the variable EMPLOYEE_1 and once for allocating space for EMPLOYEE_2. In the first case the character string 'JR' overrides the string 'XX' and 35 overrides the undetermined value in AGE, but for EMPLOYEE_2 the preassignments are all made according to the structure definition.

The preassignment specifications can be modified by the DUP operator. Once again considering the structure PERSONNEL_DATA, the statement

```
EMPLOYEES    PERSONNEL_DATA    100 DUP( )
```

would cause the allocation shown in Fig. 3-60(b). In this example the structure is duplicated 100 times to produce an array of structured elements.

A variable that is associated with a structure is referred to by an instruction by modifying the memory referencing operand with pathnames of the form

Offset notation.Field name

If `(SI) = 2`, then the third element of the LAST_NAME field in the structure EMPLOYEE_1 defined by the PERSONNEL_DATA structure could be moved into the AL register using the instruction

```
MOV    AL,EMPLOYEE_1.LAST_NAME[SI]
```

Assuming that BX contains the offset of EMPLOYEE_1, the same instruction could be written

```
MOV    AL,[BX].LAST_NAME[SI]
```

The third element of the LAST_NAME field of the fifth duplication in the structure EMPLOYEES [see Fig. 3-60(b)] could be loaded into AL by

```
MOV    AL,EMPLOYEES + 4*12.LAST_NAME[SI]
```

3-10-3 Records

The RECORD directive is for defining a bit pattern within a word or byte. Its use is similar to that of the directive STRUC in that RECORD only creates the pattern and it is left to other statements to invoke the pattern while reserving and preassigning storage. It has the form

Record name RECORD Field specification, . . . ,Field specification
where each field specification is of the form

Field name: Length = Preassignment

with the preassignment being optional. For example,

PATTERN RECORD OPCODE:5,MODE:3,OPR1:4=8,OPR2:4

would break a word into four fields and give them the names OPCODE, MODE, OPR1, and OPR2. The lengths of the fields in bits would be 5, 3, 4, and 4, respectively. The statement

INSTRUCTION PATTERN <,,5>

would actually reserve the word, associate it with the variable INSTRUCTION, and preassign OPR1 to 8 and OPR2 to 5 as shown in Fig. 3-61. The statement

INSTRUCTION_ARRAY PATTERN 100 DUP (< >)

would reserve 100 words, with the first one being associated with the variable INSTRUCTION_ARRAY. In this example the third field in each word is preassigned to 8, but the other fields are not preassigned.

The field names may appear in instructions by themselves or with a MASK or WIDTH operator. The MASK operator causes the operand to be a mask with 1s in the bit positions of the specified field and 0s elsewhere. The WIDTH operator causes the operand to be the length of the specified field. If the field name appears by itself, then the operand is the number of bits needed to right justify the field. The sequence

```
MOV AX,INSTRUCTION
AND AX,MASK OPR1
MOV CL,OPR1
SHR AX,CL
```

would bring in the word INSTRUCTION and right justify the OPR1 field of INSTRUCTION in register AX.

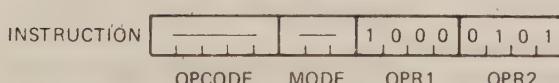


Figure 3-61 Typical use of RECORD to subdivide a word.

3-10-4 Assigning Names to Expressions

If an expression appears several times in a program, it is sometimes more convenient to give it a name and refer to it by the name. Not only does this permit a short name to replace a lengthy expression, but the name may be a meaningful abbreviation that is easy for the programmer and others to remember as they read the program. The statement that assigns a name to an expression has the form

Expression name EQU Expression

where the expression name may be any valid identifier and the expression may have the format of any valid operand, be any expression that evaluates to a constant (the expression name is then a constant name), or be any valid mnemonic. Several examples of expressions are given in Fig. 3-62. Normally, EQU statements are placed toward the beginning of a program. The MOV instruction in the sequence

```
INDXD_CHAR      EQU      CHAR_ARRAY[SI + 10]
.
.
.
MOV      AL,INDXD_CHAR
.
.
```

would be the same as

```
MOV      AL,CHAR_ARRAY[SI + 10]
```

If the expression in an EQU statement contains an identifier (variable, label, constant name, etc.), then either it must constitute the entire expression or it must have been defined by the code prior to the EQU statement. The statement

```
AB      EQU      DATA_ONE
```

is acceptable regardless of where in the code DATA_ONE is defined, but the statement

```
AB      EQU      DATA_ONE + 2
```

Figure 3-62 Examples of the EQU directive.

| Directive | | | Description |
|-----------|-----|--------------|--|
| CONSTANT | EQU | 256 | Assigns 256 the name CONSTANT |
| DATA | EQU | HEIGHT+12 | Assigns the direct operand HEIGHT+12 the name DATA |
| FIELD | EQU | [BX].ETA[SI] | Assigns the structure referencing operand [BX].ETA[SI] to the name FIELD |
| ALPHA | EQU | 7 | This combination would give 7 the name ALPHA, |
| BETA | EQU | ALPHA-2 | 7-2=5 the name BETA, and the |
| ADDR | EQU | VAR+BETA | direct operand VAR+BETA = VAR+5 the name ADDR |

would result in an error if DATA_ONE has not been previously defined. For this reason it is suggested that all data and structure definitions be placed before the EQU statements. However, an EQU statement must appear before the EQU assignment is used; therefore, the suggested order is to put the data definition statements first, the EQU statements next, and the instructions and their associated directives last.

Another important reason for using names in place of instructions is that they permit a program to be easily modified. For example, suppose that the instruction

```
ADD BX,6
```

appears several times within a program and it is known that in some applications of the program the 6 will need to be replaced with a different number. BX may contain an address and the purpose of the instruction may be to increment this address between fields in a structure. If the program must be rewritten to accommodate a different length, then the instructions would need to be changed. However, by including the statement

```
NUM EQU 6
```

at the beginning of the program and writing the ADD instructions as follows:

```
ADD BX,NUM
```

the increment could everywhere be changed to, say 8, by simply replacing the EQU statement with

```
NUM EQU 8
```

This application of the EQU statement is particularly useful in addressing I/O ports because, when the system configuration is changed, these addresses may change.

3-10-5 Segment Definition

As described earlier, a physical memory address is obtained by adding an offset to 16 times a segment address that is contained in a segment register. One of the tasks an assembler must perform is to assign the offsets of the labels and variables as it translates the instructions into machine language. The assembler must also pass to the linker (via the object modules) all of the information that the linker will need in putting the various segments and modules together to form a program. Several directives are designed to instruct the assembler how to perform these functions.

To be able to assign the variable and label offsets the assembler must know the exact structure of each segment. A data, extra data, or stack segment normally has the form

```
Segment name SEGMENT
```

Storage definition, allocation,
and alignment directives

```
Segment name ENDS
```

and a code segment normally has the form

```
Segment name      SEGMENT
                  Instructions and
                  instruction-related directives
Segment name      ENDS
```

The segment name can be any valid identifier. The directives and instructions that appear between the SEGMENT and ENDS directives are considered to be contained in the segment. The overall structure of a program having two data segments and one code segment is shown in Fig. 3-63. (The ASM-86 assembler permits certain modifiers to be appended to the SEGMENT directive, but these modifiers are not considered here, although some of them are discussed in Chap. 4.)

In addition to knowing how the segments are constructed, as the assembler translates the instructions it must know the exact correspondence between the segments and the segment registers. This allows the assembler to check for certain types of errors and inconsistencies, e.g., if a variable is defined in the appropriate data or stack segment. The assignments of the segments to the segment registers are made with directives which are written

```
ASSUME      Assignment, . . . , Assignment
```

where each assignment is written

```
Segment register name:Segment name
```

```
DATA_SEG1  SEGMENT
.
.
.
    } Directives

DATA_SEG1  ENDS
DATA_SEG2  SEGMENT
.
.
.
    } Directives

DATA_SEG2  ENDS
CODE_SEG   SEGMENT
START:
.
.
.
    } Instructions and directives

CODE_SEG   ENDS
END START
```

Figure 3-63 Representative program structure.

The statement

```
ASSUME CS:CODE_SEG,DS:SEG1,ES:SEG2
```

would inform the assembler that it is to assume that the segment address of CODE_SEG is in CS, of SEG1 is in DS, and of SEG2 is in ES. An assignment is not made for SS, presumably because either the stack is not used or the assignment for SS is in a separate ASSUME statement. *It is important to note that the ASSUME directive does not load the segment addresses into the corresponding segment registers.* For all of the segment registers except CS, which is usually loaded by an intersegment branch when the code segment is initiated, this must be done by explicit transfers, normally by MOV instructions. An ASSUME directive makes a promise to the assembler and it must be accompanied by MOV instructions that fulfill the promise.

Referring to the structure given in Fig. 3-63, the code segment might typically begin as follows:

```
CODE_SEG SEGMENT
ASSUME CS:CODE_SEG, DS:DATA_SEG1, ES:DATA_SEG2
START: MOV AX,DATA_SEG1
       MOV DS,AX
       MOV AX,DATA_SEG2
       MOV ES,AX
.
.
.
CODE_SEG ENDS
```

Note that a MOV instruction cannot transfer an immediate operand into a segment register; thus two instructions are needed for each register. Because DATA_SEG1 and DATA_SEG2 are segment names, not variables, the assembler translates the first and third MOV instructions into instructions whose source operands are immediate. The source operands are to be the segment addresses, but since the linker assigns the placement of the segments within memory, the data portions of these instructions are actually assembled by the linker.

The same segment can be assigned to two segment registers, e.g.,

```
ASSUME CS:CODE_SEG,DS:CODE_SEG
```

would cause both CS and DS to be assigned to the same segment. ASSUME statements can also be used to reassign segment registers. In the above example, if the statements

```
ASSUME DS:DATA_SEG3
MOV AX:DATA_SEG3
MOV DS,AX
```

were to appear later in the program, then the assembler would use DATA_SEG3 in place of DATA_SEG1 in translating the succeeding statements, but the assignments for the other segment registers would remain unchanged.

When the assembler encounters a variable in an instruction operand it normally assumes the default segment register according to the addressing mode as

indicated in Fig. 2-15. However, if, say, the default were the DS register and the variable were not in the segment to which the DS register is assigned, then the assembler would search the segments assigned to the CS, ES, and SS registers. If the variable were found, then the assembler inserts a CS, ES, or SS segment override prefix just prior to the instruction containing the variable. When the instruction is executed the 1-byte prefix would cause it to use the CS, ES, or SS register instead of the DS register.

The default segment registers can also be explicitly overridden by putting the abbreviation of the segment register to be used and a colon immediately before the variable. The instruction

ADD AX,ES:ALT

would cause the assembler to insert the segment override prefix

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

just before the machine code for the ADD instruction. In this case the assembler would not need to first search the segment assigned to the DS register. Unless a segment register abbreviation explicitly appears in all operands that use it, it must be included in an ASSUME statement; otherwise, an assembler error will occur. An alternate procedure is to insert the segment name in place of the segment register name, but this can be done only if an ASSUME statement has associated the two entities, e.g., the pair

ASSUME ES:EXTRA_SEG

ADD AX,EXTRA_SEG:ALT

is equivalent to the above ADD statement. The segment register override must be explicitly specified when the operand is defined later on in the program, and is in a segment not pointed by the default segment register. (See Sec. 3-11.)

3-10-6 Program Termination

Just as an END statement is needed to signal the end of a high-level language program, an END directive of the form

END Label

is needed to indicate the end of a set of assembler language code. The label appears only in the termination of the main program in a program structure, although all separately assembled program modules must conclude with an END directive. The label must also appear in the instruction to be executed first, the point at which the program begins. Specifically, the address associated with this label is the address that the loader branches to after the program has been loaded and is ready for execution.

```

DATA SEG    SEGMENT
    OPER1      DW      12
    OPER2      DW      230
    RESULT     DW      ?
DATA SEG    ENDS
CODE SEG   SEGMENT
    ASSUME CS:CODE SEG, DS:DATA SEG
    START:    MOV     AX, DATA SEG
              MOV     DS, AX
              MOV     AX, OPER1
              ADD     AX, OPER2
              JGE     STORE
              NEG     AX
              STORE:  MOV     RESULT, AX
              HLT
CODE SEG   ENDS
END      START

```

Figure 3-64 Complete program.

A complete, albeit simple, program is shown in Fig. 3-64. The program adds the contents of OPER1 and OPER2 and stores the absolute value of the sum in RESULT. It is not a useful program because it has no input or output, but it does demonstrate the structure of complete single module programs and how some of the directives are used.

Figure 3-65 shows how the same program would be structured when the data and code segments are superimposed. The appearance of this program is closer to that of programs written for computers for which segment registers are not available. For the 8086, the structure shown in Fig. 3-64 is preferred because it allows greater flexibility in assigning memory (i.e., the code and data segments are assigned independently).

3-10-7 Alignment Directives

There are two directives that are used for alignment purposes. The directive

EVEN

forces the address of the next byte to be even. It has been noted that for the 8086 words can be accessed in less time if they begin at even addresses. Therefore, just

```

EXAMPLE  SEGMENT
    OPER1      DW      12
    OPER2      DW      230
    RESULT     DW      ?
ASSUME CS:EXAMPLE DS:EXAMPLE
    START:    MOV     AX, CS
              MOV     DS, AX
              MOV     AX, OPER1
              ADD     AX, OPER2
              JGE     STORE
              NEG     AX
              STORE:  MOV     RESULT, AX
              HLT
EXAMPLE  ENDS
END      START

```

Figure 3-65 Complete program with superimposed data and code segments.

before reserving an array of words it is advisable to insert an EVEN directive as follows:

```
DATA_SEG SEGMENT
.
.
.
WORD_ARR DW 100 DUP(?)
.
.
.

DATA_SEG ENDS
```

Note that for the 8088, since each reference to a word operand requires two bus cycles regardless of the operand address, there is no advantage in using even address alignment. The directive

ORG Constant expression

causes the next byte to be the $n + 1$ th byte within a segment, where n is the value of the constant expression. For example, the code

```
VECTORS SEGMENT
.
.
.
ORG 10
VECT1 DW 47A5H
ORG 20
VECT2 DW 0C596H
.
.
.

VECTORS ENDS
```

would cause VECT1 and VECT2 to be associated with the tenth and twentieth bytes of the segment VECTORS.

3-10-8 Value-Returning Attribute Operators

Assembler languages generally include at least some operators that are not required to properly assemble a program, but do make the programmer's task a little easier by providing flexibility, increasing readability, and causing the assembler to fill in some of the numbers. The value-returning attribute operators

LENGTH, SIZE, OFFSET, SEG, and TYPE

fall into this category. All of these operators return values related to the expressions which follow them within their operands.

The LENGTH operator returns the number of units assigned to a variable. The statement

```
FEES DW 100 DUP(0)
```

would cause 100 words to be associated with the variable FEES and the statement

```
MOV CX,LENGTH FEES
```

would be equivalent to

```
MOV CX,100
```

Here, application of the LENGTH operator is advantageous because

```
MOV CX,LENGTH FEES
```

is correct even if the DW directive is changed to reserve 150 words, while the latter form of the instruction would need to be changed to

```
MOV CX,150
```

The SIZE operator is the same as the LENGTH operator except that it returns the number of bytes instead of the number of units. Assuming the above DW statement, the instruction

```
MOV CX,SIZE FEES
```

would cause 200 to be put into CX.

The OFFSET operator returns the value of the offset of a label or variable. The instruction

```
MOV BX,OFFSET OPER_ONE
```

results in the offset of OPER_ONE being assembled as an immediate operand and, at execution time, the offset of OPER_ONE is loaded into BX. This instruction is equivalent to

```
LEA BX,OPER_ONE
```

The SEG operator similarly causes the segment address of the variable or label to be inserted as an immediate operand (although the actual insertion is made by the linker). If DATA_SEG is assigned to the block of memory beginning at 05000 and OPER1 is in DATA_SEG, then the instruction

```
MOV BX,SEG OPER1
```

would put 0500 in BX.

The TYPE operator is used primarily with variables and structure names and returns the number of bytes associated with the variable or structure. The instruction

```
ADD SI,TYPE ARRAY
```

would add 1 to the SI register if ARRAY is defined in a DB statement, 2 if defined in a DW statement, 4 if defined in a DD statement, and the number of bytes in the structure if ARRAY is a structure name. TYPE seldom appears with a label, but when it does -1 is returned for a NEAR label and -2 for a FAR label.

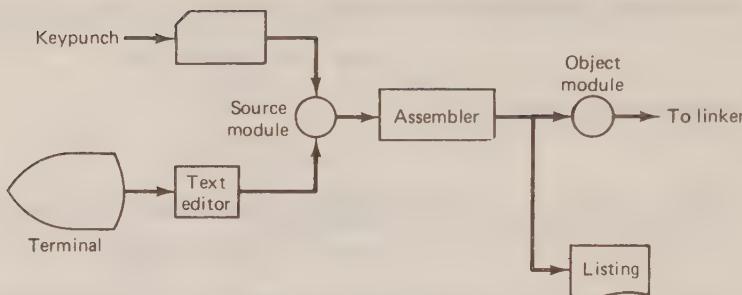


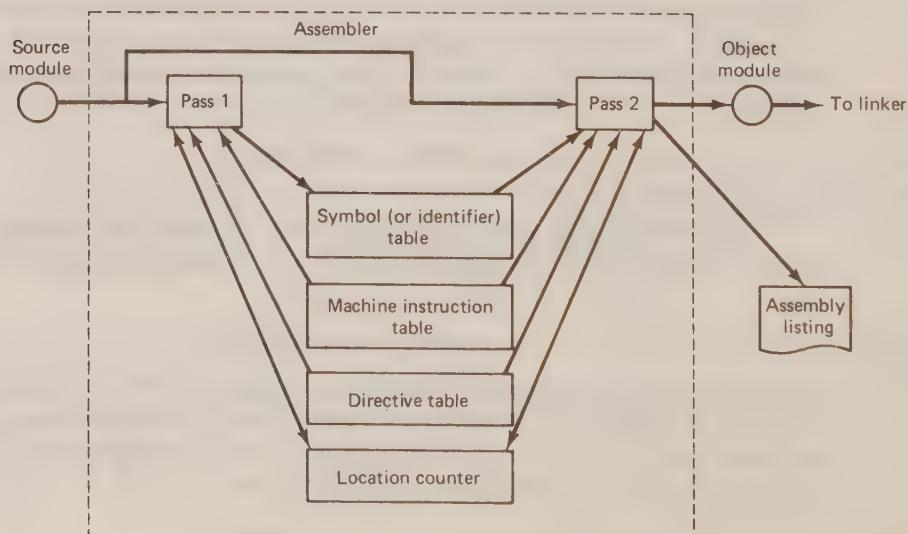
Figure 3-66 Assembler's input and output.

3-11 ASSEMBLY PROCESS

In an effort to clarify the structure of assembler language programs and why certain rules are necessary, this section examines the assembly process. As shown in Fig. 3-66, the input to the assembler is a source module and the output is an object module and a listing. The source module is normally a file on a magnetic disk or tape that has been input from a deck of cards, with one assembler statement on each card, or has been created using a text editor by typing one statement per line. The object module, which consists of the machine language code and the information needed to join the module to other object modules, is stored as a file.

A breakdown of an assembler is shown in Fig. 3-67. (It should be emphasized that the assembler discussed here is hypothetical and is simplified so that the salient

Figure 3-67 Two-pass assembler.



points are evident. The details of an actual assembler would only add clutter to this introductory discussion.) As with most assemblers, the one given here scans the source code twice and is called a *two-pass assembler*. The purpose of the first pass is to provide the assembler with the locations of the identifiers and that of the second pass is to generate the machine code. To locate the identifiers, the assembler contains a variable known as a *location counter*. As a program is being scanned its location counter increments by amounts equal to the number of bytes required by the statements. When the program passes from one segment to another the location counter is set to 0. The location counter can be thought of as a pointer that dynamically indicates the relative positions, or offsets, within each segment as the assembly progresses. During the first pass the assembler uses the location counter to construct a table, called a *symbol* (or *identifier*) *table*, that allows the second pass to use the offsets of the identifiers to generate operand addresses.

For the code

| | | | Location Counter | Length in Bytes |
|----------|--------------------------|--|------------------|-----------------|
| CODE_SEG | SEGMENT | | 0 | 0 |
| ASSUME | CS:CODE_SEG, DS:DATA_SEG | | 0 | 0 |
| START: | MOV AX, DATA_SEG | | 0 | 3 |
| | MOV DS, AX | | 3 | 2 |
| | MOV CX, COUNT | | 5 | 4 |
| REPEAT: | DEC CX | | 9 | 1 |
| | | | 10 | . |
| | | | . | . |

the location counter is, as always, initially 0. Because the first two statements are directives that do not take up space in the machine code, the location counter does not change when they are scanned by the assembler. The next four statements are instructions that translate into machine instructions whose lengths are 3, 2, 4, and 1, respectively. Therefore, they will cause the location counter to increment to 3, then to 5, then to 9, then to 10, and so on. Similarly, the following segment would cause the location counter to increment according to the number of bytes reserved by each directive:

| | | | Location Counter | Number of Bytes Allocated |
|----------|----------------|--|------------------|---------------------------|
| DATA_SEG | SEGMENT | | 0 | 0 |
| | ORG 10 | | 0 | 10 |
| NUM | DB -29 | | 10 | 1 |
| ARRAY | DB 100 DUP(?) | | 11 | 100 |
| COUNT | DW 5 | | 111 | 2 |
| MASKS | DB 82H,04H,2AH | | 113 | 3 |
| | | | 116 | . |
| | | | . | . |

For the above segments, the first pass of the assembler uses the location counter to enter

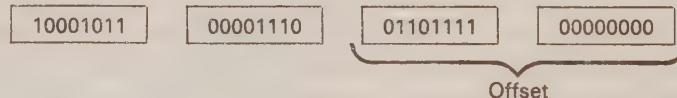
| Symbol | Offset | Segment in which it was Defined | Type |
|----------|--------|---------------------------------|---------------|
| DATA_SEG | 00H | | Segment |
| NUM | 0AH | DATA_SEG | Byte Variable |
| ARRAY | 0BH | DATA_SEG | Byte variable |
| COUNT | 6FH | DATA_SEG | Word Variable |
| MASKS | 71H | DATA_SEG | Byte Variable |

| | | | |
|----------|-----|----------|------------|
| CODE_SEG | 00H | | Segment |
| START | 00H | CODE_SEG | Near Label |
| REPEAT | 09H | CODE_SEG | Near Label |

into the symbol table. Associated with each identifier, the symbol table also includes the type and the name of the segment in which the identifier is defined. The second pass then accesses this information when assembling instructions whose operands include these identifiers. For example, while constructing the machine code for the instruction

MOV CX,COUNT

the assembler would check if the source type matches with the destination type and if COUNT is accessible through DS. Then, the assembler would note that the offset for COUNT is 006F($=111_{10}$) and would produce the machine instruction



Offset

The assembler also includes two tables, known as *permanent symbol tables*, that contain all of the information concerning the instructions and directives that might be needed by the assembler. Among other things these tables contain the mnemonics, operation codes and associated formats, and the length information that is required for incrementing the location counter.

The major functions of the first and second passes of an assembler are flowcharted in Figs. 3-68 and 3-69. When a label or variable is found in the leftmost field of a statement it is said to be *defined*, and it is at this point that the label or variable is associated with an offset by placing the current contents of the location counter in the symbol table along with the label or variable's symbolic representation and other attributes. If the same label or variable is defined twice in a source module, an error occurs. An error will also occur if an instruction or directive mnemonic is not in one of the permanent symbol tables. When the first pass encounters the END directive it is terminated and the second pass begins.

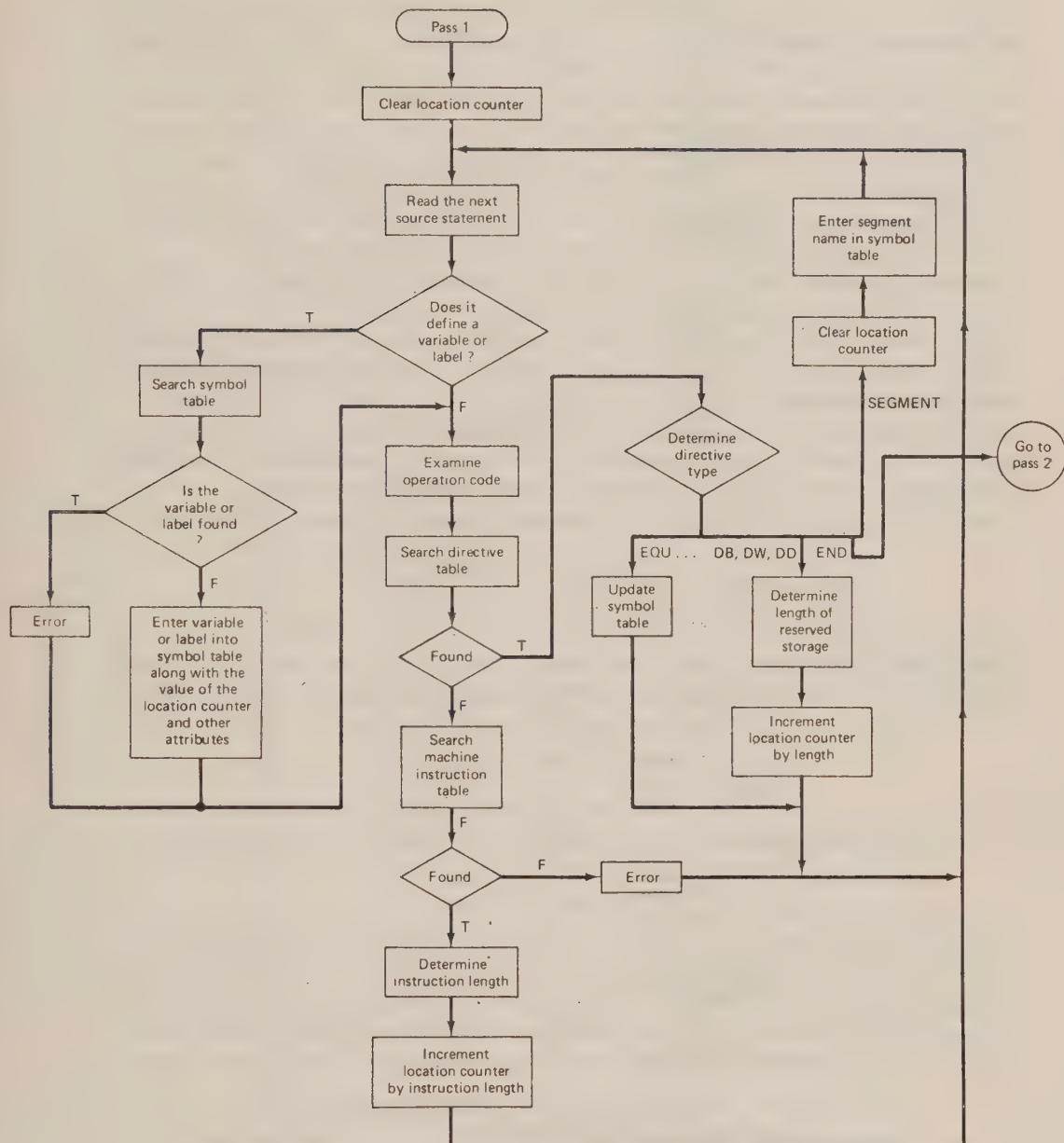


Figure 3-68 Major logic flow of the first pass.

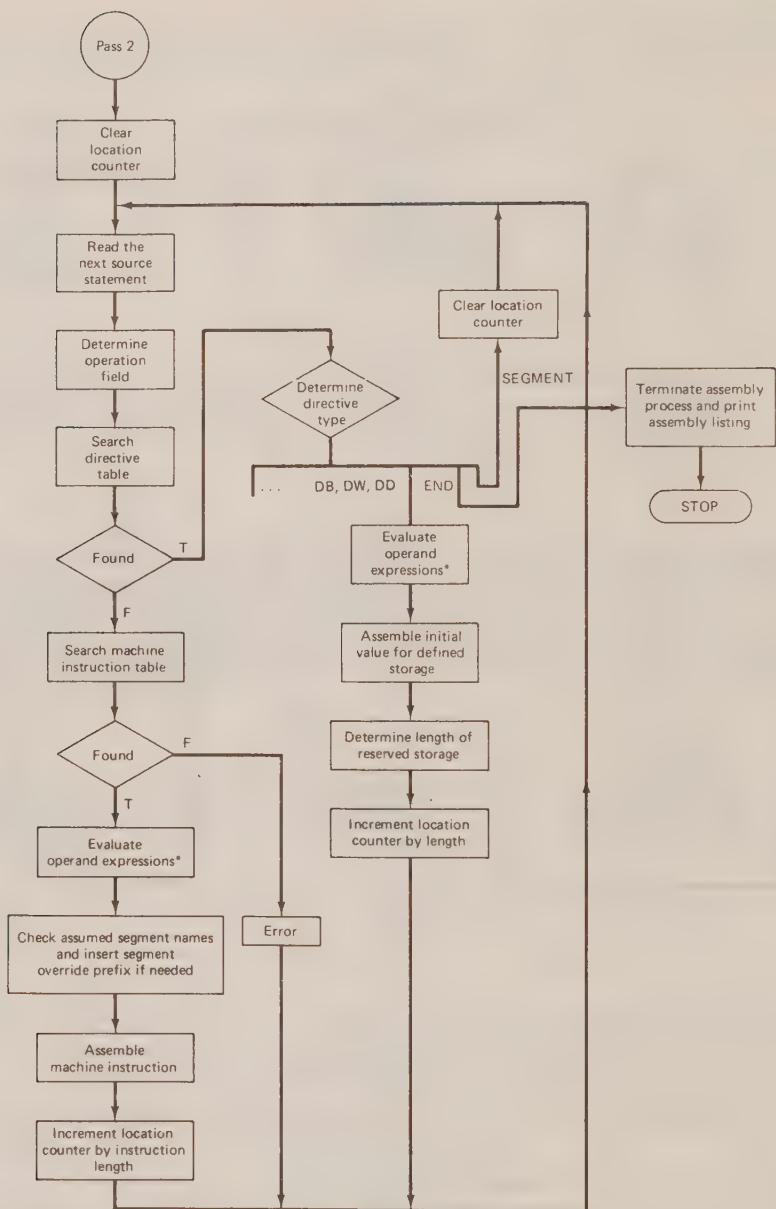


Figure 3-69 Major logic flow of the second pass.

In addition to assembling the machine instructions, the second pass must insert the preassigned constants that occur in the data definition statements and prepare the other information that will be required by the linker. Because the operands in the instructions and directives may involve expressions, the computation of an offset may involve more than simply looking up an entry in the symbol table. The second pass is also responsible for performing these computations. When the second pass encounters the END directive, the object module and listing are output to the appropriate peripheral devices and the assembly process is terminated.

If at the time a variable or label appears in an operand it has already been defined (i.e., has been in the leftmost field of a prior statement), then its appearance is referred to as a *backward reference*; otherwise, it is called a *forward reference*. Forward references are somewhat more difficult for the assembler to handle. The reason is that the length of the instruction may depend on the attributes of the operand and, unless the attributes are explicitly indicated, the assembler may need to guess how much to increment the location counter. An incorrect guess would either cause some assembly errors or force the second pass to compensate for these errors by assembling nonoptimal code. The attributes may be explicitly indicated by the attribute operators and segment prefixes such as SHORT, PTR, ES:, etc.

For example, consider the instruction

JMP EXIT

If EXIT is a forward reference, at the time the instruction is encountered the assembler has no way of knowing whether the branch is to have the short, near, or far attribute. By default, the assembler assumes EXIT to be a near label and reserves 3 bytes for the instruction. On the other hand,

JMP FAR PTR EXIT

leaves no doubt that the instruction is 5 bytes long.

A second example depicts a different but equally ambiguous situation. Consider

MOV AX,OPERAND

and suppose that OPERAND is a forward reference to a segment other than the DS segment (e.g., OPERAND is defined in the segment pointed to by SS). By the absence of other directions, during pass 1 the assembler would assume that the DS segment register is to be used and would not take into account the required override prefix byte. This will cause an error during pass 2. However, the instruction

MOV AX,SS:OPERAND

or the combination

ASSUME SS:DATA_SEG2

MOV AX,DATA_SEG2:OPERAND *

would explicitly inform the assembler that an override prefix must be inserted.

To avoid the problems and resulting errors associated with forward references it is recommended that:

1. Attribute operators be used to explicitly specify the types of operands in all JMP instructions.
2. All data definition directives be placed before the instructions.

Item 2 implies that, within a given source module, all data-related segments should appear above the code segments. Forward references can similarly cause complications in deciphering EQU statements. Therefore, a third recommendation is:

3. All EQU statements should be placed between the data-related directives and the instructions. If the expression in one EQU statement refers to a name in another EQU statement, the reference should be a backward reference.

These recommendations match the requirements of FORTRAN and other high-level languages for which storage allocation, preassignment, and parameter definition statements must precede the executable statements.

The ASM-86 assembler, as do most other assemblers, permits defaults which tend to skirt around the problems introduced above, but the overuse of defaults often leads to poor programming habits and programs that are difficult to read. It is best not to use defaults that cannot be immediately understood by anyone who might need to review the program. Only those defaults that are considered helpful are explained in this book.

Most assemblers are designed to accept a symbol that allows the programmer to directly reference the location counter. For the ASM-86 assembler this symbol is "\$". The instruction

JNE \$+6

would cause the assembler to compute the offset of the branch address by adding 6 to the current contents of the location counter. Although the location counter symbol is seldom used, its existence deserves mention. One problem with applying such symbols is seen by realizing that in the above example, if the code between the JNE instruction and the point 4 bytes beyond this instruction is changed, then the branch address computed from \$+6 may be incorrect. The location counter symbol is more likely to be used in directives such as

ORG OFFSET \$+100

This directive would serve only to increment the location counter by 100. Even in the second pass nothing would be assembled into the 100 bytes that are skipped over by this directive. The directive would offset the assembly in exactly the same way as

DB 100 DUP(?)

Besides the object module, the assembler outputs a listing that contains information concerning the program and its translation. Depending on the command given to invoke the assembler, this listing may consist only of a list of the errors

detected during the assembly or include a complete detailed listing of the program, or something in between. A complete listing, including machine code, normally contains a listing of the program, with error messages interspersed with the code, and a cross-reference symbol table.

A typical listing is shown in Fig. 3-70. The first column in the program portion of the listing gives the value of the location counter immediately before the corresponding statement is assembled. The second column shows the machine code that the statement is assembled into, the third column is simply the line number in the source code, and the remainder of each line is the source code just as it is presented to the assembler. If an error is found, an error identifying number and message are output on the line following the line containing the error.

Note that in line 8 the MOV instruction has not been completely assembled. As mentioned in Sec. 3-10-5, if the second operand is the name of a segment, then it represents the segment address, not the contents of a memory location. Since the beginning addresses of the segments are not known until a program is linked, the assembler cannot completely construct this MOV instruction. The "R" in this line means "relocatable" and is related to the fact that the linker must help determine the address. The term "relocatable" is discussed in Chap. 4 when the linking process is considered.

Figure 3-70 Sample assembler listing.

| LOC | OBJ | LINE | SOURCE |
|---------------|-----|------|---------------------------------|
| ---- | | 1 | DATA SEG SEGMENT |
| 0000 0C00 | | 2 | OPER1 DW 12 |
| 0002 E600 | | 3 | OPER2 DW 230 |
| 0004 ???? | | 4 | RESULT DW ? |
| ---- | | 5 | DATA SEG ENDS |
| ---- | | 6 | CODE SEG SEGMENT |
| 0000 B8---- | R | 7 | ASSUME CS:CODE SEG, DS:DATA SEG |
| 0003 8ED8 | | 8 | START: MOV AX, DATA SEG |
| 0005 A10000 | | 9 | MOV DS, AX |
| 0008 03060200 | | 10 | MOV AX, OPER1 |
| 000C 7D02 | | 11 | ADD AX, OPER2 |
| 000E F7D8 | | 12 | JGE STORE |
| 0010 A30400 | | 13 | NEG AX |
| 0013 F4 | | 14 | STORE: MOV RESULT, AX |
| ---- | | 15 | |
| 0000 | | 16 | CODE SEG ENDS |
| | | 17 | END START |

XREF SYMBOL TABLE LISTING

| NAME | TYPE | VALUE | ATTRIBUTES, XREFS |
|-----------|---------|-------|--------------------------|
| ?SEG .. | SEGMENT | | SIZE=0000H PARA PUBLIC |
| CODE SEG. | SEGMENT | | SIZE=0014H PARA 6# 7 16 |
| DATA SEG. | SEGMENT | | SIZE=0006H PARA 1# 5 7 8 |
| OPER1 .. | V WORD | 0000H | DATA SEG 2# 10 |
| OPER2 .. | V WORD | 0002H | DATA SEG 3# 11 |
| RESULT .. | V WORD | 0004H | DATA SEG 4# 14 |
| START .. | L NEAR | 0000H | CODE SEG 8# 17 |
| STORE .. | L NEAR | 0010H | CODE SEG 12 14# |

The cross-reference symbol table summarizes the information regarding the identifiers in the program. For each segment name this table indicates the size of the segment in hexadecimal and the line numbers of the lines in which the name appears. The line number of the segment directive containing the name is marked with the symbol "#". The variables and labels are accompanied by letters indicating whether they are a variable (V) or label (L), an attribute designation, the offset assigned to them, the name of the segment in which they are defined, and the line numbers of the lines in which they appear. As with the segment names, the lines in which they are defined are marked with "#" signs. For example, OPER2 is a word variable that is defined in the segment DATA_SEG. Its offset within DATA_SEG is 0002H. It appears in lines 3 and 11 of the listing and is defined in line 3. Similarly, STORE is seen to be a near label that is defined in CODE_SEG and has an offset of 0010H within CODE_SEG. It appears in lines 12 and 14 and is defined in line 14. The reader should carefully verify all entries in the symbol table and the machine code in the program listing.

3-12 TRANSLATION OF ASSEMBLER INSTRUCTIONS

The translation from assembler instructions to machine instructions is, in most cases, quite straightforward. All of the 8086 machine instructions consist of 1 or 2 bytes of op code and addressing mode designations with from 0 to 4 bytes of immediate, displacement, or segment address information appended to them. The number of appended bytes depends on the addressing modes. If the addressing modes call for both an immediate operand and a displacement the displacement will appear first, and if both an offset and a segment address are present, the offset will appear first.

The machine code formats of all of the 8086/8088 instructions, including those that are defined in the later chapters, are given in Fig. 3-71. For convenience, a summary of the register addresses, addressing modes, etc., is also included in this figure. All but one of the double-operand instructions (MOV, ADD, ADC, SUB, SBB, CMP, AND, OR, XOR, and TEST) in which the source operand is immediate and the other operand is the AX or AL register (i.e., accumulator) have both a long form and a short form. In the short form the AX (for word operations) or AL (for byte operations) is implied by the op code. In addition, the MOV instruction has a short form for any immediate to register transfer. The XCHG instruction has a short form if the exchange is between a register and AX or AL. Several of the single-operand instructions (INC, DEC, PUSH, and POP) have short forms if the operand is a register.

In all cases the short form is one byte shorter than the corresponding long form, which in some instances implies that the instruction is only 1 byte in length. As a general rule, the assembler will always assemble the short form, if possible. For example,

CMP AX,5A2H

Figure 3-71 Machine code for the 8086/8088 instructions. (Reprinted by permission of Intel Corporation. Copyright 1979.)

AL = 8-bit accumulator
 AX = 16-bit accumulator
 CX = Count register
 DS = Data segment
 ES = Extra segment
 Above/below refers to unsigned value
 Greater = more positive;
 Less = less positive (more negative) signed values
 if d = 1 then "to" reg; if d = 0 then "from" reg
 if w = 1 then word instruction; if w = 0 then byte instruction

if s:w = 01 then 16 bits of immediate data form the operand
 if s:w = 11 then an immediate data byte is sign extended to
 form the 16-bit operand.
 if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
 x = don't care
 z is used for string primitives for comparison with ZF FLAG
SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

if mod = 11 then r/m is treated as a REG field
 if mod = 00 then DISP = 0*, disp-low and disp-high are absent
 if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
 if mod = 10 then DISP = disp-high: disp-low
 if r/m = 000 then EA = (BX) + (SI) + DISP
 if r/m = 001 then EA = (BX) + (DI) + DISP
 if r/m = 010 then EA = (BP) + (SI) + DISP
 if r/m = 011 then EA = (BP) + (DI) + DISP
 if r/m = 100 then EA = (SI) + DISP
 if r/m = 101 then EA = (DI) + DISP
 if r/m = 110 then EA = (BP) + DISP*
 if r/m = 111 then EA = (BX) + DISP
 DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

REG is assigned according to the following table

| 16-Bit [w 1] | 8-Bit (w = 0) | Segment |
|--------------|---------------|---------|
| 000 AX | 000 AL | 00 ES |
| 001 CX | 001 CL | 01 CS |
| 010 DX | 010 DL | 10 SS |
| 011 BX | 011 BL | 11 DS |
| 100 SP | 100 AH | |
| 101 BP | 101 CH | |
| 110 SI | 110 DH | |
| 111 DI | 111 BH | |

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS : X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF) X:(PF) X:(CF)

8086/8088 Instruction Encoding

DATA TRANSFER

MOV = Move:

| 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1 0 0 0 1 0 d w | mod | reg | r/m | (DISP-LO) | (DISP-HI) | |
| 1 1 0 0 0 1 1 w | mod 0 0 0 | r/m | | (DISP-LO)* | (DISP-HI) | data |
| 1 0 1 1 1 w reg | | data | | data if w = 1 | | |
| 1 0 1 0 0 0 0 w | | addr-lo | | addr-hi | | |
| 1 0 1 0 0 0 1 w | | addr-lo | | addr-hi | | |
| 1 0 0 0 1 1 1 0 | mod 0 SR | r/m | | (DISP-LO) | (DISP-HI) | |
| 1 0 0 0 1 1 0 0 | mod 0 SR | r/m | | (DISP-LO) | (DISP-HI) | |

PUSH = Push:

| | | | | |
|-----------------|-----------|-----|-----------|-----------|
| 1 1 1 1 1 1 1 1 | mod 1 1 0 | r/m | (DISP-LO) | (DISP-HI) |
| 0 1 0 1 0 reg | | | | |
| 0 0 0 reg 1 1 0 | | | | |

POP = Pop:

| | | | | |
|-------------------|-----------|-----|-----------|-----------|
| 1 0 0 0 0 1 1 1 1 | mod 0 0 0 | r/m | (DISP-LO) | (DISP-HI) |
| 0 1 0 1 1 reg | | | | |
| 0 0 0 reg 1 1 1 | | | | |

Figure 3-71 Continued.

DATA TRANSFER (Cont'd.)

XCHG = Exchange:

7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0 7 8 5 4 3 2 1 0

Register/memory with register

| | | | | | |
|-----------------|-----|-----|-----|-----------|-----------|
| 1 0 0 0 0 1 1 w | mod | reg | r/m | (DISP-LO) | (DISP-HI) |
| 1 0 0 1 0 reg | | | | | |

IN = Input from:

| | |
|-----------------|--------|
| 1 1 1 0 0 1 0 w | DATA-8 |
| 1 1 1 0 1 1 0 w | |

OUT = Output to:

| | | | | | |
|-----------------|--------|-----|-----|-----------|-----------|
| 1 1 1 0 0 1 1 w | DATA-8 | | | | |
| 1 1 1 0 1 1 1 w | | | | | |
| 1 1 0 1 0 1 1 1 | | | | | |
| 1 0 0 0 1 1 0 1 | mod | reg | r/m | (DISP-LO) | (DISP-HI) |
| 1 1 0 0 0 1 0 1 | mod | reg | r/m | (DISP-LO) | (DISP-HI) |
| 1 1 0 0 0 1 0 0 | mod | reg | r/m | (DISP-LO) | (DISP-HI) |
| 1 0 0 1 1 1 1 1 | | | | | |
| 1 0 0 1 1 1 1 0 | | | | | |
| 1 0 0 1 1 1 0 0 | | | | | |
| 1 0 0 1 1 1 0 1 | | | | | |

基础训练

ADD = Add:

| | | | | | |
|-----------------|---------------|-------------|-----------|------|----------------|
| 0 0 0 0 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| 1 0 0 0 0 0 s w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | data if s:w=01 |
| 0 0 0 0 0 1 0 w | data | data if w=1 | | | |

ADC = Add with carry:

| | | | | | |
|-----------------|---------------|-------------|-----------|------|----------------|
| 0 0 0 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| 1 0 0 0 0 0 s w | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) | data | data if s:w=01 |
| 0 0 0 1 0 1 0 w | data | data if w=1 | | | |

INC = Increment

| | | | |
|-----------------|---------------|-----------|-----------|
| 1 1 1 1 1 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) |
| 0 1 0 0 0 reg | | | |
| 0 0 1 1 0 1 1 1 | | | |
| 0 0 1 0 0 1 1 1 | | | |

Figure 3-71 Continued.

ARITHMETIC (Cont'd.)**SUB = Subtract:**

Reg/memory and register to either

| | | | | | |
|-----------------|---------------|-------------|-----------|------|----------------|
| 0 0 1 0 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| 1 0 0 0 0 0 s w | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) | data | data if s w=01 |
| 0 0 1 0 1 1 0 w | data | data if w=1 | | | |

SBB = Subtract with borrow:

Reg/memory and register to either

| | | | | | |
|-----------------|---------------|-------------|-----------|------|----------------|
| 0 0 0 1 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| 1 0 0 0 0 0 s w | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | data | data if s w=01 |
| 0 0 0 1 1 1 0 w | data | data if w=1 | | | |

DEC Decrement:

Register/memory

| | | | | | |
|-----------------|---------------|-----------|-----------|--|--|
| 1 1 1 1 1 1 1 w | mod 0 0 1 r/m | (DISP-LO) | (DISP-HI) | | |
| 0 1 0 0 1 reg | | | | | |
| 1 1 1 1 0 1 1 w | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | | |

CMP = Compare:

Register/memory and register

| | | | | | |
|-----------------|-----------------|-------------|-----------|------|----------------|
| 0 0 1 1 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | |
| 1 0 0 0 0 0 s w | mod 1 1 1 r/m | (DISP-LO) | (DISP-HI) | data | data if s w=01 |
| 0 0 1 1 1 1 0 w | data | data if w=1 | | | |
| 0 0 1 1 1 1 1 1 | | | | | |
| 0 0 1 0 1 1 1 1 | | | | | |
| 1 1 1 1 0 1 1 w | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 1 1 0 1 1 w | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 0 1 0 1 0 0 | 0 0 0 0 1 0 1 0 | | | | |
| 1 1 1 1 0 1 1 w | mod 1 1 0 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 1 1 0 1 1 w | mod 1 1 1 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 0 1 0 1 0 1 | 0 0 0 0 1 0 1 0 | | | | |
| 1 0 0 1 1 0 0 0 | | | | | |
| 1 0 0 1 1 0 0 1 | | | | | |

LOGIC**NOT Invert**

| | | | | | |
|-----------------|---------------|-----------|-----------|--|--|
| 1 1 1 1 0 1 1 w | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 0 1 0 0 v w | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 0 1 0 0 v w | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 0 1 0 0 v w | mod 1 1 1 r/m | (DISP-LO) | (DISP-HI) | | |
| 1 1 0 1 0 0 v w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | | |

Figure 3-71 Continued.

LOGIC (Cont'd.)

ROR Rotate right

RCL Rotate through carry flag left

RCR Rotate through carry right

| | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 |
|-----|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| ROR | 1 1 0 1 0 0 v w | mod 0 0 1 r/m | (DISP-LO) | (DISP-HI) | | | |
| RCL | 1 1 0 1 0 0 v w | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) | | | |
| RCR | 1 1 0 1 0 0 v w | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) | | | |

AND = And:

Reg/memory with register to either

Immediate to register/memory

Immediate to accumulator

| | | | | | | |
|-----------------|---------------|-----------|-------------|------|--|-------------|
| 0 0 1 0 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | | |
| 1 0 0 0 0 0 0 w | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) | data | | data if w=1 |
| 0 0 1 0 0 1 0 w | | data | data if w=1 | | | |

TEST = And function to flags no result:

Register/memory and register

Immediate data and register/memory

Immediate data and accumulator

| | | | | | | |
|-----------------|---------------|-----------|-------------|------|--|-------------|
| 1 0 0 0 0 1 0 w | mod reg r/m | (DISP-LO) | (DISP-HI) | | | |
| 1 1 1 1 0 1 1 w | mod 0 0 0 r/m | (DISP-LO) | (DISP-HI) | data | | data if w=1 |
| 1 0 1 0 1 0 0 w | | data | data if w=1 | | | |

OR = Or:

Reg/memory and register to either

Immediate to register/memory

Immediate to accumulator

| | | | | | | |
|-----------------|---------------|-----------|-------------|------|--|-------------|
| 0 0 0 0 1 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | | |
| 1 0 0 0 0 0 0 w | mod 0 0 1 r/m | (DISP-LO) | (DISP-HI) | data | | data if w=1 |
| 0 0 0 0 1 1 0 w | | data | data if w=1 | | | |

XOR = Exclusive or:

Reg/memory and register to either

Immediate to register/memory

Immediate to accumulator

| | | | | | | |
|-----------------|---------------|-----------|-------------|------|--|-------------|
| 0 0 1 1 0 0 d w | mod reg r/m | (DISP-LO) | (DISP-HI) | | | |
| 1 0 0 0 0 0 0 w | mod 1 1 0 r/m | (DISP-LO) | (DISP-HI) | data | | data if w=1 |
| 0 0 1 1 0 1 0 w | | data | data if w=1 | | | |

STRING MANIPULATION

REP = Repeat

MOVS = Move byte/word

CMPS = Compare byte/word

SCAS = Scan byte/word

LODS = Load byte/wd to AL/AH

STDS = Stor byte/wd from AL/AH

| |
|-----------------|
| 1 1 1 1 0 0 1 z |
| 1 0 1 0 0 1 0 w |
| 1 0 1 0 0 1 1 w |
| 1 0 1 0 1 1 1 w |
| 1 0 1 0 1 1 0 w |
| 1 0 1 0 1 0 1 w |

Figure 3-71 Continued.

CONTROL TRANSFER**CALL = Call:** 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

| | | | | |
|-------------------------|-----------------|---------------|-----------|-----------|
| Direct within segment | 1 1 1 0 1 0 0 0 | IP-INC-LO | IP-INC-HI | |
| Indirect within segment | 1 1 1 1 1 1 1 1 | mod 0 1 0 r/m | (DISP-LO) | (DISP-HI) |
| Direct intersegment | 1 0 0 1 1 0 1 0 | IP-Lo | IP-hi | |
| Indirect intersegment | | CS-Lo | CS-hi | |
| | 1 1 1 1 1 1 1 1 | mod 0 1 1 r/m | (DISP-LO) | (DISP-HI) |

JMP = Unconditional Jump:

| | | | | |
|-----------------------------|-----------------|---------------|-----------|-----------|
| Direct within segment | 1 1 1 0 1 0 0 1 | IP-INC-LO | IP-INC-HI | |
| Direct within segment-short | 1 1 1 0 1 0 1 1 | IP-INC8 | | |
| Indirect within segment | 1 1 1 1 1 1 1 1 | mod 1 0 0 r/m | (DISP-LO) | (DISP-HI) |
| Direct intersegment | 1 1 1 0 1 0 1 0 | IP-Lo | IP-hi | |
| Indirect intersegment | | CS-Lo | CS-hi | |
| | 1 1 1 1 1 1 1 1 | mod 1 0 1 r/m | (DISP-LO) | (DISP-HI) |

RET = Return from CALL:

| | | | |
|---|-----------------|---------|---------|
| Within segment | 1 1 0 0 0 0 1 1 | | |
| Within seg adding immmed to SP | 1 1 0 0 0 0 1 0 | data-Lo | data-hi |
| Intersegment | 1 1 0 0 1 0 1 1 | | |
| Intersegment adding immediate to SP | 1 1 0 0 1 0 1 0 | data-Lo | data-hi |
| JE/JZ = Jump on equal/zero | 0 1 1 1 0 1 0 0 | IP-INC8 | |
| JL/JNGE = Jump on less/not greater or equal | 0 1 1 1 1 1 0 0 | IP-INC8 | |
| JLE/JNG = Jump on less or equal/not greater | 0 1 1 1 1 1 1 0 | IP-INC8 | |
| JB/JNAE = Jump on below/not above or equal | 0 1 1 1 0 0 1 0 | IP-INC8 | |
| JBE/JNA = Jump on below or equal/not above | 0 1 1 1 0 1 1 0 | IP-INC8 | |
| JP/JPE = Jump on parity/parity even | 0 1 1 1 1 0 1 0 | IP-INC8 | |
| JO = Jump on overflow | 0 1 1 1 0 0 0 0 | IP-INC8 | |
| JS = Jump on sign | 0 1 1 1 1 0 0 0 | IP-INC8 | |
| JNE/JNZ = Jump on not equal/not zero | 0 1 1 1 0 1 0 1 | IP-INC8 | |
| JNL/JGE = Jump on not less/greater or equal | 0 1 1 1 1 1 0 1 | IP-INC8 | |
| JNLE/JG = Jump on not less or equal/greater | 0 1 1 1 1 1 1 1 | IP-INC8 | |
| JMB/JAE = Jump on not below/above or equal | 0 1 1 1 0 0 1 1 | IP-INC8 | |
| JNBE/JA = Jump on not below or equal/above | 0 1 1 1 0 1 1 1 | IP-INC8 | |
| JNP/JPO = Jump on not par/par odd | 0 1 1 1 1 0 1 1 | IP-INC8 | |
| JNO = Jump on not overflow | 0 1 1 1 0 0 0 1 | IP-INC8 | |

Figure 3-71 Continued.

CONTROL TRANSFER (Cont'd.)

JNS = Jump on not sign

7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

LOOP = Loop CX times

| | |
|-----------------|---------|
| 0 1 1 1 1 0 0 1 | IP-INC8 |
| 1 1 1 0 0 0 1 0 | IP-INC8 |
| 1 1 1 0 0 0 0 1 | IP-INC8 |
| 1 1 1 0 0 0 0 0 | IP-INC8 |
| 1 1 1 0 0 0 1 1 | IP-INC8 |

LOOPZ/LOOPE = Loop while zero/equal

_LOOPNZ/LOOPNE = Loop while not zero/equal

JCXZ = Jump on CX zero

INT = Interrupt:

Type specified

| | |
|-----------------|--------|
| 1 1 0 0 1 1 0 1 | DATA-8 |
| 1 1 0 0 1 1 0 0 | |
| 1 1 0 0 1 1 1 0 | |
| 1 1 0 0 1 1 1 1 | |

Type 3

INTO = Interrupt on overflow

IRET = Interrupt return

PROCESSOR CONTROL

CLC = Clear carry

1 1 1 1 1 0 0 0

CMC = Complement carry

1 1 1 1 0 1 0 1

STC = Set carry

1 1 1 1 1 0 0 1

CLD = Clear direction

1 1 1 1 1 1 0 0

STD = Set direction

1 1 1 1 1 1 0 1

CLI = Clear interrupt

1 1 1 1 1 0 1 0

STI = Set interrupt

1 1 1 1 1 0 1 1

HLT = Halt

1 1 1 1 0 1 0 0

WAIT = Wait

1 0 0 1 1 0 1 1

ESC = Escape (to external device)

1 1 0 1 1 x x x mod y y r / m (DISP-LO) (DISP-HI)

LOCK = Bus lock prefix

1 1 1 1 0 0 0 0

SEGMENT = Override prefix

0 0 1 reg 1 1 0

would be assembled as

00111101

10100010

00000101

instead of

10000001

11111000

10100010

00000101

BIBLIOGRAPHY

1. *MCS-86 User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
2. *The 8086 Family User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
3. *MCS-86 Macro Assembly Language Reference Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
4. Rector, Russell, and George Alexy, *The 8086 Book* (Berkeley, Calif.: Osborne/McGraw-Hill, 1980).
5. Morse, Stephen P., *The 8086 Primer: An Introduction to Its Architecture, System Design, and Programming* (Rochelle Park, N.J.: Hayden Book Company, Inc., 1978).
6. Donovan, John J., *Systems Programming* (New York: McGraw-Hill Book Company, 1972).

EXERCISES

1. Assume that LAB is a label, VAR is a variable, and CON is the name of a constant, and identify all possible addressing modes for each of the following operands.

| | | | |
|-----------------|-----------------|-------------------------|---------|
| (a) VAR[BX] | (b) CON + 63H | (c) LAB | (d) VAR |
| (e) VAR[BX + 6] | (f) VAR[BX][SI] | (g) VAR[BX + 3][DI + 9] | |
2. Explain why the PTR attribute operator is sometimes necessary.
3. Write a program sequence that will put 75_{16} into register DX and then transfer (DX) to each of the three words beginning at the location associated with the word BLOCK. Draw a figure similar to Fig. 3-8(b) which shows the action that takes place as this sequence is executed.
4. Write a program sequence that will reverse the contents of the 4 bytes LIST through LIST + 3.
5. Which of the following are invalid assembler language instructions? State the error for each invalid instruction. Assume that all identifiers are variables and are associated with words.

```

MOV    BP,AL
MOV    WORD_OP[BX + 4*3][DI],SP
MOV    WORD_OP1,WORD_OP2
MOV    AX,WORD_OP1[DX]
MOV    CS,AX
MOV    DS,BP
MOV    SAVE_WORD,DS
MOV    SP,SS:DATA_WORD[BX][SI]
MOV    [BX][SI],2
MOV    AX,WORD_OP1+WORD_OP2
MOV    AX,WORD_OP1-WORD_OP2+100
MOV    WORD_OP1,WORD_OP1-WORD_OP2

```

6. What is the difference between the instructions

MOV AX, TABLE_ADDR

and

LEA AX, TABLE_ADDR

7. Draw a diagram similar to Fig. 3-8(b) to show the action that takes place while the sequence in Fig. 3-14 is being executed.
8. For each of the following instructions draw a figure similar to those in Fig. 3-7:

| | |
|----------------|----------------------|
| (a) MOV BX,CX | (b) XCHG OPR[SI],AX |
| (c) ADD DX,67H | (d) LEA CX,X[BX][DI] |
9. Write program sequences that will carry out the following binary operations:

| | |
|--------------------------------------|--|
| (a) $Z \leftarrow W + (Z - X)$ | (b) $Z \leftarrow W - (X + 6) - (Y + 9)$ |
| (c) $Z \leftarrow (W * X) / (Y + 6)$ | R ← Remainder |
| | (d) $Z \leftarrow ((W - X) / 10 * Y) ** 2$ |
10. Show that the carry is always properly set when carrying out multiple-precision addition on 2's complement numbers. Also show that it is properly set during a subtraction.
11. Use conditional branch instructions to modify the code in Fig. 3-20 so that it will multiply signed double-precision 2's complement numbers.
12. Write a program sequence for performing an unsigned binary division on an n -word number by a one-word number.
13. Use conditional branch instructions to modify your answer to Exercise 12 so that it can operate on signed numbers.
14. Write program sequences that will perform the following operations on two-digit packed BCD numbers:

| | |
|--------------------------------|--------------------------------------|
| (a) $U \leftarrow V + (S - 6)$ | (b) $U \leftarrow (X + W) - (Z - U)$ |
|--------------------------------|--------------------------------------|
15. Repeat Exercise 14 assuming four-digit packed BCD numbers.
16. Repeat Exercise 14 assuming one-digit unpacked BCD numbers.
17. Repeat Exercise 14 for two-digit unpacked BCD numbers. Also write the code for performing the operation (where Z is one digit)

$$U \leftarrow X * Y / Z \quad (\text{ignore remainder})$$

18. Extend the example in Fig. 3-26 to three-digit 10's complement numbers.
19. Mathematically show that an unpacked BCD product is properly adjusted by dividing the (AL) by 10 and putting the quotient in AH and the remainder in AL.
20. Assume that an unpacked BCD digit is being divided into a sequence of unpacked BCD digits. Mathematically show that for each successive digit the correct unpacked BCD quotient can be obtained by multiplying the (AH) (the remainder from the previous division) by 10 and adding it to the (AL) (the next digit) before performing the binary division.
21. Use looping to write a sequence to add two 16-digit 10's complement packed BCD numbers. Repeat for unpacked BCD numbers.
22. An algorithm for the unsigned double-precision division of $c2^{16} + d$ by $a2^{16} + b$ ($a > 0$) is: Compute q and r so that

$$c = qa + r$$

(q and r may be obtained by single-precision division). But $c = qa + r$ implies that

$$\begin{aligned} c2^{16} + d &= qa2^{16} + r2^{16} + d \\ &= q(a2^{16} + b) + r2^{16} + d - qb \end{aligned}$$

The quantity $r2^{16} + d - qb$ is considered to be the remainder for the original division; if it is negative, repeatedly add $a2^{16} + b$ and decrement q until it becomes positive. Implement this procedure using DIV.

23. Reconsider the algorithm in Exercise 22 and explain what should be done in the case for which a is zero.

24. Given that VAR1 and VAR2 are word variables and LAB is a label, find the errors in the following instructions:

- | | |
|-----------------|-------------------|
| (a) AAA DX | (b) ADD VAR1,VAR2 |
| (c) SUB AL,VAR1 | (d) JMP LAB[SI] |
| (e) JNZ VAR1 | (f) JMP NEAR LAB |

25. Give the machine code for each of the following instructions assuming that ADDR has an offset of 10B0 and the instruction has an offset of 10A0:

- | | |
|--------------------|-----------------------|
| (a) JNZ ADDR | (b) JB ADDR-2 |
| (c) JMP SHORT ADDR | (d) JMP NEAR PTR ADDR |

Repeat the exercise after assuming the instruction has an offset of 10C0.

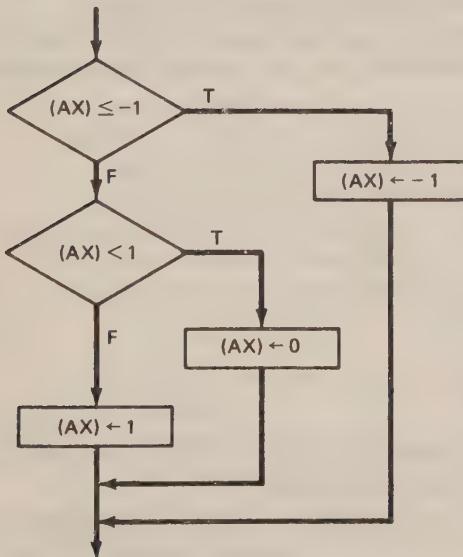
26. Suppose that BR_ADDR has an offset of 0900 in segment CODE2 and the segment address of CODE2 is 0100. Find the machine code for the following instruction:

JMP FAR BR_ADDR

27. Suppose that TAB has an offset of 1C00 and give the machine code for the following intrasegment indirect branches:

- | | | |
|-------------|-----------------|---------------------------|
| (a) JMP TAB | (b) JMP TAB[SI] | (c) JMP WORD PTR [BX][DI] |
|-------------|-----------------|---------------------------|

28. Write a program sequence that implements the following flowchart:



29. Write a program sequence that puts the letters whose ASCII codes are in CHAR, CHAR + 1, and CHAR + 2 in alphabetical order.

30. Write a program sequence that will move the information in LIST through LIST + 100 to BLK through BLK + 100.

31. Write a program sequence that will reverse the contents of the bytes ARRAY through ARRAY + N - 1, where N is the contents of NUM.
32. Modify the bubble sort example in Fig. 3-44 so that it checks for early completion, thus making it more efficient.
33. Use the algorithm suggested at the end of Sec. 3-3-1 to extend the answer to Exercise 22 to permit the dividend to be 8-precision.
34. An insertion sort for putting numbers in descending order proceeds by putting the first two elements in the array in order; then inserting the third element in its proper place by comparing it with the current second element and possibly the current first element; then inserting the fourth element in its proper place by successive comparisons; and so on. Write a program sequence which uses an insertion sort to order a word array LIST that contains N elements, where N is contained in LEN.
35. Write a program sequence that will convert the 16-digit packed BCD number beginning at PACKED to an unpacked BCD number and store the result beginning at UNPACKED. Modify the sequence so that the unpacked number is an ASCII string.
36. Write a program sequence that will test the byte STATUS and branch to ROUTINE_1 if bit 1, 2, or 4 is 1. Otherwise, it is to branch to ROUTINE_2 if both bits 1 and 3 are 1 and to ROUTINE_3 if both bits 1 and 3 are 0. In all other cases it is to execute ROUTINE_4. Assume that the routines are more than 128 bytes long. Also give a flowchart of the sequence.
37. Consider a string of characters stored in STRING through STRING + 99. Suppose that bit 5 of register DL is to be set to 1 if the string contains a digit; otherwise, this bit is to be set to 0. In any case only bit 5 is to be affected. Flowchart a solution to the problem and then implement the flowchart in assembler language.
38. For the following equivalent program sequences, assume the 8086 is driven by a 5-MHz clock and estimate their execution times. (See the Appendix for the numbers of clock cycles.)

(a)

| | | |
|-------|------|---------|
| | MOV | CX,5 |
| NEXT: | ADD | AX,[SI] |
| | INC | SI |
| | LOOP | NEXT |

(b)

| | | |
|-------|-----|---------|
| | MOV | BX,5 |
| NEXT: | ADD | AX,[SI] |
| | INC | SI |
| | DEC | BX |
| | JNE | NEXT |

39. Four of the op code bits in the conditional branch instructions determine the test made by the instruction. Which 4 bits are they? (See Fig. 3-71.)
40. Treat the concatenation of the contents of DX:AX as a 32-bit quantity and write a program sequence that puts the bit position of the least significant bit that contains a 1 in register BL.
41. Write a program sequence that will pack four 12-bit quantities in four consecutive words into three consecutive words.
42. The shift and rotate instructions are the same except for 3 bits. Which bits distinguish these instructions from each other? (See Fig. 3-71.)

43. Show the allocated space and initialized data caused by the following statements:
- BYTE_VAR DB 'BYTE',12,-12H,3 DUP(0,?,2 DUP(1,2),?)
 - WORD_VAR DW 5 DUP(0,1,2),?-5,'BY','TE',256H
44. Assume that DECIMAL_STRING and BINARY_WORD are defined in the segment DATA_SEG and write a complete code segment that will:
- Convert a positive BCD number, which has four unpacked BCD digits stored in DECIMAL_STRING, to a 16-bit binary number and store it in BINARY_WORD.
 - Convert a binary number stored in BINARY_WORD to its equivalent in unpacked BCD format and store the results in DECIMAL_STRING.
45. Define a structure INVEN_REC which begins with a word named PART (no preassignment), continues with 3 bytes named TYPP containing 05H, 0A2H, and 52H, and ends with a double word named SIZP (no preassignment). Also give a statement that invokes INVEN_REC and preassigns the PART field with 6257H, but leaves the other fields alone. Next define a data segment INVENTORY that begins with a 9-byte string named NAME which is preassigned the string 'INVENTORY', continues with an unpreassigned byte LEN, and ends with 100 duplications of INVEN_REC (no preassignment).
46. Assume that eight Boolean variables X_0, X_1, \dots, X_7 are to be stored in the variable INPUT with the following bit assignment:

| | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| X_7 | X_6 | X_5 | X_4 | X_3 | X_2 | X_1 | X_0 |

- Define a RECORD for the above bit assignment and reserve storage for INPUT.
- Write a program sequence to evaluate the Boolean expression

$$\bar{X}_7X_2X_1\bar{X}_0 + \bar{X}_3X_0 + \bar{X}_5\bar{X}_4X_3$$

and store the result (0 or 1) in DX.

- Write a program sequence that uses a branch table BR_TAB to pass control to ROUTINE_i if X_i is 1, $i = 0, 1, 2, 3, 4, 5, 6, 7$. The search for a 1-bit is to be from right to left and if none of the bits is 1, then no branch is to be taken.
47. Assume that the word variables X, Y, Z are defined in segments X_SEG, Y_SEG, and Z_SEG, respectively, and that DS, ES, and SS are reserved for X_SEG, Y_SEG, and Z_SEG. Write a program sequence to calculate

$$X \leftarrow X+Y+Z$$

which includes the necessary ASSUME statements and instructions for loading the segment registers.

48. Write a complete data segment DATA_SEG that would assign the integer 5 to a byte NUM and the integers -1, 0, 2, 5, and 4 to the first five elements of the 10-word array DATA_LIST. Then write a complete code segment that would:
- Place the largest and smallest of the first five numbers of DATA_LIST in BX and DX, respectively.
 - Calculate the sum and the product of the first five numbers in DATA_LIST and store the results in SUM and PRODUCT, respectively.
49. Write a complete program that adds AUGEND in segment D_SEG to ADDEND in segment E_SEG and puts the result in SUM in D_SEG. AUGEND, ADDEND, and SUM are to be double-precision binary numbers with AUGEND preassigned with 99251

and ADDEND preassigned with -15962. The code is to be in C_SEG and D_SEG, E_SEG, and C_SEG are to be associated with DS, ES, and CS, respectively.

50. Determine the contents of the symbol table and the machine code produced by the assembler for the following program:

```

PROG      SEGMENT
ASSUME    CS:PROG, DS:PROG
DATA_ARRAY DW     10 DUP(?)
SUM        DW     ?
START:    MOV    AX,CS
          MOV    DS,AX
          XOR    AX,AX
          MOV    BX,AX
          MOV    CX,10
NEXT:     ADD    AX,DATA_ARRAY[BX]
          ADD    BX,2
          LOOP   NEXT
          MOV    SUM,AX
          HLT
PROG      ENDS
END      START

```

51. Modify the code in Exercise 50 so that the data are in a separate segment NUMBERS and then repeat the exercise.
52. Assume that the offset of DATA_WORD is 123E and show all possible machine instructions equivalent to
- (a) MOV AX,DATA_WORD (b) MOV BX,34
 - (c) MOV DATA_WORD,34
53. Assume that the variables AMT and QUAN have offsets 0300 and 1A00, respectively, and give the machine code for each of the instructions:
- | | |
|--------------------------|-----------------------|
| (a) ADD AMT,-32H | (b) ADC DX,AMT+4 |
| (c) SUB AX,QUAN | (d) SBB QUAN[SI+2],AX |
| (e) CMP AX,QUAN | (f) DAA |
| (g) ADD AX,DX | (h) CMP DX,0 |
| (i) INC CX | (j) NEG QUAN[BX][DI] |
| (k) ADD DX,AMT[BX][SI+2] | (l) CWD |

4

Modular Programming

Chapter 3 concentrated on individual instructions and how to put these instructions together to perform simple tasks. Although the construction of complete programs was discussed, the programs were single-module programs. Clearly, most useful programs are much more complex and must include several tasks such as those that appear in the examples of Chap. 3. The present chapter considers the formulation of complex programs from numerous small sequences, called *program modules* (or simply *modules*) each of which performs a well-defined task. Such formulation of computer code is referred to as *modular programming*.

Assembler language programs are developed by essentially the same procedure as high-level language programs, that is, by:

1. Precisely stating what the program is to do.
2. Breaking the overall problem into tasks.
3. Defining exactly what each task must do and how it is to communicate with the other tasks.
4. Putting the tasks into assembler language modules and connecting the modules together to form the program.
5. Debugging and testing the program.
6. Documenting the program.

This chapter is concerned primarily with steps 2, 3, and 4. These steps are similar to those followed by a hardware designer, who first breaks the design into circuit modules, which may correspond to printed circuit (PC) boards, implements the modules, and then integrates them into the desired system.

The primary aid used in subdividing a program into modules is the *hierarchical diagram*, which is a block-oriented figure that summarizes the relationships between the modules (tasks) and submodules (subtasks). A typical hierarchical diagram is shown in Fig. 4-1. These diagrams look like, and serve the same purpose as, organizational charts for corporations. The main module corresponds to the president of the corporation, the principal submodules, A, B, and C, to the vice-presidents, and so on. A hierarchical diagram shows the chain of subordination that exists among the modules and their submodules just as an organizational chart indicates the chain of command within a corporation. It helps the programmer and others to visualize the functional structure of the program. Note that a submodule may be subordinate to more than one module just as a person can appear more than once on an organizational chart.

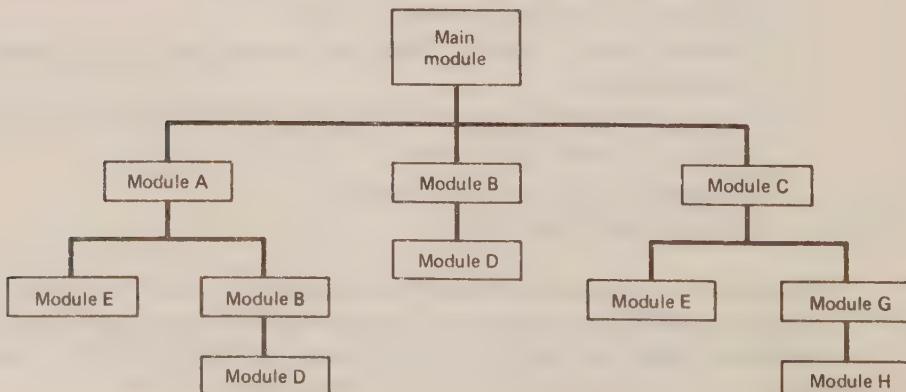
The reasons for breaking a program into small parts are that:

1. Modules are easier to comprehend.
2. Different modules can be assigned to different programmers.
3. Debugging and testing can be done in a more orderly fashion.
4. Documentation can be more easily understood.
5. Modifications may be localized.
6. Frequently used tasks can be programmed into modules that are stored in libraries and used by several programs.

With regard to item 1, the human mind is capable of concentrating on only a limited amount of material at one time. Therefore, it is necessary for us to subdivide our thinking no matter what kind of problem we are confronted with. Modular programming is a natural extension of this phenomenon to programming.

When designing the modules of a program one must be concerned with how and under what circumstances the modules are entered and exited, the *control coupling*, and how information is communicated between the modules, the *data*

Figure 4-1 Typical hierarchical diagram.



coupling. The coupling between two modules depends on several factors, including whether they are assembled together or separately and the organization of the data. In general, the task selection and module design should be such that control coupling is kept simple and data coupling is minimized.

Most assembler languages are designed to aid the modularization process in three ways. One is to allow data to be structured so that they can be readily accessed by the various modules, another is to provide for procedures (or subroutines), and the third is to permit sections of code, known as *macros*, to be inserted by the appearance of single statements, each of which contains a name and a set of arguments. The structuring of data was discussed in Sec. 3-10 and is considered further in the first two sections of this chapter. The first section of this chapter also discusses the way modules that are assembled separately are linked together and how programs are prepared for execution. Sections 4-2 and 4-3 examine stacks and procedures and how they are implemented on the 8086. The next section introduces interrupts and Sec. 4-5 considers macros. Sections 4-6 and 4-7 use the material presented in the first five sections to discuss the design of assembler language programs, especially programs written for the 8086.

4-1 LINKING AND RELOCATION

In constructing a program some program modules may be put in the same source module and assembled together; others may be in different source modules and assembled separately. If they are assembled separately, then the main module, which has the first instruction to be executed, must be terminated by an END statement with the entry point specified, and each of the other modules must be terminated by an END statement with no operand. In any event, the resulting object modules, some of which may be grouped into libraries, must be linked together to form a load module before the program can be executed. In addition to outputting the load module, normally the linker prints a *memory map* that indicates where the linked object modules will be loaded into memory. After the load module has been created it is loaded into the memory of the computer by the loader and execution begins. Although the I/O can be performed by modules within the program, normally the I/O is done by I/O drivers that are part of the operating system. All that appears in the user's program are references to the I/O drivers that cause the operating system to execute them.

The general process for creating and executing a program is illustrated in Fig. 4-2. The process for a particular system may not correspond exactly to the one diagrammed in the figure, but the general concepts are the same. The arrows indicate that corrections may be made after any one of the major stages. In the MCS-86 software the linker is divided into two parts, one being called the linker and the other the locator, and the loading is done by the operating system. Regardless of the system, however, the linker/loader combination must make all of the segment and address assignments needed to allow the program to work properly.

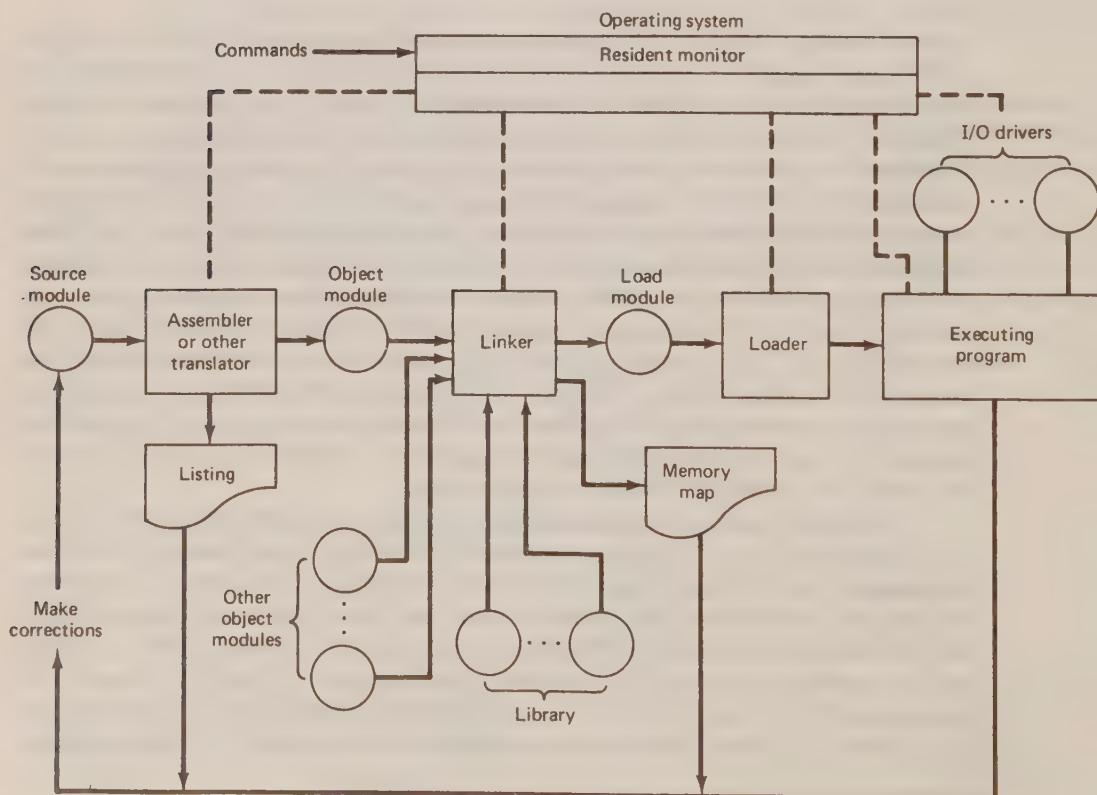
More specifically, this combination must:

1. Find the object modules to be linked.
2. Construct the load module by assigning the positions of all of the segments in all of the object modules being linked.
3. Fill in all offsets that could not be determined by the assembler.
4. Fill in all segment addresses.
5. Load the program for execution.

The object modules to be linked are determined by naming them in the command to the linker and by the operating system searching through libraries. The format of a linker command depends on the system, but it typically contains a command word followed by a list of object modules and libraries. Normally, one library, the system library, does not have to be listed because it is automatically searched for needed object modules that are not in the command list.

The order in which the object modules appear in the linker command may determine the order in which they are stacked together to form the load module,

Figure 4-2 Creation and execution of a program.



or the arrangement of the load module may be decided by the operating system according to a set of fixed rules. The former would provide the user with control over the construction of the load module, but ordinarily the user does not care about the exact ordering within the load module and would prefer to have the operating system take care of some of the details. The amount of control given to the user and the way in which it is given is system dependent. Intel's MCS-86 software allows the user to specify the module and segment ordering in detail.

4-1-1 Segment Combination

In addition to the linker commands, the ASM-86 assembler provides a means of regulating the way segments in different object modules are organized by the linker. Sometimes segments with the same name are concatenated and sometimes they are overlaid. Just how the segments with the same name are joined together is determined by modifiers attached to the SEGMENT directives. A SEGMENT directive may have the form

| | | |
|--------------|---------|--------------|
| Segment name | SEGMENT | Combine-type |
|--------------|---------|--------------|

where the combine-type indicates how the segment is to be located within the load module. Segments that have different names cannot be combined and segments with the same name but no combine-type will cause a linker error. The possible combine-types are:

PUBLIC—If segments in different object modules have the same name and the combine-type PUBLIC, then they are concatenated into a single segment in the load module. The ordering in the concatenation is specified by the linker command.

COMMON—If the segments in different object modules have the same name and the combine-type is COMMON, then they are overlaid so that they have the same beginning address. The length of the common segment is that of the longest segment being overlaid.

STACK—If segments in different object modules have the same name and the combine-type STACK, then they become one segment whose length is the sum of the lengths of the individually specified segments. In effect, they are combined to form one large stack.

AT—The AT combine-type is followed by an expression that evaluates to a constant which is to be the segment address. It allows the user to specify the exact location of the segment in memory.

MEMORY—This combine-type causes the segment to be placed at the last of the load module. If more than one segment with the MEMORY combine-type is being linked, only the first one will be treated as having the MEMORY combine-type; the others will be overlaid as if they had COMMON combine-types.

(SEGMENT directives can also have alignment modifiers for specifying the alignment of the beginning address of a segment, but the default alignment, which causes a segment to begin at an address that is divisible by 16, is the only one considered in this book. For information about the other segment alignments, one should refer to the appropriate manuals.)

Figure 4-3 shows how the PUBLIC and COMMON combine-types affect the construction of a load module. By causing two or more code segments to be put in a single segment, the use of PUBLIC eliminates the need to change the contents of the CS register as the program passes between sets of instructions within the code segment, i.e., it allows intersegment branches to be replaced by intrasegment branches. Data segments can be given the PUBLIC combine-type to cause several

Figure 4-3 Segment combinations resulting from the PUBLIC and COMMON combine types.

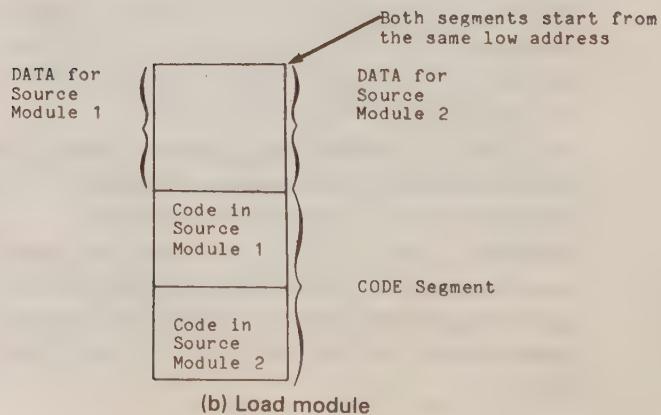
```

Source Module 1
DATA      SEGMENT      COMMON
.
.
.
DATA      ENDS
CODE      SEGMENT      PUBLIC
.
.
.
CODE      ENDS

Source Module 2
DATA      SEGMENT      COMMON
.
.
.
DATA      ENDS
CODE      SEGMENT      PUBLIC
.
.
.
CODE      ENDS

```

(a) Code



(b) Load module

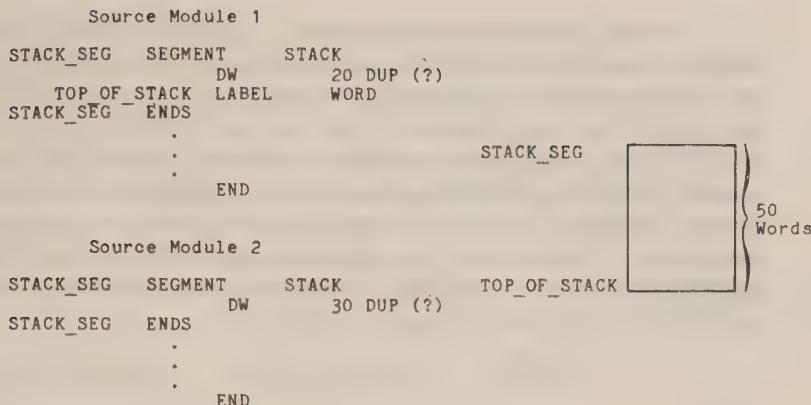


Figure 4-4 Formation of a stack from two segments.

sets of data to be combined into one larger set. The COMMON combine-type makes it possible to create a pool of locations that can be shared by several modules and is considered further in Sec. 4-3-3.

STACK provides a means of obtaining one stack that can be shared by several modules while allowing each module to contribute to the overall stack length. An example of the use of the STACK modifier is provided in Fig. 4-4. Stacks are also discussed in detail in Sec. 4-2.

The AT combine-type permits the user to specify the exact beginning location of a segment of code or data. It is used primarily by systems programmers or someone who is writing the software for a special purpose system. The final physical addresses of the variables and labels defined in segments not given the AT combine-type are set by the linker/loader process and these variables and labels are said to be *relocatable*. On the other hand, the addresses of the variables and labels defined in segments having the AT combine-type are set by the programmer and assembler and are referred to as *absolute* variables and labels. AT is most often applied in connection with I/O handling, which involves special routines and data buffers and other memory areas that must be directly accessible by a variety of programs.

Sometimes a programmer needs to force one segment to be located in a higher part of memory than the other segments. The MEMORY combine-type can be applied to such situations.

4-1-2 Access to External Identifiers

Clearly, object modules that are being linked together must be able to refer to each other. That is, there must be a way for a module to reference at least some of the variables and/or labels in the other modules. If an identifier is defined in an object module, then it is said to be a *local* (or *internal*) *identifier* relative to the module, and if it is not defined in the module but is defined in one of the other modules being linked, then it is referred to as an *external* (or *global*) *identifier* relative to the module.

For single-object module programs all identifiers that are referenced must be locally defined or an assembler error will occur. For multiple-module programs, the assembler must be informed in advance of any externally defined identifiers that appear in a module so that it will not treat them as being undefined. Also, in order to permit other object modules to reference some of the identifiers in a given module, the given module must include a list of the identifiers to which it will allow access. Therefore, each module may contain two lists, one containing the external identifiers it references and one containing the locally defined identifiers that can be referred to by other modules. These two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

EXTRN Identifier:Type, . . . , Identifier:Type

and

PUBLIC Identifier, . . . , Identifier

where the identifiers are the variables and labels being declared as external or as being available to other modules. Because the assembler must know the type of all external identifiers before it can generate the proper machine code, a type specifier must be associated with each identifier in an EXTRN statement. For a variable the type may be BYTE, WORD, or DWORD and for a label it may be NEAR or FAR. In the statement

INC VAR1

if VAR1 is external and is associated with a word, then the module containing the statement must also contain a directive such as

EXTRN . . . ,VAR1:WORD, . . .

and the module in which VAR1 is defined must contain a statement of the form

PUBLIC . . . ,VAR1, . . .

One of the primary tasks of the linker is to verify that every identifier appearing in an EXTRN statement is matched by one in a PUBLIC statement. If this is not the case, then there will be an undefined external reference and a linker error will occur. Figure 4-5 shows three modules and how the matching is done by the linker while joining them together. Relative to Module 1, VAR1 and LAB1 are local and declared public and VAR3 and VAR4 are local but not declared public. Therefore, VAR3 and VAR4 are ignored by the linker. Module 1 refers to VAR2 and LAB2, which are not locally defined but are available to Module 1 because they have been declared PUBLIC in at least one of the other modules. Module 2 contains the definition of VAR2 and Module 3 defines LAB2. Module 2 references VAR1 and VAR4, which are defined in Module 1, but because VAR4 does not appear in a PUBLIC statement a linker error will occur. There is no ambiguity in defining VAR3 in Module 2 as well as Module 1 because there is no cross-reference involving VAR3. LAB3, which is defined in Module 3, is made PUBLIC but does not require a match; it is simply not used by the other modules to which Module 3 is currently being linked.

```

Source Module 1 . . .
EXTRN VAR2:WORD, LAB2:FAR
PUBLIC VAR1, LAB1

DATA1      SEGMENT
VAR1       DB      2
VAR3       DW      ?
VAR4       DW      ?
DATA1      ENDS
:
:
Source Module 2 . . .
EXTRN VAR1:BYTE, VAR4:WORD
PUBLIC VAR2

DATA2      SEGMENT
VAR2       DW      0
VAR3       DB      5 DUP (?)
DATA2      ENDS
:
:
A linker error will
occur because there
is no matching
PUBLIC declaration

Source Module 3 . . .
EXTRN LAB1:FAR
PUBLIC LAB2, LAB3
:
:
Does not need a match

```

Figure 4-5 Illustration of the matching verified by the linker.

As we have seen, there are two parts to every address, an offset and a segment address. The offsets for the local identifiers can be and are inserted by the assembler, but the offsets for the external identifiers and all segment addresses must be inserted by the linking process. The offsets associated with all external references can be assigned once all of the object modules have been found and their external symbol tables have been examined. The assignment of the segment addresses is called *relocation* and is done after the linking process has determined exactly where each segment is to be put in memory. (In the MCS-86 system, relocation is done by the locator, LOC-86.)

Let us consider the program in Fig. 4-6. The first word in the DD directive and the offsets of VAR1 in the DEC instruction and POINTER in the LES instruction are inserted by the assembler. Because DATA_SEG is a segment name, the addressing mode of the MOV instruction is immediate with the operand being the segment address of DATA_SEG. This address and the second word of the DD directive must be inserted by the linker. Also, the linker must fill in both the offset and the segment address for the external reference in the JMP instruction.

Although informing the assembler which variables are external is necessary, this alone is not sufficient. Just as with local variables, at the time the program is executing the segment registers must be dynamically filled with the proper segment addresses. If VAR is an external variable, then before it can be used in an instruction the segment address of VAR must be put in the appropriate segment register. This can be done as follows:

| | | |
|-----|------------|--------------------------------|
| MOV | AX,SEG VAR | ;PUT THE SEGMENT ADDR |
| MOV | ES,AX | ;OF VAR IN ES |
| . | | |
| ADD | DX,ES:VAR | ;INSTRUCTION THAT ACCESSES VAR |

```

    EXTRN   LAB:FAR
DATA SEG  SEGMENT
    POINTER  DD      VAR2?
    VAR1    DW      ?
DATA SEG  ENDS
EXTRA   SEGMENT
    VAR2    DW      ?
EXTRA   ENDS
CODE SEG  SEGMENT
    ASSUME CS:CODE SEG, DS:DATA SEG, ES:EXTRA
        MOV     AX, DATA_SEG
        MOV     DS, AX
        .
        .
        DEC     VAR1
        LES     DI, POINTER
        ADD     BX, ES:[DI]
        .
        .
JMP     FAR PTR LAB
        .
        .
CODE SEG  ENDS

```

Figure 4-6 Program for discussing insertion of addresses.

If an ASSUME statement that associates the segment address of VAR with ES appears before this code sequence, then the "ES:" prefix in the ADD instruction could be deleted.

As a more extensive example consider the three source modules given in Fig. 4-7. The first two MOV instructions load DS with the segment address LOCAL_DATA. This would need to be done even in a single-module program. The next two MOV instructions put the address of the segment containing VAR1 into ES so that the following ADD instruction will correctly address VAR1. The third pair of MOV instructions changes (ES), thus allowing VAR2 to be properly addressed by the SUB instruction. No extra instructions are needed to set CS before the JMP instruction is executed because the new contents of CS are part of any intersegment branch instruction. The segment address of the segment ROUTINE which contains the label OUTPUT is put into the last word of the JMP instruction by the linker and CS is loaded when the JMP instruction is executed. Note that the external and public declarations in a source module can be divided among multiple EXTRN statements and multiple PUBLIC statements.

If several variables in an external segment are to be accessed by a module and the name of the external segment is known, there is an alternate method for handling the linkage that is compatible with setting up the segment addresses for local variables using an ASSUME statement. This method consists of defining a segment with the same name as the external segment and filling it with EXTRN statements that include the names of the external variables. This segment should consist of only EXTRN statements and the EXTRN statements should include only the names of the referenced variables in the external segment. Also, its SEGMENT statement must be given the combine-type PUBLIC. Once such a segment has been defined, then the segment name and the variables listed in its EXTRN statements can be treated just like a local segment name and local variables. The procedure is demonstrated by the two modules given in Fig. 4-8. Note

```

Source Module 1                               Source Module 2

EXTRN  VAR1:WORD, OUTPUT:FAR
EXTRN  VAR2:WORD
LOCAL_DATA SEGMENT
.
.
.
LOCAL_DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:LOCAL DATA
START:    MOV     AX, LOCAL_DATA
          MOV     DS, AX           ;SET UP DS TO ACCESS LOCAL DATA
.
.
.
MOV     AX, SEG VAR1      ;SET UP ES
MOV     ES, AX             ;TO ACCESS VAR1
ADD    BX, ES:VAR1        ;USE VAR1 IN PROCESSING
.
.
.
MOV     AX, SEG VAR2      ;SET UP ES
MOV     ES, AX             ;TO ACCESS VAR2
SUB    ES:VAR2, 50         ;USE VAR2 IN PROCESSING
.
.
.
JMP     OUTPUT            ;LAST WORD OF INSTRUCTION
                           ;CONTAINS SEGMENT ADDR OF ROUTINE
.
.
.
CODE ENDS
END START

Source Module 3

PUBLIC  VAR2
EXTDATA2 SEGMENT
.
.
.
VAR2:      DW      ?
.
.
.
EXTDATA2 ENDS
END
PUBLIC  OUTPUT
ROUTINE SEGMENT
ASSUME CS:ROUTINE
.
.
.
OUTPUT   LABEL  FAR
.
.
.
ROUTINE ENDS
END

```

Figure 4-7 Example showing how segment registers are loaded from external references.

that by including ES:GLOBAL in the ASSUME statement, the "ES:" prefix is not needed in the third and fourth MOV instructions.

4-2 STACKS

There are many situations in which a program needs to temporarily store information and then retrieve it in reverse order. One example of such a situation is saving and restoring the counters when nesting loops. On the 8086 the CX register and the loop instructions can be conveniently used to provide the counting, testing, and branching needed in a loop, but because the loop instructions are designed to

```

Source Module 1

GLOBAL      SEGMENT      PUBLIC
    EXTRN  VAR1:WORD, VAR2:WORD
GLOBAL      ENDS
LOCAL_DATA  SEGMENT

.
.

LOCAL_DATA  ENDS
CODE        SEGMENT
ASSUME CS:CODE, DS:LOCAL_DATA, ES:GLOBAL

.
.

MOV        AX,GLOBAL      ;USE ES TO
MOV        ES,AX           ;ACCESS SEGMENT GLOBAL
MOV        BX,VAR1
MOV        VAR2,BX

.
.

CODE        ENDS
END

Source Module 2

GLOBAL      SEGMENT      PUBLIC
    PUBLIC  VAR1, VAR2
    VAR1    DW      ?
    VAR2    DW      ?

.
.

GLOBAL      ENDS
.
.

END

```

Figure 4-8 Accessing multiple external symbols defined in the same segment.

use only the CX register a problem arises when loops are nested. However, if as shown in Fig. 4-9 there were an efficient means of saving the loop counters in order and then restoring them by retrieving the last stored counter first, at least part of the problem would be alleviated.

Most often a data storage situation such as the one mentioned above is resolved by designing into the computer *last-in/first-out (LIFO) stack* (or *push down stack*) facilities, in which the stack itself is simply a part of memory. Stacks are also frequently used for other storage manipulations, but the most important applications of stacks are related to procedures, which are discussed in the next

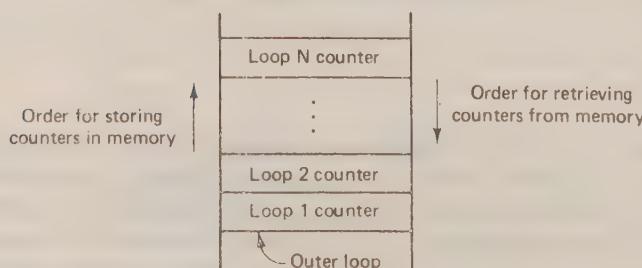


Figure 4-9 Storing and retrieving counters for nested loops.

| Name | Mnemonics and Format | | Description |
|---------------------|----------------------|-----|--|
| Push onto the stack | PUSH | SRC | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (SRC)$ |
| Pop from the stack | POP | DST | $(DST) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ |
| Push the flags | PUSHF | | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (PSW)$ |
| Pop the flags | POPF | | $(PSW) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ |

Flags: The flags are affected only by the POPF instruction.

Addressing modes: For the PUSH and POP instructions the operand must be a word and may not be immediate. A segment register can be specified as the operand in a PUSH or POP instruction. However, CS cannot be used in a POP instruction.

Figure 4-10 Stack instructions.

section. Stack facilities normally involve the use of indirect addressing through a special register, the stack pointer, that is automatically decremented as items are put on the stack and incremented as they are retrieved. Putting something on the stack is called a *push* and taking it off is called a *pop*. The address of the last element pushed onto the stack is known as the *top of the stack (TOS)*.

The 8086 instructions for directly pushing and popping the stack are given in Fig. 4-10. Only words can be pushed or popped and the data cannot be immediate, but the PUSH and POP instructions can use all of the other addressing modes. Only the POPF instruction, which pops the top of the stack into the PSW, affects the flags. Note that the PUSH and POP instructions are the only two instructions other than the MOV instruction that may explicitly specify a segment register as an operand. However, for the POP instruction, CS is not a valid operand. As illustrated in Fig. 4-11, the stack pointer register points to the top of the stack and a word is pushed onto the stack by decrementing (SP) by 2, so that it points to the next lower word in memory, and then storing the source operand into the address indicated by (SP). A pop consists of loading the top of the stack into the destination operand and then incrementing (SP) by 2. Thus, if the stack were pushed and popped the same number of times, the top of the stack would return to its original position. Figure 4-12 shows how nested loops could be implemented using the 8086 instructions.

On the 8086, the physical stack address is obtained from both (SP) and (SS) or (BP) and (SS), with SP being the implied stack pointer register for all push and pop operations and SS being the stack segment register. The (SS) are the lowest address in (i.e., limit of) the stack area and may be referred to as the *base of the stack*. The original contents of the SP are considered to be the largest offset the stack should attain. Therefore, the stack is considered to occupy the memory locations from 16 times (SS) to 16 times (SS) plus the original (SP). The location

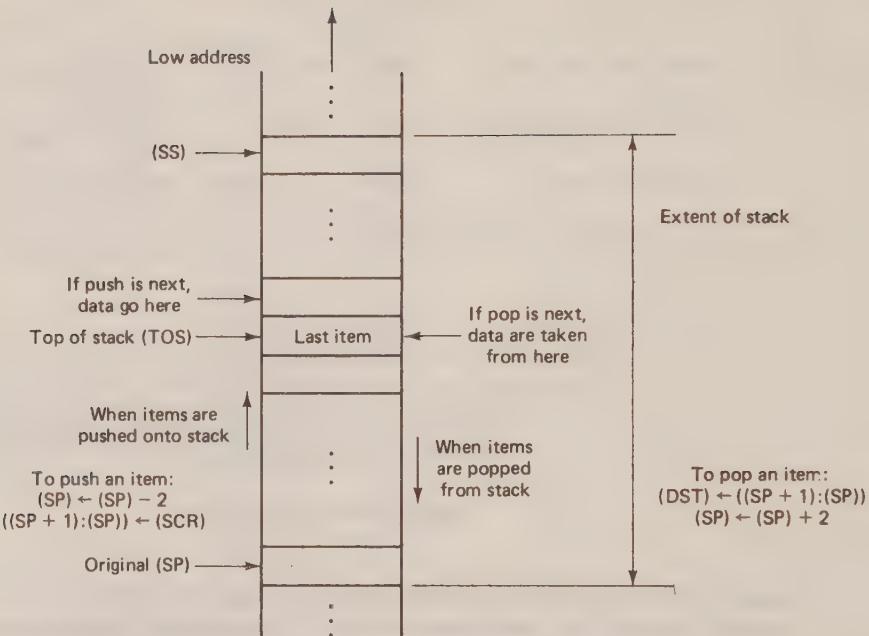
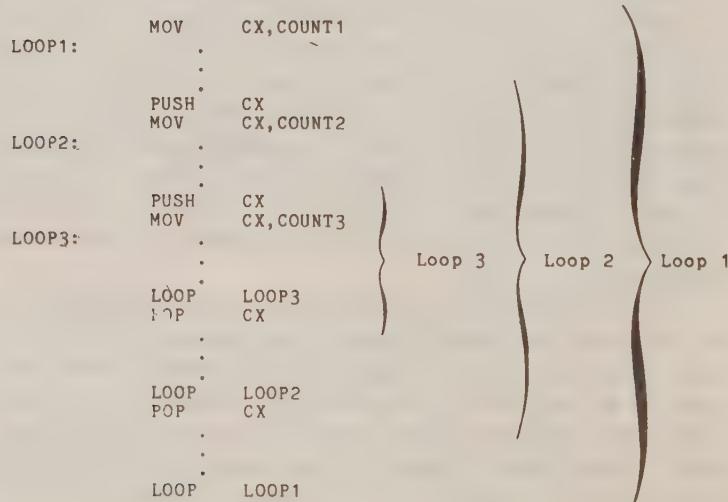


Figure 4-11 8086 stack.

of the stack must, of course, be initially set by software, which may be a code sequence in either the operating system or the user's program. This is done by loading the (SS) and (SP) as shown in Fig. 4-13. The BP register is primarily for

Figure 4-12 Using the stack for nesting loops.



```

STACK_SEG SEGMENT
    DW      30 DUP(?)
    LABEL  WORD
TOS      ENDS
STACK_SEG SEGMENT
ASSUME CS:CODE SEG, SS:STACK SEG
START:   MOV     AX,STACK_SEG
         MOV     SS,AX      ;INITIALIZES SS
         MOV     SP,OFFSET TOS ;INITIALIZES SP
         .
         .
CODE_SEG ENDS

```

Figure 4-13 Setting up a stack by loading the SS and SP registers.

allowing random accesses to the stack. For example,

MOV AX,[BP][SI]

will load AX from a location in the stack segment, the offset of the location being the sum of (BP) and (SI). On the other hand, for the instruction

MOV AX,[BX][SI]

the source operand is from the data segment.

Stack facilities are more efficient than ordinary memory in two ways. The push and pop instructions are shorter because one of the operands is indirectly addressed through the SP register, and the (SP) are automatically incremented or decremented to point to a new address. For example, the registers CX and DX could be stored in memory beginning at SAVE by the following instructions, which occupy a total of 8 bytes:

MOV SAVE,CX
MOV SAVE+2,DX

If the offset of SAVE is in SI, they could be saved by the sequence

MOV [SI],CX
INC SI
INC SI
MOV [SI],DX

which occupies only 6 bytes but contains four instructions. By saving CX and DX on the stack, only the instructions

PUSH CX
PUSH DX

which require only 2 bytes of memory and can be executed very quickly, are needed. The contents of CX and DX could be retrieved by the sequence

POP DX
POP CX

4-3 PROCEDURES

A *procedure* (or *subroutine*) is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which

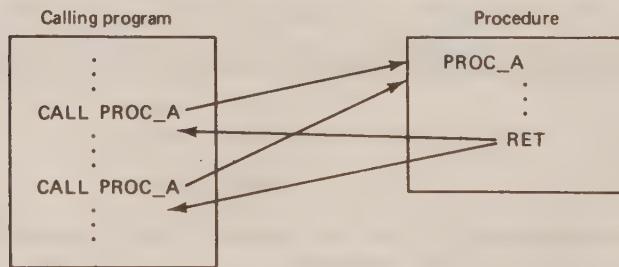
it is branched to. The branch to a procedure is referred to as the *call*, and the corresponding branch back is known as the *return*. The return is always made to the instruction immediately following the call regardless of where the call is located. If, as shown in Fig. 4-14(a), several calls are made to the same procedure, the return after each call is made to the instruction following that call. Therefore, only one copy of the procedure needs to be stored in memory even though the procedure may be called several times. When procedures are nested as shown in Fig. 4-14(b), each return is made to the corresponding calling procedure, not to a procedure higher in the hierarchy.

Procedures provide the primary means of breaking the code in a program into modules. Although not all modules are procedures, most of them are because procedures can easily be individually designed, tested, and documented. They can also be stored in libraries and used by a variety of programs. In short, they offer all of the conveniences accrued to modular programming and do so in a flexible manner. Procedures have one major disadvantage in that extra code is needed to join them together in such a way that they can communicate with each other. This extra code is referred to as *linkage* and is one of the main subjects of this section.

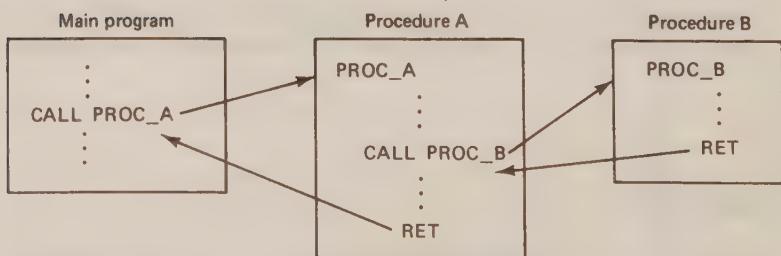
The following three requirements must be satisfied when calling a procedure:

1. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.

Figure 4-14 Use of procedures.



(a) Multiple calls



(b) Nested procedures

2. The registers used by the procedure need to be stored before their contents are changed, and then restored just before the procedure is exited.
3. A procedure must have a means of communicating or sharing data with the routine that calls it and other procedures.

4-3-1 Calls, Returns, and Procedure Definitions

The first of the above requirements is met by having special call and return branch instructions. These instructions for the 8086 are given in Fig. 4-15. The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from

Figure 4-15 Call and return instructions.

| Name | Mnemonic and Format | Description |
|---|---------------------|---|
| Intrasegment direct call | CALL DST | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (IP)+D16^*$ |
| Intrasegment indirect call | CALL DST | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (EA)$ |
| Intersegment direct call | CALL DST | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (CS)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow D16$ $(CS) \leftarrow \text{Segment address}$ (Last word of instruction) |
| Intersegment indirect call | CALL DST | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (CS)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (EA)$ $(CS) \leftarrow (EA+2)$ |
| Intrasegment return | RET EXP** | $(IP) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ |
| Intrasegment return with immediate data | RET EXP** | Same as above except, also $(SP) \leftarrow (SP)+D16$ |
| Intersegment return | RET | $(IP) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ $(CS) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ |
| Intersegment return with immediate data | RET EXP** | Same as above except, also $(SP) \leftarrow (SP)+D16$ |

*Displacement between the destination and the instruction following the CALL instruction.

**EXP is an expression that evaluates to a constant and becomes the D16 portion of the instruction.

Flags: No flags are affected.

Addressing modes: May be any branch addressing mode except a short CALL.

the stack. This assumes that, if the stack is accessed by the procedure, then the stack pointer is returned to its proper value and that the stack activity is not such that the return address is destroyed. Obviously, when working with procedures a programmer must be very careful as to how he manipulates the stack; otherwise, returns may be made to nonsensical locations.

The addressing modes for the CALL instruction are the same as those for the JMP instruction except that a short CALL is not available. A CALL may be direct or indirect and intrasegment or intersegment. If a call statement is a forward reference, an attribute operator NEAR PTR or FAR PTR may be added before the operand to indicate an intrasegment direct call or an intersegment direct call, respectively. However, if the CALL is intrasegment (intersegment), then the return must be intrasegment (intersegment). This is because an intrasegment call pushes only (IP), the offset of the return address, onto the stack, but an intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop one or two words from the stack. The RET instruction may optionally include a 16-bit immediate operand. If it does, then this operand is added to (SP) after the return address has been popped. This allows the top of the stack to be re-assigned, an option whose application will be discussed later.

Figure 4-16 shows the stack action that would occur when the following sequence of calls and returns is executed:

1. Main program calls FAR procedure PRO_A.
2. PRO_A calls NEAR procedure PRO_B.
3. PRO_B calls NEAR procedure PRO_C.
4. Return is made to PRO_B.
5. Return is made to PRO_A.
6. PRO_A calls NEAR procedure PRO_C.
7. Return is made to PRO_A.
8. Return is made to main program.

In Fig. 4.16 it has been assumed that the only stack action is due to the calls and returns and the return addresses are:

Main program return address for

| | | |
|-------------------------------------|-------------|-------------|
| PRO_A call | (CS) = 0500 | (IP) = 1000 |
| PRO_A return address for PRO_B call | | (IP) = 2500 |
| PRO_B return address for PRO_C call | | (IP) = 4000 |
| PRO_A return address for PRO_C call | | (IP) = 3700 |

Note that when the stack is popped the information is not destroyed, but it is erased by subsequent pushes, as is the case when PRO_A calls PRO_C.

Procedures are delimited within the source code by placing a statement of the form

| Procedure name | PROC | Attribute |
|----------------|------|-----------|
|----------------|------|-----------|

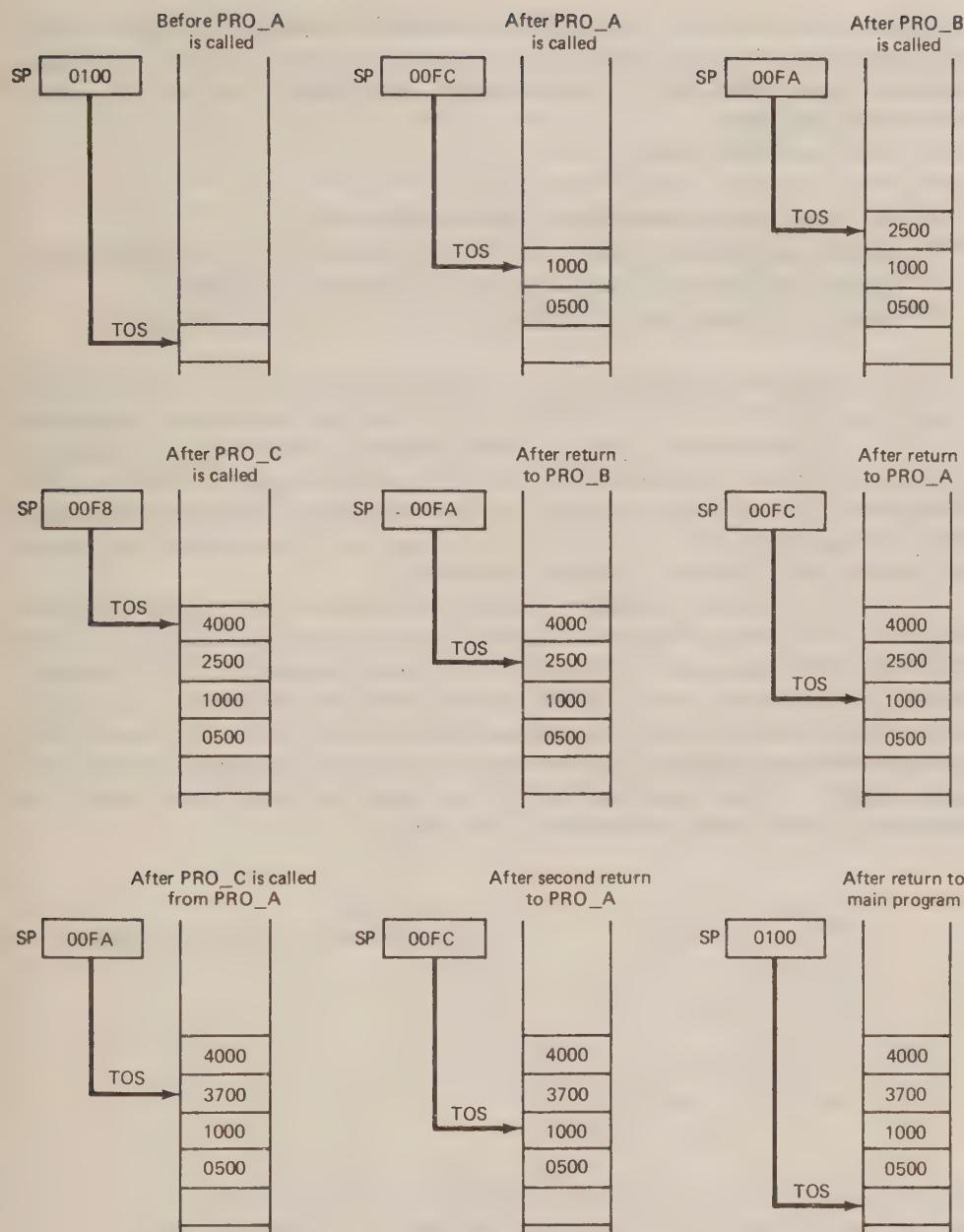


Figure 4-16 Stack action during a typical sequence of procedure calls and returns.

at the beginning of the procedure and by terminating the procedure with a statement

Procedure name ENDP

The procedure name is the identifier used for calling the procedure and the attribute

is either NEAR or FAR. The attribute is needed to determine the type of RET statement that is to be assembled. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case the attribute could be NEAR provided that *all* calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If a procedure is given a FAR attribute, then *all* calls to it must be intersegment calls even if the call is from the same code segment. Otherwise, the return would pop two words from the stack even though only one word was pushed onto the stack by the call. In addition, for the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

Figure 4-17 shows a situation in which procedure SUBT is defined in code segment SEGX and is called from both code segments SEGX and SEGY. Because SUBT is called from both SEGX and SEGY it must be given the FAR attribute. Since SUBT has been given the FAR attribute, the calls from within SEGX must push both (IP) and (CS) onto the stack as well as the call from SEGY. Since SUBT is defined before it is called, the assembler could actually implicitly translate the calls properly without using the FAR PTR attribute operator, but for clarity it is suggested that the attribute operator be included.

```

SEGX      SEGMENT
.
.
.
SUBT      PROC    FAR
.
.
.
RET
SUBT      ENDP
.
.
.
CALL    FAR PTR SUBT
.
.
.
SEGX      ENDS
SEGY      SEGMENT
.
.
.
CALL    FAR PTR SUBT
.
.
.
SEGY      ENDS

```

Figure 4-17 Procedure declaration.

4-3-2 Saving and Restoring Registers

Since both the calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and to restore them before returning to the calling program. Various means could be used for saving and retrieving the contents of the registers that are changed by a procedure, but if there are stack facilities (as with the 8086) the stack is almost always used for this purpose. Figure 4-18(a) gives the code needed to store and restore the contents of the AX, BX, CX, and DX registers and Fig. 4-18(b) shows how the stack and stack pointer are affected. Note that the register contents are popped in reverse order.

Figure 4-18 Storing and restoring the registers.

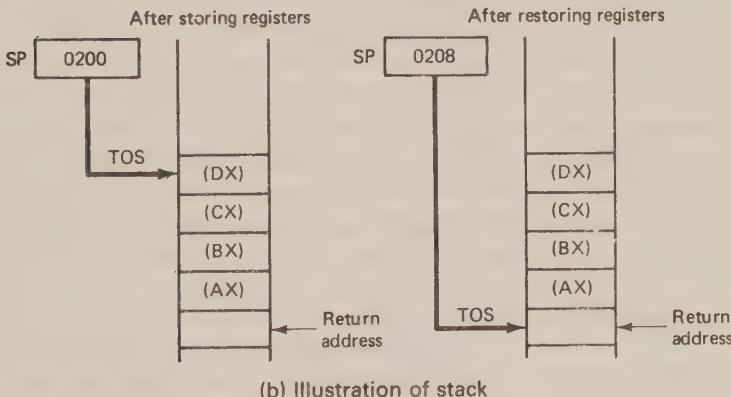
SUBT

```
PROC    NEAR
    PUSH   AX
    PUSH   BX
    PUSH   CX
    PUSH   DX
    .
    .
    POP    DX
    POP    CX
    POP    BX
    POP    AX
    RET
```

SUBT

ENDP

(a) Code



(b) Illustration of stack

The code in Fig. 4-18 does the storing and restoring from within the procedure. The storing could be done just before the procedure is called and the restoring just after the return, but this is not the method normally employed. By including these tasks in the procedure the code required to perform them needs to be written only once. If this code were part of the calling program, it would have to be repeated for each call.

Not all registers need to be saved, only those that are altered by the procedure. However, unless the procedure is short and not likely to be changed, it is normally

easier and less error prone to automatically save at least all of the registers in the data group, and possibly the pointer, index, and segment registers. If, say, the DX register is not saved because it is not accessed, but later the procedure is modified so that DX is used, then PUSH DX and POP DX would have to be properly inserted in the procedure. Otherwise, upon the return to the calling program (DX) may not be correct. An obvious exception would be the case in which DX is intentionally used to return a result. In this case, one would not want (DX) to be changed by a POP.

4-3-3 Procedure Communication

There are two general types of procedures, those that always operate on the same set of data and those that may process a different set of data each time they are called. For example, a procedure may be written so that it always adds the first COUNT elements in an array ARY and returns the result in a variable SUM, or it may be written so that it can add the elements in an arbitrary array and return the results to an arbitrary variable.

If a procedure is of the former type and is in the same source module as the calling program, then the procedure can refer to the variables directly as indicated in the program in Fig. 4-19. When the procedure is in a separate source module

Figure 4-19 Procedure that refers directly to the variables on which it operates.

```

DATA      SEGMENT
ARY      DW      100 DUP(?)
COUNT    DW      ?
SUM      DW      ?

.
.

DATA      ENDS
CODE      SEGMENT
.

.

CALL      NEAR PTR PROADD
.

.

PROADD   PROC      NEAR
    PUSH    AX      ;SAVE THE REGISTERS
    PUSH    CX
    PUSH    SI
    LEA     SI, ARY ;PUT ADDR OF ARY IN SI
    MOV     CX, COUNT ;PUT COUNT IN CX
    XOR     AX, AX ;CLEAR AX
NEXT:    ADD     AX, [SI] ;ADD THE FIRST
    ADD     SI, 2 ;COUNT NUMBERS
    LOOP   NEXT ;IN ARY
    MOV     SUM, AX ;STORE THE RESULT IN SUM
    POP     SI
    POP     CX
    POP     AX
    RET      ;RETURN
PROADD   ENDP
.

.

CODE      ENDS

```

it can still refer to the variables directly, provided that the calling program contains the directive

```
PUBLIC ARY, COUNT, SUM
```

and the procedure source module contains the directive

```
EXTRN ARY:WORD, COUNT:WORD, SUM:WORD
```

Alternatively, ARY, COUNT, and SUM could be put in a common area by placing the code

| | | |
|-------|---------|------------|
| DATA | SEGMENT | COMMON |
| ARY | DW | 100 DUP(?) |
| COUNT | DW | ? |
| SUM | DW | ? |
| DATA | ENDS | |

at the beginning of both source modules. This would cause both source modules to share the segment DATA. If the calling program uses different variable names, say NUM for ARY, N for COUNT, and TOTAL for SUM, then the DATA segment in its source module would need to be defined as follows:

| | | |
|-------|---------|------------|
| DATA | SEGMENT | COMMON |
| NUM | DW | 100 DUP(?) |
| N | DW | ? |
| TOTAL | DW | ? |
| DATA | ENDS | |

In either case the segments DATA would be overlaid by the linking process and the first 100 words of DATA would contain the numbers being summed, the next word would contain the count, and the last word would hold the result.

Whether the variables are referred to directly or a common area is employed, both of the above approaches to communicating with the procedure PROADD are restricted to summing the numbers in the same area of memory whenever the procedure is called. The only way the numbers in a different array WEIGHTS could be added would be to transfer them to ARY (or NUM) and the count to COUNT (or N) before calling the procedure, and then transfer the result from SUM (or TOTAL) to its desired location after the procedure has been executed. This could require a considerable amount of code and execution time since there could be as many as 102 numbers to be moved.

The solution is to pass the addresses of the array, the count, and the result to the procedure, thus enabling the procedure to work directly with different memory locations each time it is called. Even though 102 words may be involved, only three addresses would need to be passed since the array elements are in sequence. The variables whose addresses are passed are called *parameters*. There are two principal ways of passing parameter addresses: (1) to construct a parameter address table (or array) and pass the address of the table via a register, and (2) to push the parameter addresses onto the stack.

Figure 4-20(a) shows how the parameter address table could be constructed for the procedure PROADD and Fig. 4-20(b) gives the modified version of PROADD

```

PROG SEG  SEGMENT
    ARY      DW      100 DUP(?)      ;RESERVE 100 WORDS FOR ARY
    COUNT   DW      ?                  ;AND 1 WORD FOR COUNT
    SUM     DW      ?                  ;AND 1 WORD FOR RESULT
    TABLE   DW      3 DUP(?)       ;RESERVE 3 WORDS FOR
                                ;PARAMETER ADDRESSES
                                DW      100 DUP(?)      ;RESERVE SPACE
    TOS     LABEL WORD      ?          ;FOR STACK
ASSUME CS:PROG SEG, DS:PROG SEG, SS:PROG SEG
START:  MOV AX,CS
        MOV DS,AX      ;INITIALIZE DS
        MOV SS,AX      ;INITIALIZE SS
        MOV SP,OFFSET TOS ;INITIALIZE SP
        .
        .
        .
        MOV TABLE,OFFSET ARY      ;PUT THE ADDRESSES
        MOV TABLE+2,OFFSET COUNT;OF ARY, COUNT, AND SUM
        MOV TABLE+4,OFFSET SUM   ;IN PARAMETER TABLE
        MOV BX,OFFSET TABLE     ;PUT ADDR OF TABLE IN BX
        FAR PTR PROADD         ;CALL PROADD

        .
        .

PROG SEG  ENDS

```

(a) Calling program

```

PROADD  PROC  FAR
        PUSH AX      ;SAVE REGISTERS
        PUSH CX
        PUSH SI
        PUSH DI
        MOV  SI,[BX]   ;GET THE ADDR OF
        MOV  DI,[BX+2];ARRAY, THE VALUE OF COUNT, AND
        MOV  CX,[DI]   ;THE ADDR OF
        MOV  DI,[BX+4];RESULT
        XOR  AX,AX    ;CLEAR AX
NEXT:   ADD  AX,[SI]   ;ADD THE ELEMENTS
        ADD  SI,2     ;OF THE ARRAY
        LOOP NEXT    ;TO AX
        MOV  [DI],AX   ;RETURN RESULT
        POP  DI      ;RESTORE REGISTERS
        POP  SI
        POP  CX
        POP  AX
        RET           ;RETURN
PROADD  ENDP

```

(b) Procedure

Figure 4-20 Passing parameters through a parameter address table.

which retrieves the needed addresses from the table. Immediately before calling PROADD the offsets of ARY, COUNT, and SUM are put into the array TABLE and the address of TABLE is put into BX. Then after entering PROADD the addresses of ARY and SUM are taken from TABLE and put into SI and DI, respectively, and the value of COUNT is put into CX. The previous contents of these registers and of AX, which is used as an accumulator, are stored on the stack at the beginning of the procedure and restored before the procedure is exited. The procedure could be in the same code segment as the call, a separate code segment, or a separate source module. In this example the data, stack, and calling program were combined into the same segment. Note that prior to the call statement, the DS register must point to the data segment in which the parameters and parameter address table reside.

Figure 4-21 shows how the parameter addresses can be passed through the stack. Figure 4-21(a) gives the calling sequence and Fig. 4-21(b) shows the procedure. To demonstrate an alternate approach, the data, stack, and code have been put in separate segments in the calling program. As in the previous example, prior to a call to PROADD, DS must point to the segment where the parameters reside. To provide better access to the parameters, in the procedure module the top portion of the stack has been given a structure and structure names are used in the operands. (Note that, because the calling program reserves the space, the structure does not need to be invoked. The structure is used solely as a means of giving meaningful names to its elements in the source code.) The BP register is used to index the elements in the structure. Note that the stack pointer is adjusted by the RET instruction so that upon return to the calling program the TOS will be the same location as just before the parameter addresses were pushed onto the stack. Figure 4-22 illustrates the stack while PROADD is executing. After the contents of the registers have been popped the TOS will contain the old (IP), but after the RET instruction it will be the same as the TOS immediately prior to the beginning of the calling sequence.

Although the above discussion implies that the parameter addresses are always passed instead of the parameters, this is not necessarily the case. If an input parameter (as opposed to a result) is only 1 byte or one word long, it would be easier to put the parameter itself into the parameter table or onto the stack. In the preceding examples, COUNT could have been passed instead of its address. However, because the person writing the procedure must know exactly how the required information is to be communicated, it is best for a programming group to establish a convention that is always adhered to even if it leads to less efficient code. Always following the same method is often a small sacrifice compared to the increased clarity of the overall program. When writing general purpose procedures, having a convention to follow is mandatory.

4-3-4 Recursive Procedures

Sometimes an algorithm is defined in terms of itself. That is, the algorithm is nested within itself in such a way that its computational procedure calls upon itself. As an example, Horner's rule for evaluating the polynomial

$$((\dots ((a_n x + a_{n-1}) x + a_{n-2}) x + \dots) x + a_1) x + a_0$$

involves multiplying x and a_n , adding a_{n-1} to the result, multiplying x times this result, adding a_{n-2} to this result, and so on. The next result is obtained by multiplying the previous result by x and adding the next lower coefficient. A single application of the algorithm consists of a multiplication and an addition and the overall result could be obtained by nesting the simple multiply-add algorithm within itself. Such algorithms are said to be *recursive* and are sometimes needed in systems programming for performing searches, decoding the syntax of a high-level language, and executing other tasks.

Recursive algorithms may be implemented by having a procedure call itself, but care must be taken to assure that each successive call does not destroy the

```

PARM SEG  SEGMENT
    ARY      DW      100 DUP(?)
    COUNT   DW      ?
    SUM     DW      ?
PARM SEG  ENDS
STACK SEG  SEGMENT
    TOS      DW      100 DUP(?)
    LABEL   WORD
STACK SEG  ENDS
CODE1    SEGMENT
ASSUME CS:CODE1, DS:PARM SEG, SS:STACK SEG
START:   MOV    AX, PARM SEG
         MOV    DS, AX          ;SET UP DS
         MOV    AX, STACK SEG
         MOV    SS, AX          ;SET UP SS
         MOV    SP, OFFSET TOS ;SET UP SP
         .
         .
         MOV    BX, OFFSET ARY
         PUSH   BX              ;PUSH ADDR OF ARY ONTO STACK
         MOV    BX, OFFSET COUNT
         PUSH   BX              ;PUSH ADDR OF COUNT ONTO STACK
         MOV    BX, OFFSET SUM
         PUSH   BX              ;PUSH ADDR OF SUM ONTO STACK
         CALL   FAR PTR PROADD
         .
         .
CODE1    ENDS

```

(a) Calling program

```

CODE2    SEGMENT
ASSUME CS:CODE2
STACK_STRC STRUC
    SAVE BP    DW      ?
    SAVE CS IP DW      2 DUP(?)
    PAR3_ADDR DW      ?
    PAR2_ADDR DW      ?
    PAR1_ADDR DW      ?
STACK_STRC ENDS
PROADD    PROC    FAR
    PUSH   BP          ;SAVE BP
    MOV    BP, SP       ;USE BP TO ACCESS PARAMETERS
    PUSH   AX          ;SAVE OTHER REGISTERS
    PUSH   CX
    PUSH   SI
    PUSH   DI
    MOV    SI, [BP].PAR1_ADDR ;GET ADDR OF PAR1 FROM STACK
    MOV    DI, [BP].PAR2_ADDR ;GET VALUE OF PAR2
    MOV    CX, [DI]       ;AND PUT IN CX
    MOV    DI, [BP].PAR3_ADDR ;GET ADDR OF PAR3 FROM STACK
    XOR    AX, AX        ;CLEAR AX
    ADD    AX, [SI]       ;COMPUTE SUM
    ADD    SI, 2
    LOOP
NEXT:    ADD    AX, [DI]
    POP    DI
    POP    SI
    POP    CX
    POP    AX
    POP    BP
    RET    6             ;ADJUST STACK AND RETURN
PROADD    ENDP
CODE2    ENDS

```

(b) Procedure

Figure 4-21 Use of a stack to pass parameters.

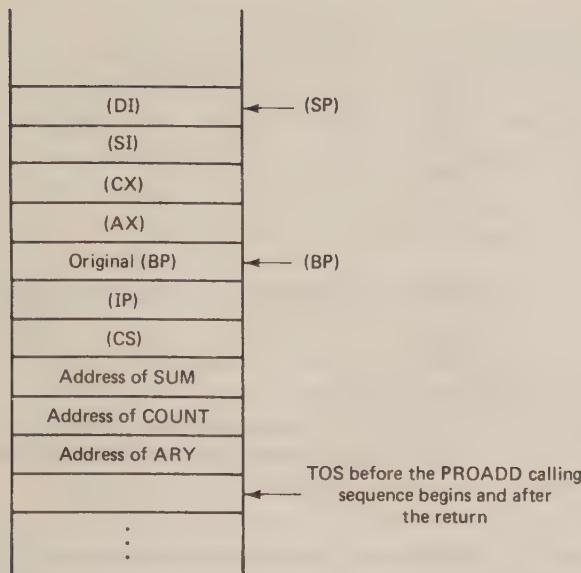


Figure 4-22 Contents of the stack while PROADD is executing.

parameters and results generated by the previous call and to make sure that the procedure does not modify itself. This means that each call must store its set of parameters, registers, and all temporary results in a different place in memory. To guarantee that separate areas of memory are used a stack is normally employed. By pushing the data onto the stack as shown in Fig. 4-23, it can be conveniently retrieved in reverse order as the process emerges from its nesting. The data stored by an application of the procedure is called a *frame*, and the 8086's BP register can be used to permit ready access to the individual items within a frame. Normally, a frame contains the saved register contents, the parameter addresses, and a temporary storage area. Properly storing the information on the stack provides for an orderly exit. To prevent indefinite nesting of a recursive algorithm, clearly the algorithm must include a conditional branch that will eventually allow the nest to be exited.

A trivial example of a recursive procedure is one for evaluating factorials. Computing a factorial can be done using a recursive procedure FACT as follows:

```

BEGIN FACT(N,RESULT)
    SAVE registers on stack
    IF N=0 THEN
        RESULT ← 1
    ELSE
        PUSH address of RESULT onto stack
        PUSH N-1 onto stack
        CALL FACT (N-1,RESULT)
        RESULT ← N*RESULT
    ENDIF
    RESTORE registers from stack
    DELETE parameters from stack
RETURN

```

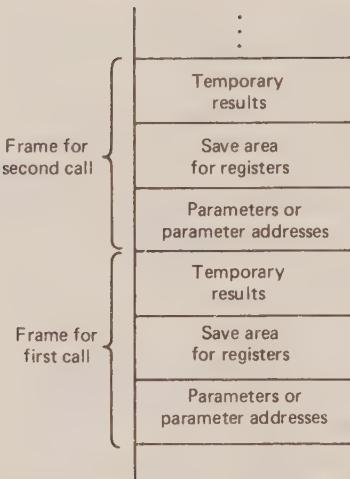


Figure 4-23 Use of a stack to dynamically provide storage during recursive calls.

The parameters N and RESULT represent the factorial to be found and the result, respectively. The program for implementing this algorithm is given in Fig. 4-24. Although the above description indicates only the parameters as being pushed onto and popped from the stack, the program includes the other necessary stack activity. A structure is used to define the current stack frame and the BP register is used to access the frame.

Figure 4-24 Recursive procedure to calculate the factorial of N.

```

CODE SEGMENT
    FRAME STRUCT
        SAVE_BP DW ?
        SAVE_CS_IP DW ? DUP(?)
        N DW ?
        RESULT_ADDR DW ?
    FRAME ENDS
ASSUME CS:CODE
FACT PROC FAR
    PUSH BP ;SAVE BP
    MOV BP,SP ;BP POINTS TO FRAME
    PUSH BX ;SAVE REGISTERS USED
    PUSH AX
    MOV BX,[BP].RESULT_ADDR ;LOAD ADDR OF RESULT INTO BX
    MOV AX,[BP].N
    CMP AX,0 ;N=0?
    JE DONE
    PUSH BX ;PUSH ADDR OF RESULT FOR NEXT CALL
    DEC AX
    PUSH AX ;PUSH N-1 FOR NEXT CALL
    CALL FACT PTR FACT
    MOV BX,[BP].RESULT_ADDR
    MOV AX,[BX]
    MUL [BP].N ;(AX) = N*RESULT
    JMP SHORT RETURN
DONE: MOV AX,1 ;(AX) = 1
RETURN: MOV [BX],AX ;RESULT = (AX)
    POP AX
    POP BX
    POP BP
    RET 4
FACT ENDP
CODE ENDS

```

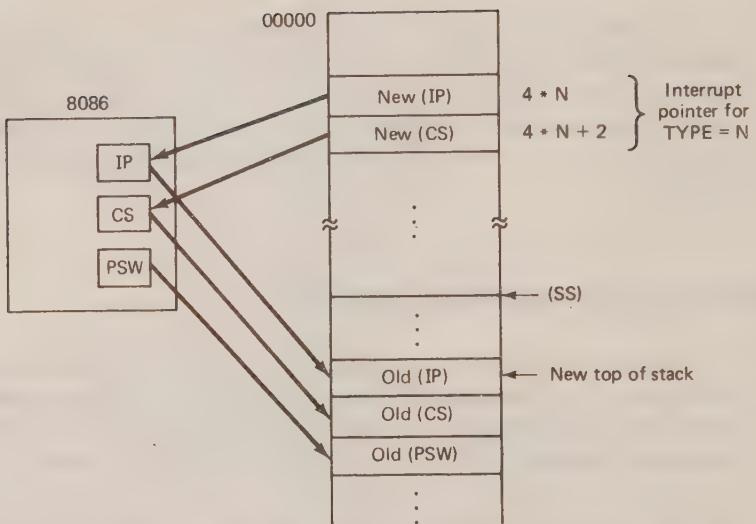
Even though using a recursive procedure is an inefficient and unreasonable way of making a simple factorial calculation, the example does demonstrate the essentials of recursive programming. As mentioned above, other more meaningful applications can be found in systems programming. Also, many of the concepts introduced here are needed when considering the reentrant procedures discussed in Chap. 7.

4-4 INTERRUPTS AND INTERRUPT ROUTINES

It is sometimes necessary to have the computer automatically execute one of a collection of special routines whenever certain conditions exist within a program or the computer system. The action that prompts the execution of one of these routines is called an *interrupt* and the routine that is executed is called an *interrupt routine*. There are two general classes of interrupts and associated routines. There are the *internal interrupts* that are initiated by the state of the CPU or by an instruction, and the *external interrupts* that are caused by a signal being sent to the CPU from elsewhere in the computer system. Typical internal interrupts are those caused by a division by zero or a special instruction and typical external interrupts are caused by the need of an I/O device to be serviced by the CPU. Because external interrupts primarily concern I/O, a detailed discussion of their applications is delayed until Chap. 6.

An interrupt routine is similar to a procedure in that it may be branched to from any other program and a return branch is made to that program after the interrupt routine has executed. The interrupt routine must be written so that, except for the lapse in time, the interrupted program will proceed just as if nothing had happened. This means that the PSW and the registers used by the routine must be saved and restored and the return must be made to the instruction following the last instruction executed before the interrupt. An interrupt routine is unlike a procedure in that, instead of being linked to a particular program, it is sometimes put in a fixed place (absolutely located) in memory. Because it is not linked to other segments, it can use only common areas that are absolutely located to communicate with other programs. Because some kinds of interrupts are initiated by external events, they occur at random points in the interrupted program. For such interrupts, no parameters or parameter addresses can be passed to the interrupt routine. Instead, data communication must be made through variables that are directly accessible by both routines.

Regardless of the type of interrupt, the actions that result from an interrupt are the same and are known as the *interrupt sequence*. The interrupt sequence for the 8086 is shown in Fig. 4-25. Some kinds of interrupts are controlled by the IF and TF flags and in those cases these flags must be properly set or the interrupt action is blocked. If the conditions for an interrupt are met and the necessary flags are set, the instruction that is currently executing is completed and the interrupt sequence proceeds by pushing the current contents of the PSW, CS, and IP onto the stack, inputting the new contents of IP and CS from a double word whose



Note: The instruction being executed at the time the interrupt occurs is completed before the interrupt sequence begins.

Figure 4-25 Interrupt sequence.

address is determined by the type of interrupt, and clearing the IF and TF flags. The new contents of the IP and CS determine the beginning address of the interrupt routine to be executed. After the interrupt has been executed the return is made to the interrupted program by an instruction that pops the IP, CS, and PSW from the stack.

The double word containing the new contents of IP and CS is called an *interrupt pointer* (or *vector*). Each interrupt type is given a number between 0 and 255, inclusive, and the address of the interrupt pointer is found by multiplying the type by 4. If the type is 9, then the interrupt pointer will be in bytes 00024 through 00027. Since it takes 4 bytes to store a double word, the interrupt pointers may occupy the first 1024 bytes of memory and these bytes should never be used for other purposes. Some of the 256 interrupt types may be reserved by the operating system and may be initialized when the computer system is first brought up. Users may tailor the other interrupt types according to their particular applications.

The kinds of interrupts and their designated types are summarized in Fig. 4-26 by illustrating the layout of their pointers within memory. Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts. From the figure it is seen that the type associated with a division error interrupt is 0. Therefore, if a division by 0 is attempted, the computer will push the current contents of the PSW, CS, and IP onto the stack, fill the IP and CS registers from addresses 00000 and 00002, and continue executing at the address indicated by the new contents of IP and CS. A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF and TF settings.

The type 1 interrupt is the single-step trap and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occur at the end

| | Contents | Address |
|----------------------|-----------------------|--|
| Pointer for type 0 | New (IP) for type 0 | 00000 |
| Pointer for type 1 | New (CS) for type 0 | Reserved for divide error |
| | New (IP) for type 1 | 00004 |
| Pointer for type 2 | New (CS) for type 1 | Reserved for single step trap — TF must be set |
| | | 00008 |
| Pointer for type 3 | | Reserved for nonmaskable interrupt |
| | | 0000C |
| Pointer for type 4 | | Reserved for one-byte interrupt instruction, INT |
| | | 00010 |
| Pointer for type 5 | | Reserved for overflow, INTO instruction |
| | | 00014 |
| Pointer for type 6 | | 00018 |
| | | 0001C |
| Pointer for type N | ⋮ | *Reserved for two-byte INT instructions and maskable external interrupts |
| | New (IP) for type N | 4*N |
| Pointer for type 255 | New (CS) for type N | 4*N + 4 |
| | ⋮ | 003FC |
| | New (IP) for type 255 | 00400 |
| | New (CS) for type 255 | |

*For the two-byte INT instruction the type is specified by the second byte and for an external interrupt the type is sent to the CPU over 8 of the data lines.

Figure 4-26 Layout of interrupt pointers.

of the next instruction that will cause a branch to the location indicated by the contents of 00004 through 00007. Because the TF flag is cleared by the interrupt sequence, interrupts will not occur after the instructions in the interrupt routine, but because the original PSW (including the enabled TF flag) is restored by the return, an interrupt will occur immediately after the instruction following the return. Therefore, an interrupt will take place after each instruction in the interrupted program as long as the TF flag is set. The single-step trap is used primarily for debugging by having the interrupt routine print out the contents of the pertinent registers and memory locations after the execution of each instruction. This gives the programmer a snapshot of his program after each instruction is executed. The TF flag can be enabled by pushing the PSW onto the stack, ORing the top of the

stack with 0100, and then popping the stack. It can be disabled by similarly ANDing the PSW with FFFF.

The type 2 interrupt is the nonmaskable external interrupt. It is the only external interrupt that can occur regardless of the IF flag setting. It is caused by a signal sent to the CPU through the nonmaskable interrupt pin and is discussed later.

The remaining interrupt types correspond to either interrupt instructions imbedded in the interrupted program or to external interrupts. The interrupt instructions are summarized in Fig. 4-27, and their interrupts are *not* controlled by the IF flag. The maskable external interrupts may initiate an interrupt sequence only if IF is 1 and are considered in Chap. 6. Instructions that initiate interrupts are sometimes called *software interrupts*. Also, given in the figure is the definition of the instruction IRET which is used to return from an interrupt routine. It is similar to the RET instruction except that it pops the original contents of the PSW from the stack as well as the return address.

The INT instruction has one of the forms

INT

or

INT Type

Figure 4-27 Interrupt instructions.

| Name | Mnemonics | Description |
|------------------------|---------------|--|
| Interrupt with type* | INT TYPE | $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (PSW)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (CS)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (TYPE*4)$ $(CS) \leftarrow (TYPE*4+2)$ |
| One-byte interrupt* | INT | Same as above except TYPE = 3 |
| Interrupt if overflow* | INTO | If (OF)=1 then $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (PSW)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (CS)$ $(SP) \leftarrow (SP)-2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (10H)$ $(CS) \leftarrow (12H)$ |
| Return from interrupt | IRET | $(IP) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ $(CS) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ $(PSW) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP)+2$ |

*These interrupts are not controlled by either the IF or TF flags.

Flags: Except for IRET, both IF and TF are cleared by the interrupt sequence and the remaining flags are not affected. IRET affects all flags by loading them from the stack.

Addressing modes: Not applicable except for INT TYPE, in which case TYPE must be a constant expression with a value between 0 and 255.

If the type does not appear, then the instruction is 1 byte long and has type 3; otherwise, the instruction is 2 bytes long and the interrupt pointer begins at

4*Type

The INT instruction is also often used as a debugging aid in cases where single-stepping provides more detail than is wanted. By inserting INT instructions at key points, called *breakpoints*, within a program a programmer can use an interrupt routine to print out messages and other information at these points. If the information to be printed is the same for all breakpoints, then the 1-byte INT instruction can be employed, but if the information is to vary, then a type needs to be included in the instruction. Figure 4-28 shows how the 2-byte INT instruction can be applied to debugging.

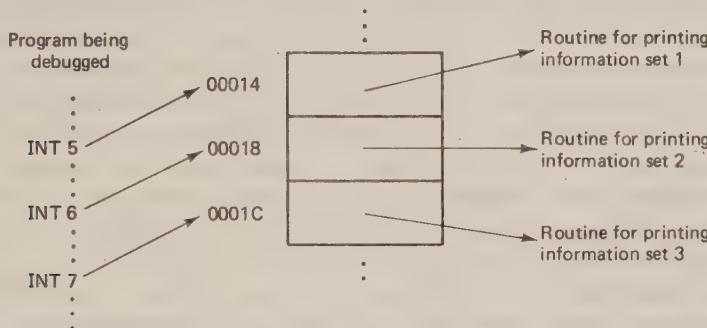
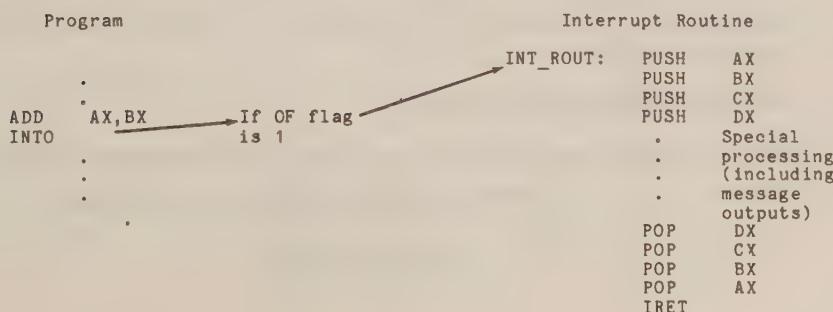


Figure 4-28 Use of the INT instruction for debugging a program.

The INTO instruction has type 4 and causes an interrupt if and only if the OF flag is set to 1. It is often placed just after an arithmetic instruction so that special processing will be done if the instruction causes an overflow. Unlike a divide-by-zero fault, an overflow does not cause an interrupt automatically; the interrupt must be explicitly specified by the INTO instruction. A typical application of this instruction is given in Fig. 4-29. Note that the PSW does not need to be saved because this was done by the interrupt sequence.

Figure 4-29 Use of the INTO instruction.



4-5 MACROS

The advantages of procedures are that they save memory and programming time by allowing code to be reused and provide a modularity that makes it easier to debug and modify a program. The disadvantage of procedures is the linkage associated with them. It sometimes requires more code to program the linkage than is needed to perform the task. When this is the case a procedure may not save memory and the execution time is considerably increased. For situations in which similar, but short, code segments are to appear several places within a program, a means is needed for providing the programming ease of a procedure while avoiding the linkage. This need is fulfilled by macros.

A *macro* is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each appearance. A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced; therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved.

The code that is to be repeated is called the *prototype code*, and the prototype code along with the statements for referencing and terminating it is called the *macro definition*. Included in the first statement of a macro definition is the macro's name. The procedure for using a macro is to give the macro definition and then cause the macro to be inserted at various points within a program by placing a statement that includes the macro's name at these points. These statements are known as *macro calls*. When a macro call is encountered by the assembler, the assembler replaces the call with the macro's code. This replacement action is referred to as a *macro expansion*.

In order to allow the prototype code to be used in a variety of situations, portions of the statements in this code (e.g., operands) are such that they can be replaced by different character strings each time the macro is expanded. These statement portions are called *dummy parameters* and are listed in the first statement of a macro's definition. The statement that calls the macro includes a matching list of *actual parameters* which are character strings that are to replace the dummy parameters when the macro is expanded. During a macro expansion, the first actual parameter replaces the first dummy parameter in the prototype code, the second actual parameter replaces the second dummy parameter, and so on. The correspondence of actual parameters to dummy parameters is similar to the matching of arguments to parameters when calling a subprogram in a high-level language such as FORTRAN.

4-5-1 ASM-86 Macro Facilities

For the ASM-86 assembler a macro definition is constructed as follows:

```
%*DEFINE (Macro name (Dummy parameter list))
(
    :
        Prototype code
)

```

where the macro name (which must be an identifier that begins with a letter and contains only letters, numbers, and underscore characters) is used to reference the macro, and the dummy parameters in the parameter list are separated by commas. ASM-86 requires that each dummy parameter appearing in the prototype code be preceded by a % character. A macro call has the form

%Macro name (Actual parameter list)

with the actual parameters being separated by commas.

As an example, the definition and use of a macro for multiplying two word-operands and storing the result, which is assumed not to exceed 16 bits, in a third operand is shown in Fig. 4-30. The macro definition shows that the macro first temporarily stores the contents of the AX and DX registers on the stack, carries out the multiplication, and then pops the stack so that AX and DX are restored. The dummy parameters are OPR1, OPR2, and RESULT. In the first macro call they are replaced by CX, VAR, and XYZ[BX], respectively, and in the second call they are replaced by 240, BX, and SAVE. If a dummy parameter appears in a statement, then the matching actual parameter must be such that the statement is valid; otherwise, an assembler error will occur. For the macro defined in Fig. 4-30 the call

%MULTIPLY (BX,240,SAVE)

would cause an assembler error because, in the expansion, the fourth statement would be

IMUL 240

and the IMUL instruction cannot have an immediate operand.

The ASM-86 assembler also permits dummy parameters to be concatenated to form a single character string. Given the macro definition

```
%*DEFINE (MSGGEN (LAB,NUM,XYZ))
(%LAB%NUM DB 'HELLO MR. %XYZ'
)
```

```
%*DEFINE (MULTIPLY (OPR1,OPR2,RESULT))
(
    PUSH DX
    PUSH AX
    MOV AX,%OPR1
    IMUL %OPR2
    MOV %RESULT,AX
    POP AX
    POP DX
)
:
:
:
%MULTIPLY (CX,VAR,XYZ[BX])
{
    PUSH DX
    PUSH AX
    MOV AX,CX
    IMUL VAR
    MOV XYZ[BX],AX
    POP AX
    POP DX
}
:
:
:
%MULTIPLY (240,BX,SAVE)
{
    PUSH DX
    PUSH AX
    MOV AX,240
    IMUL BX
    MOV SAVE,AX
    POP AX
    POP DX
}
```

Figure 4-30 Example of the use of a macro

the call

```
%MSGGEN (MSG,1,TAYLOR)
```

would result in the expansion

```
MSG1 DB 'HELLO MR. TAYLOR'
```

It is possible to define a macro with no dummy parameters, but in this case the call must not include any parameters. A useful example of a macro with no parameters is one for pushing the register contents at the beginning of a procedure. If the macro definition

```
/*DEFINE (SAVEREG)
(   PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    PUSH SI
    PUSH DI
```

)

were to appear in a source module, then

```
%SAVEREG
```

could be used at the beginning of each procedure in the module to save the registers. A similar macro could be used to restore the register contents at the end of each procedure.

There are two principal ways of designing macro capability into an assembler. One is to include the macro facilities in the first pass of the assembler and the other, the method used in developing ASM-86, is to extend the assembler by adding a pre-pass. The pre-pass, or *macro processor*, scans the source code for macro definitions and macro calls. When a macro definition occurs, the macro name and its associated instruction sequence are entered into a macro definition table. Then, when a macro call is recognized, the macro processor searches the macro definition table, and then substitutes the corresponding definition for the call statement, with the dummy parameters being replaced by actual parameters. This requires that a macro be defined before it is called (a restriction that could be eliminated with a two-pass macro processor). The output of a macro processor, which is the expanded source code, is then translated into machine code during an assembly process such as the one discussed in Sec. 3-11.

4-5-2 Local Labels

Quite often, a macro definition has to include labels. For example, consider a macro ABSOL, which replaces an operand by its absolute value. This macro might be defined as

```
/*DEFINE (ABSOL (OPER))
(   CMP %OPER,0
    JGE NEXT
    NEG %OPER
NEXT: NOP
```

After the macro ABSOL is called for the first time, the label NEXT will appear in the program and, therefore, it becomes defined. Any subsequent call for ABSOL will cause NEXT to again be defined. This will generate an error during the assembly process because NEXT has been associated with more than one location. One solution to this problem would be to have NEXT replaced by a dummy parameter. Then each call for ABSOL could use a different actual parameter for the label. This would require the programmer to keep track of which labels have already been used and to make certain that labels are not reused.

To avoid the problems associated with including the label in the parameter list, some assemblers permit the designation of special labels, called *local labels*, that have suffixes that increment each time the macros are called. For ASM-86 these suffixes are two-digit numbers that increment by 1 starting from zero; thus a different label is generated for each expansion. Labels are declared to be local by attaching

LOCAL List of Local Labels

to the end of the first statement in the macro definition. In the above example, ABSOL could be redefined as follows:

```
%*DEFINE (ABSOL (OPER)) LOCAL NEXT
(
    CMP    %OPER,0
    JGE    %NEXT
    NEG    %OPER
%NEXT:   NOP
)
```

Assuming this definition of ABSOL, the first two calls, %ABSOL (VAR) and %ABSOL (BX), would result in the following expansions:

```
CMP    VAR,0
JGE    NEXT00
NEG    VAR
NEXT00: NOP
.

.

.

CMP    BX,0
JGE    NEXT01
NEG    BX
NEXT01: NOP
```

4-5-3 Nested Macros

It is possible for a macro call to appear within a macro definition. This is referred to as *macro nesting* and has the limitation that all macros included in the definition of a given macro must be defined before the given macro is called. Any macro, assuming that its dummy parameters are properly defined and used, can be called directly or from within one or more other macros. An example of macro nesting is shown in Fig. 4-31. In this example the outer macro DIFSQR is for computing

$$(OPR1 - OPR2)^2$$

and it uses the macro DIF for computing the difference.

```
%#DEFINE (DIFSQR (OPR1,OPR2,RESULT))
(
    PUSH    DX
    PUSH    AX
%DIF ($OPR1,%OPR2)
    IMUL    AX
    MOV     %RESULT,AX
    POP     AX
    POP     DX
)
%#DEFINE (DIF (X,Y))
{
    MOV     AX,%X
    SUB     AX,%Y
}
:
%DIFSQR (VAR1,VAR2,ERROR)
{
    PUSH    DX
    PUSH    AX
    MOV     AX,VAR1
    SUB     AX,VAR2
    IMUL    AX
    MOV     ERROR,AX
    POP     AX
    POP     DX
}
```

Figure 4-31 Example of nested macros.

It is also possible to include macro definitions within macro definitions. A way of defining macros for performing a variety of binary operations and storing the result in an arbitrary location is shown in Fig. 4-32. The macro DEFMAC contains the definition of a macro whose name is the dummy parameter MACNAM. Therefore, when DEFMAC is called it will cause a macro to be defined whose name is the actual parameter in the call statement that matches MACNAM in the first statement of DEFMAC. The statement

```
%DEFMAC (ADDITION,ADD)
```

```
%#DEFINE (DEFMAC (MACNAM,OPERATOR))
(%#DEFINE (%MACNAM (X,Y,Z))
(
    PUSH    AX
    MOV     AX,%X
    %OPERATOR AX,%Y
    MOV     %Z,AX
    POP     AX
)
)

;
;DEFINE THE MACRO ADDITION
%DEFMAC (ADDITION,ADD)
.

;
;DEFINE THE MACRO SUBTRACT
%DEFMAC (SUBTRACT,SUB)
.

;
;DEFINE THE MACRO LOGOR
%DEFMAC (LOGOR,OR)
.

;
;MACRO CALL FOR ADDITION
%ADDITION (VAR1,VAR2,VAR3)
```

Figure 4-32 Using a macro to define macros.

causes a macro ADDITION to be defined that adds the contents of two locations and puts the result in a third location. Similarly, the macros SUBTRACT and LOGOR are defined for subtracting and taking a logical OR. The call

```
%ADDITION (VAR1,VAR2,VAR3)
```

would result in the expansion

| | |
|------|---------|
| PUSH | AX |
| MOV | AX,VAR1 |
| ADD | AX,VAR2 |
| MOV | VAR3,AX |
| POP | AX |

Because using macros to define macros results in code that is difficult to read, this capability should be applied only to very special situations. Ordinary nesting of macros can also produce code that is difficult to read, but can be quite helpful, and this disadvantage is more likely outweighed by the advantages than is the case when nesting definitions.

4-5-4 Controlled Expansion and Other Functions

It is frequently necessary to write a source module that is assembled differently for different situations. For instance, the module may be designed to service two types of terminals and parts of the source code may need to be written according to the exact terminal used, even though most of the source code is the same for both types of terminals. In another situation, it may be desirable for the expansion of a macro's prototype code to depend on the types of the actual parameters passed to it by the call, thus allowing the call to determine whether or not some of the prototype code should be ignored or which of two alternate sets of prototype code is to be used. The capability of being able to select the code that is to be assembled is called *controlled expansion* (or *conditional assembly*) and for the ASM-86 assembler this capability is built into the preprocessor.

For ASM-86 the selection of which code is to be assembled is determined by the %IF control function defined in Fig. 4-33. Also defined in Fig. 4-33 are the %REPEAT and %WHILE control functions, which permit source code to be duplicated before it is assembled. For the %REPEAT function the expression must be arithmetic and evaluate to the number of times the code block is to be repeated in the source code. For the %IF and %WHILE functions the expression may be an arithmetic expression, two arithmetic expressions joined by one of the following relational operators:

| | |
|-----------------|-----------------------------|
| EQ—Equal | NE—Not equal |
| LT—Less than | LE—Less than or equal to |
| GT—Greater than | GE—Greater than or equal to |

or a complete logical expression which includes a combination of the logical operators NOT, AND, OR, and XOR. In all cases the expression results in a TRUE

| Format | Description |
|---|---|
| %IF (expr) THEN (: } Code Block 1) ELSE (: } Code Block 2) FI | If expr evaluates to an odd number or TRUE, code block 1 is included in the expansion; otherwise code block 2 is included. |
| %IF (expr) THEN (: } Code Block) FI | If expr evaluates to an odd number or TRUE, the code block is included in the expansion; otherwise the code block is ignored. |
| %REPEAT (expr) (: } Code Block) | The code block is duplicated "expr" times. |
| %WHILE (expr) (: } Code Block) | The code block is repeated until expr evaluates to an even number or FALSE. |
| %EXIT | Terminates the current macro expansion. |

Note: If the expression in a %IF or %WHILE function is arithmetic, an odd number will be interpreted as TRUE and an even number as FALSE. If the expression is logical, TRUE is represented by -1H and FALSE by 00H.

Figure 4-33 Control functions.

or FALSE value. An odd number is interpreted as TRUE and an even number as FALSE if the expression is arithmetic, and -1H as TRUE and 00H as FALSE if the expression is not arithmetic. (Since such expansions are evaluated by the macro processor, symbols defined by assembler directives, such as EQU, cannot be used in an expression.)

The following code

```
%IF (%VECTYPE EQ 0) THEN
(
    INC     VECT0
)
ELSE
(
    INC     VECT1
)FI
```

would cause INC VECT0 to be assembled if the value of VECTYPE equals 0;

otherwise INC VECT1 is to be assembled. The code

```
%IF (%REG_NUM) THEN
(
  %REPEAT (3)
  (
    ADD AX,INCREMENT
  )
)FI
```

would result in (INCREMENT) being added to AX three times if REG_NUM is odd; otherwise, the ADD instruction is not assembled.

To prevent the code block associated with a %WHILE control function from being expanded indefinitely it must contain statements that change the value of the expression in the %WHILE statement. This is normally done by using the %SET function given in Fig. 4-34 to change the value of a name that appears in the expression in the %WHILE statement. For example,

```
%SET (X,0)
%WHILE (%X LT 10 AND %VAR GT 0)
(CONS%X DB %X
%SET (X,%X+1)
)
```

would cause

| | | |
|---------|----|-----|
| CONS00H | DB | 00H |
| CONS01H | DB | 01H |
| CONS02H | DB | 02H |
| . | | |
| CONS09H | DB | 09H |

to be assembled if %VAR is greater than 0; otherwise, the code block would be skipped. Note how the value of %X, which is the ASCII representation of X, is concatenated with the string CONS by appending %X to CONS.

The last definition given in Fig. 4-33 is that of %EXIT. This control function is used to terminate a macro definition early, e.g., from within a %IF code block.

Also given in Fig. 4-34 is the definition of the %LEN function, which returns the length of its argument. If %LEN appears in the expression in a %IF, %REPEAT, or %WHILE control function, then the value returned by %LEN is a

Figure 4-34 %SET and %LEN functions.

| Format | Description |
|-------------------------|--|
| %SET (Name,Value) | The name is associated with the value. |
| %LEN (Character string) | If %LEN is part of the expression in a %IF, %REPEAT, or %WHILE function, then the length of the string is returned as a number; otherwise the returned value is the ASCII representation of the length appended with an ASCII "H". |

number, but if %LEN is used elsewhere, then the ASCII representation of the value is returned with an ASCII "H" appended. If the macro

```
%*DEFINE (MSGGEN (X,Y,Z))
(%X DW %LEN(%Z)
%Y DB '%Z'
)
```

were called by the statement

```
%MSGGEN (NUM1,MSG1,GOOD MORNING)
```

the expansion would be

```
NUM1 DW 0CH
MSG1 DB 'GOOD MORNING'
```

In addition to the relational operators which compare numeric values, ASM-86 also includes the set of functions defined in Fig. 4-35 for lexically comparing character strings. Each of the functions results in a TRUE (-1H) or FALSE (00H) value.

Let us now consider how the MULTIPLY macro defined in Fig. 4-30 can be extended so that AX can be used to store the result. As this macro was originally defined AX could not be used as the third argument because, if it were, the

```
POP AX
```

instruction would destroy the product. However, if the definition of MULTIPLY were changed to

```
%*DEFINE (MULTIPLY (OPR1,OPR2,RESULT))
(
    PUSH DX
    %IF (%NES(%RESULT,AX)) THEN
    (
        PUSH AX
    )FI
        MOV AX,%OPR1
        IMUL %OPR2
    %IF (%EQS(%RESULT,AX)) THEN
    (
        POP DX
    )
    ELSE
    (
        MOV %RESULT,AX
        POP AX
        POP DX
    )FI
)
```

then the statement

```
%MULTIPLY (CX,VAR,AX)
```

would expand to

```
PUSH DX
MOV AX,CX
IMUL VAR
POP DX
```

| Format | Description* |
|---------------------------|--|
| %EQS (String 1, String 2) | TRUE(-1H) if the two strings are identical. |
| %NES (String 1, String 2) | TRUE(-1H) if the two strings are not identical. |
| %LTS (String 1, String 2) | TRUE(-1H) if String 1 alphabetically precedes String 2. |
| %LES (String 1, String 2) | TRUE(-1H) if String 1 is alphabetically equal to or precedes String 2. |
| %GES (String 1, String 2) | TRUE(-1H) if String 1 is alphabetically equal to or follows String 2. |
| %GTS (String 1, String 2) | TRUE(-1H) if String 1 alphabetically follows String 2. |

*FALSE is represented by 00H.

Figure 4-35 String comparing functions.

6 PROGRAM DESIGN

Almost all programs are structured according to a hierarchy, with the top of the hierarchy being a controlling program module, called the *main program*, that directs the execution of the modules that lie just below it in the hierarchy. In turn, these principal submodules control the use of their submodules, and so on. As indicated in Fig. 4-1, submodules may appear several places within a hierarchical diagram, e.g., a module for inputting a specific set of data or for carrying out BCD division.

The design of a program is strongly related to its hierarchical structure. After the problem to be programmed has been precisely stated, the first step is to determine the major tasks. Typically, these tasks would include an input task, an output task, and several processing tasks. The modules corresponding to the major tasks should be carefully defined along with their relationships to each other. Then special jobs within the principal modules are assigned to submodules and the relationships within the resulting substructures are defined. This process, which is referred to as *top-down design*, is continued until the program has been subdivided into pieces that can be readily understood and implemented.

During a top-down design, it is important to recognize the fundamental nature of the communication between the modules, the *data coupling*, and the way in which the modules call and control each other, the *control coupling*. In breaking a program into tasks and defining the corresponding modules one must first base the divisions on the functions to be performed. It is generally unwise to break a function (e.g., evaluating a determinant) into modules on the same level, modules that do not form a subhierarchical structure. Beyond the functional subdivision, the decisions regarding the subdivision of a program should be based on data and control coupling. There are no hard and fast rules for making these decisions, but in most cases the following rules apply:

1. If a task requires that an inordinate amount of data be transferred between

it and its parent task, then the two tasks should probably be programmed into the same module. What is meant by "inordinate" depends on the size and importance of the subordinate task and how easily the data can be referenced. The use of common is a possibility in reducing parameter passing, but this approach also has problems associated with it.

2. Multiple entry points to and exit points from a module should be avoided. The ties between the modules should be kept as simple as possible.

Modules are normally programmed as procedures and complex linkage between procedures should be avoided.

Module length is also an important consideration when deciding how the tasks should be divided up between the modules. The more statements a module contains, the more difficult it is to understand and debug. On the other hand, a module that is very short is possibly not worth the linkage required to communicate with it (unless it is a macro and requires no linkage). Generally, a procedure module that consists of fewer than 20 statements or more than 100 statements should be examined carefully and consideration should be given to combining it with its parent module or assigning some of its work to subordinate modules.

Once the hierarchical structure of a program has been determined, the structure of the data to be accessed by the program must be decided upon and the inputs and outputs of each module must be clearly stated. Some of the data may be put in common areas that may be accessed by all of the modules and some of it may be localized and must be passed between modules via a stack or parameter table. Keep in mind that localized data can be passed only to those modules that are subordinate to the one in which it is defined.

The statements of the inputs and outputs can be made by means of a module description. Normally, such a description simply lists the inputs and outputs and gives a brief narrative description of the function. It is important that a module description be given for each module and that these descriptions can be easily related to their corresponding blocks in the hierarchical diagram.

As a simple example let us consider a program that inputs a set of coefficients to an array A and numbers to variables X and N, performs the computation

$$Y \leftarrow A_N X^N + \dots + A_1 X + A_0$$

and outputs the contents of variable Y. Suppose that all quantities are input, processed, and output as 16-digit unpacked BCD numbers. A hierarchical diagram of the program is given in Fig. 4-36. It consists of the modules MAIN, INPUT, EVALUATE, ADDITION, MULTIPLICATION, and OUTPUT. MAIN calls upon INPUT to input the needed data, uses EVALUATE to carry out the computation, and then directs OUTPUT to print the value of Y. EVALUATE uses the submodules ADDITION and MULTIPLICATION to perform the arithmetic operations. Module descriptions for INPUT, EVALUATE, and ADDITION are shown in Fig. 4-37.

After the modules have been determined, their inputs and outputs have been defined, the data structure has been specified, the hierarchical diagram has been

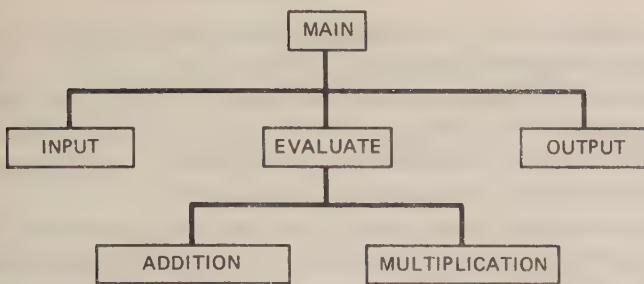


Figure 4-36 Hierarchical diagram for the polynomial evaluation example.

drawn, and the module descriptions have been written, the programmer is ready to design and code the individual modules. At this point the algorithms to be used in the modules may have already been decided upon because some knowledge of the algorithms is normally needed in assigning the work to the modules. However, the details of the algorithms will still be unresolved.

Designing and coding an assembler language module is essentially the same

Figure 4-37 Module descriptions for some of the modules in the polynomial evaluation example.

Name: INPUT

Inputs: From input terminal
N, X, and A_N, \dots, A_0

Outputs: To MAIN through a parameter table:
N, X, and A_N, \dots, A_0

Function: To input all of the data needed for evaluating

$$A_N X^N + \dots + A_1 X + A_0$$

(a) Input

Name: EVALUATE

Inputs: From MAIN through a parameter table:
N, X, A_N, \dots, A_0

Output: To MAIN through a parameter table:
Value of the polynomial to Y

Function: To evaluate

$$Y \leftarrow A_N X^N + \dots + A_1 X + A_0$$

using Horner's rule and the submodules ADDITION and MULTIPLICATION for performing the arithmetic operations on the 16-bit unpacked BCD numbers.

(b) Evaluate

Name: ADDITION

Inputs: From EVALUATE through a parameter table:
Operands to be added

Output: To EVALUATE through a parameter table:
Sum

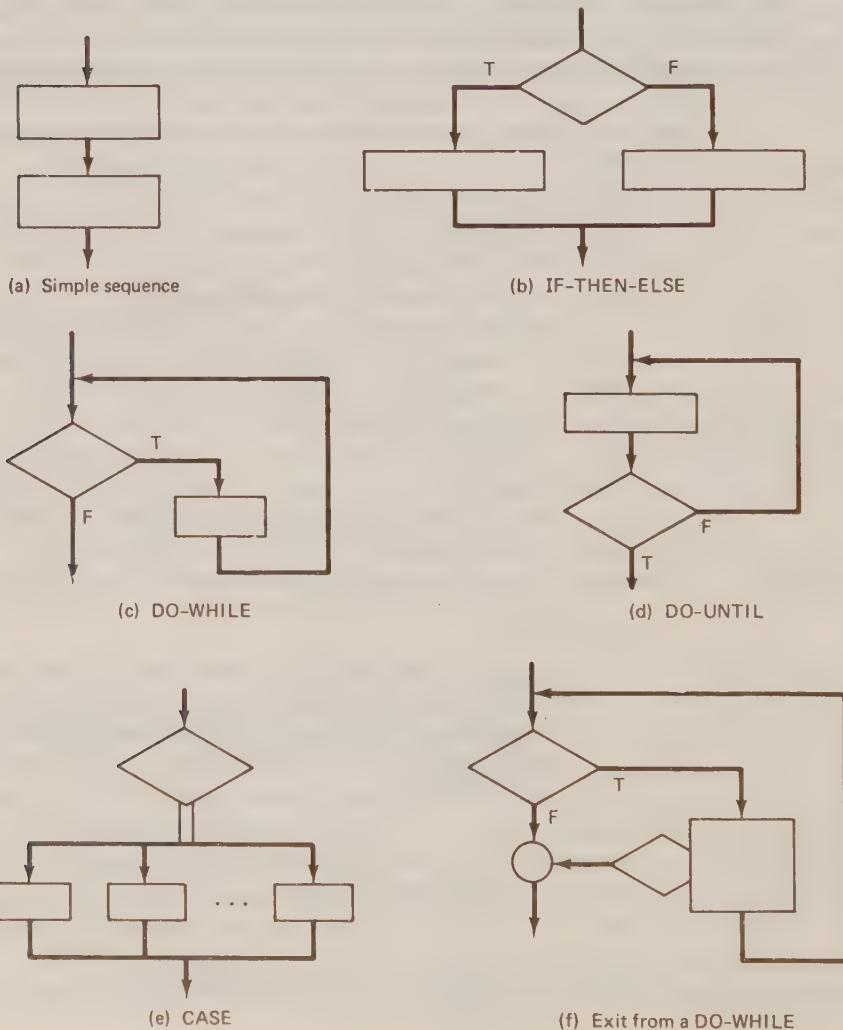
Function: To add two 16-bit unpacked BCD numbers.

(c) Addition

as that of a high-level language module, although some extra thought may be given to efficiency. The standard elementary programming structures, or *constructs*, are shown in Fig. 4-38(a) through (e). Modules should be constructed by placing these constructs in simple sequence or by nesting them. An exit may be used as a modification to an IF-THEN-ELSE, DO-WHILE, DO-UNTIL, or CASE construct. An exit from a DO-WHILE loop is shown in Fig. 4-38(f).

A flowchart of a representative combination of constructs is given in Fig. 4-39. The flowchart shows a THEN branch that contains only a procedure call and an ELSE branch that consists of a simple sequence. The first part of the simple sequence is a DO-UNTIL loop with an exit, the second part is a procedure call, and the third part is an IF-THEN-ELSE construct with a null ELSE branch.

Figure 4-38 Elementary constructs and an example of an exit.



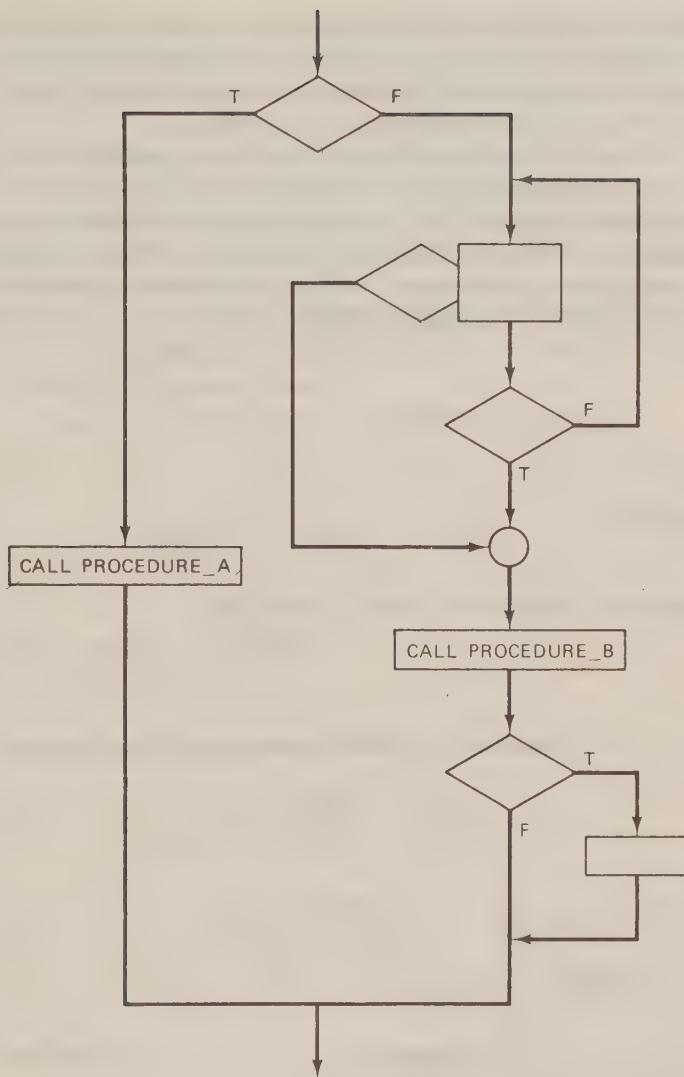


Figure 4-39 Representative combination of constructs.

Exceptions to the standard constructs should be avoided, but because more attention is given to efficiency when programming in assembler language, more deviations are acceptable than when programming in a high-level language. The most prevalent exception is an exit which branches to a point that is a few statements beyond the end of its construct. Especially in assembler language programming, it is sometimes convenient to put a small segment of code between the end of the construct and the branched-to point.

The reason for relying on the elementary constructs is that the resulting programs are easier to read and debug. Except in looping, backward branches

should not be used because they can be very confusing and frequently introduce logical errors that are hard to find. Nonstandard structures can be avoided by the judicious application of procedures and programming flags (i.e., variables that are for controlling the logical flow of a program).

Typical 8086 assembler language implementations of all but the simple sequence construct are given in Fig. 4-40. The IF-THEN-ELSE structure shown in Fig. 4-40(a) assumes that one of two actions is to be taken based on whether two quantities are equal or unequal. If the action program sequences are lengthy, they could be put in procedures, or if they are not put in procedures, the JNE instruction would need to be replaced with the combination

```
JE      ACTION_1
JMP    NEAR PTR ACTION_2
ACTION_1:
```

and the

```
JMP    SHORT EXIT
```

instruction would need to be replaced with

```
JMP    NEAR PTR EXIT
```

Figure 4-40 Typical 8086 implementations of the elementary constructs.

| | |
|--|--|
| <pre>MOV AX, VAR1 CMP AX, VAR2 JNE ACTION_2 : } Action 1 ACTION_2: JMP SHORT EXIT : } Action 2 EXIT: :</pre> | <pre>BEGIN: CMP AX, QUAN JNE EXIT : } Initialization : } Action 1 : } Action 2 EXIT: JMP SHORT BEGIN :</pre> |
|--|--|

(a) IF-THEN-ELSE

(b) DO-WHILE

| | |
|---|---|
| <pre>BEGIN: : } Initialization : } Action 1 : } Action 2 : } Action 3 EXIT: :</pre> | <pre>CMP CODE, 0 JE ACTION_2 CMP CODE, 1 JE ACTION_3 : } Action 1 : } Action 2 : } Action 3 EXIT: :</pre> |
|---|---|

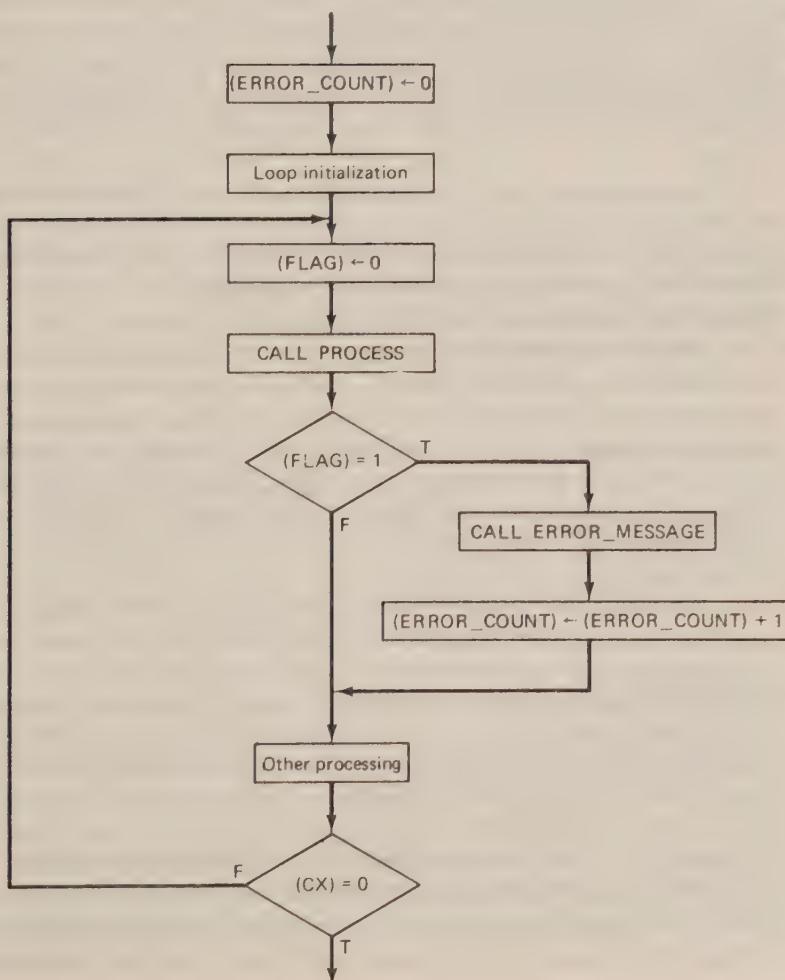
(c) DO-UNTIL

(d) CASE

The DO-WHILE implementation shown in Fig. 4-40(b) is based only on a comparison of two quantities, but the test in the DO-UNTIL loop in Fig. 4-40(c) is based on both a count and a comparison. The CASE structure in Fig. 4-40(d) assumes that different actions are to be taken depending on the value of CODE. If CODE=0, then Action 2 is to be executed; if CODE=1, then Action 3 is to be executed; and if CODE equals any other number, then Action 1 is to be performed. As with the IF-THEN-ELSE example, if the actions in the looping and CASE implementations are too lengthy, the code will need to be modified so that unconditional NEAR branches are used.

To demonstrate how a combination of constructs may be coded, let us consider the flowchart in Fig. 4-41 and the corresponding code in Fig. 4-42. The program

Figure 4-41 Implementation of a combination of constructs.



```

        MOV    ERROR_COUNT,0
        : } Loop initialization
        :
        BEGIN: MOV    FLAG,0
        : } Linkage for PROCESS
        :
        CALL   PROCESS
        CMP    FLAG,1
        JNE    OTHER
        :
        : } Linkage for ERROR_MESSAGE
        :
        CALL   ERROR_MESSAGE
        INC    ERROR_COUNT
        :
        OTHER: : } Other processing
        :
        LOOP   BEGIN
    
```

Figure 4-42 Code for the flowchart in Fig. 4-41.

is to enter a loop with `ERROR_COUNT` equal to 0. Inside the loop the program sets `FLAG` to 0 and calls the procedure `PROCESS`. If a processing error of some type occurs while `PROCESS` is executing, then `FLAG` is set to 1. Then, immediately after returning from `PROCESS` the contents of `FLAG` are tested and, if they are 1, the procedure `ERROR_MESSAGE` is called and `ERROR_COUNT` is incremented.

Also demonstrated in this example is the use of procedures to subdivide and shorten a module. It may have been possible to include one or both of the procedures called within the loop as in-line code (i.e., as part of a simple sequence), but by using procedures the module containing the loop is not cluttered by the `PROCESS` and `ERROR_MESSAGE` functions.

Pseudocode and flowcharts may be used to aid the coding process and then saved for documentation purposes, just as they are when programming in a high-level language. The translation of the pseudocode into assembler is not as straightforward because there is no correspondence between the pseudocode keywords and keywords in the language statements. There are normally several assembler language statements for each pseudocode statement. In addition to detailed program flowcharts, system flowcharts may be utilized to illustrate the flow of data to and from files and I/O devices.

For assembler language programming considerable thought must be given to deciding when to use a procedure or loop, as opposed to in-line code. The linkage to a procedure or the initialization of a loop may require more code than is saved by utilizing the procedure or loop. For example, it would be simpler to put four `ADD` instructions in a simple sequence than to implement a loop. The four `ADD` instructions would, however, be inflexible and would only permit the addition of exactly five numbers. With procedures, several factors enter the picture. These factors include the amount of linkage, the length of the procedure, whether or not the procedure is in the same source module as the module that calls it, and whether or not it would be advantageous to put the procedure in a library. It may be beneficial to put a procedure that finds wide application in a library even though calling it may require more linkage code than is required for performing the task.

Macros are also an important part of modular program design. Although they do not reduce the machine code, they do shorten the source code and that is very important when examining the lengths of the modules.

Execution time is often important when programming in assembler language. In fact, frequently the reason for programming in assembler language instead of a high-level language is to give the programmer more control over a process so that the execution time can be reduced. If this is the case, then the use of loops and procedures should be examined carefully. The initialization, updating, testing, and branching associated with a loop and the linkage associated with a procedure take time.

4-7 PROGRAM DESIGN EXAMPLE

As an example of overall program design, suppose that personnel data records are kept on a mass storage device along with a directory that indicates where each personnel record is located on the device. What is needed is a program for updating these records, a program that can insert new records for new employees, delete records for terminating employees, and change the data in the records of the current employees. The program is to:

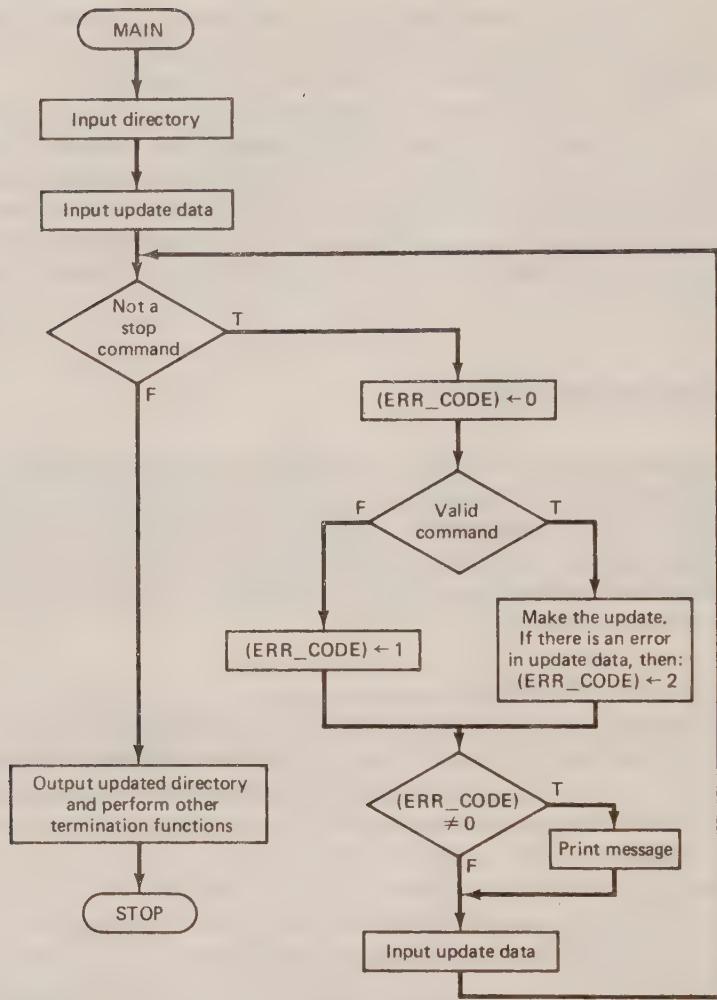
1. Input the directory.
2. Input an "S" (stop) command or a command that begins with a letter "I" (insert), "D" (delete), or "C" (change) and ends with a set of update data.
3. If the command letter is "S", then the program is to output the updated directory, clean up the system files, and terminate; otherwise, it is to continue with step 4.
4. Set an error code to 0.
5. If the command letter is "I", "D", or "C", continue with step 6; otherwise, set the error code to 1 and go to step 7.
6. Make the requested update. If there is an error in the update data, set the error code to 2.
7. If there is an error, print the appropriate error message.
8. Input a new set of update data and go to step 3.

Both a program flowchart and a system flowchart of the overall program are shown in Fig. 4-43.

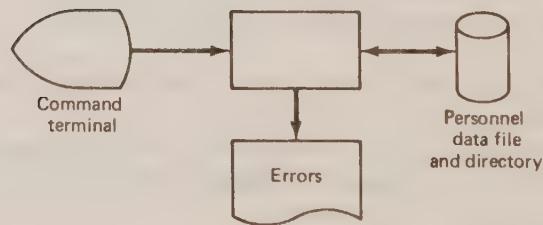
A hierarchical diagram showing a way of breaking down the program is given in Fig. 4-44. The modules are to perform the following functions:

MAIN—Control the activity by calling the other modules as they are needed. Also, it is to perform all program initialization and termination functions.

INPUTM—Input the directory and personnel records from the mass storage device.



(a) Program flowchart



(b) System flowchart

Figure 4-43 Flowcharts for the personnel records example.

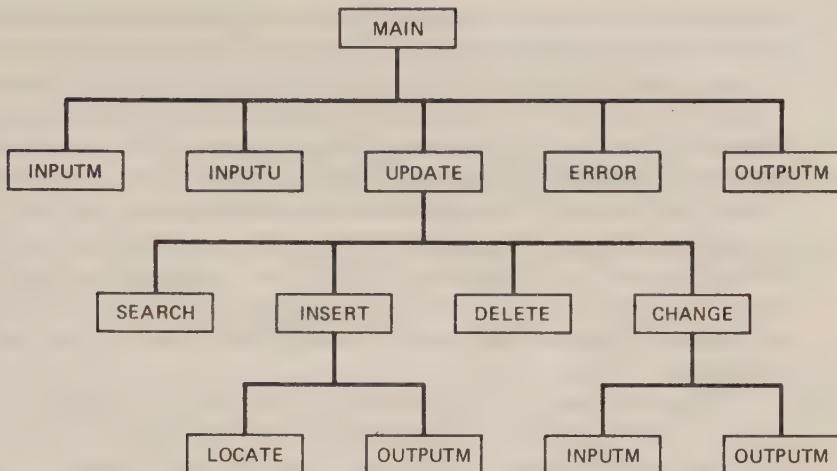


Figure 4-44 Hierarchical diagram for the personnel records example.

OUTPUTM—Output the directory and personnel records to the mass storage device.

INPUTU—Input the set of data needed to perform an update.

UPDATE—Determine the type of update and control the update activity.

SEARCH—Search the directory for a given record entry.

INSERT—Insert a new record entry in the directory and output the new record to mass storage.

LOCATE—Determine the information needed by the system to store the new record.

DELETE—Delete a record entry from the directory.

CHANGE—Input a record from mass storage, make the requested change, and output the updated record.

ERROR—Determine the type of error and print the corresponding error message.

Note that the error messages are printed by a separate module. The **ERROR** module receives a code value through the variable **ERR_CODE**, which is set by the other modules when an error is found, and prints the appropriate error message. This technique eliminates the need to have calls to an error-printing routine scattered throughout a program and thereby shortens the modules that may detect errors. It does, however, require the extra variable **ERR_CODE**, but in terms of aesthetics and debugging the trade-off is a good one.

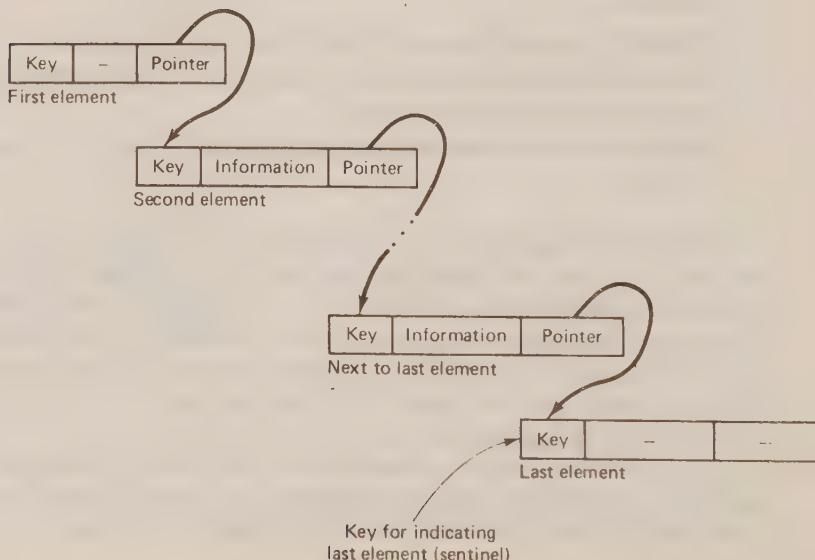
It is assumed that the directory is structured as a linked list that is ordered according to ascending employee numbers. As shown in Fig. 4-45, a *linked list* is a collection of elements, with one of the fields in each element being a pointer to

the location of the next element in the list. There are two other fields in each element, one being an identification key and the other being the information portion of the element. For the problem at hand, the list is a directory with a list element for each employee. The key is the employee's identification number and the information field provides the mass storage input and output procedures with the information needed to locate the employee's record on the mass storage device. On the 8086 the list can be defined as a collection of structures, with each structure being a list element. The pointer to the next element can be the physical position within the array of structures relative to the beginning, or base, address of the array. This means that by putting the base address in BX and the pointer in SI, the beginning address of an element is immediately available through based indexed addressing.

The available employee identification numbers are assumed to have four digits and are 1000 through 9999. The first element in the array is solely for pointing to the first employee entry. It is to have a key of 1 and is never to be deleted. The key fields for the elements in the array that are not currently linked into the list are to be set to 0. The last element in the array is to always be the last element in the list. It is not to be deleted and is to have 10000 in its key field. The 10000 signals that the end of list has been reached (such an element is known as a *sentinel*). The elements are to consist of one word for the key, three words for the information, and one word for the pointer.

To insert an element in an ordered linked list one must first locate the point at which the insertion is to be made and an unused element in the list array. Next the new element is put in the unused element. Last, the pointer in the element in

Figure 4-45 Organization of a linked list.



the list that immediately precedes the new element is put in the pointer field of the new element, and then it is changed to point to the new element. This process is illustrated in Fig. 4-46(a). If the key of the element to be inserted is 5082, the elements with keys adjacent to 5082 are

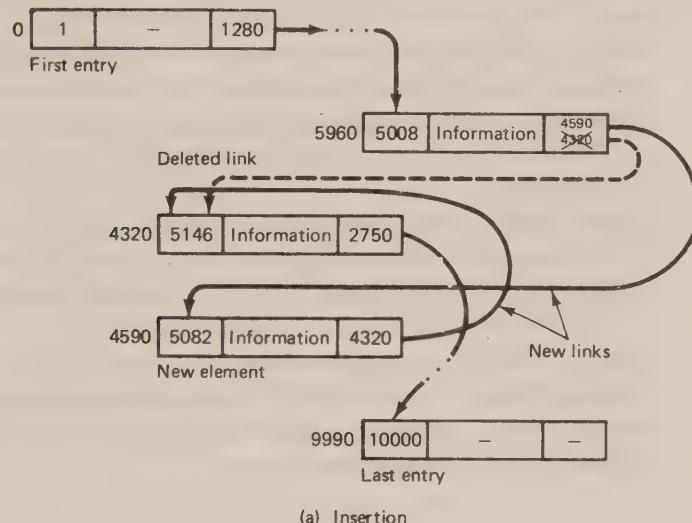
| | | | |
|---------------|------|-------------|------|
| Location 5960 | 5008 | Information | 4320 |
| Location 4320 | 5146 | Information | 2750 |

and the new element is to be put at location 4590 within the list array, then the new element would be

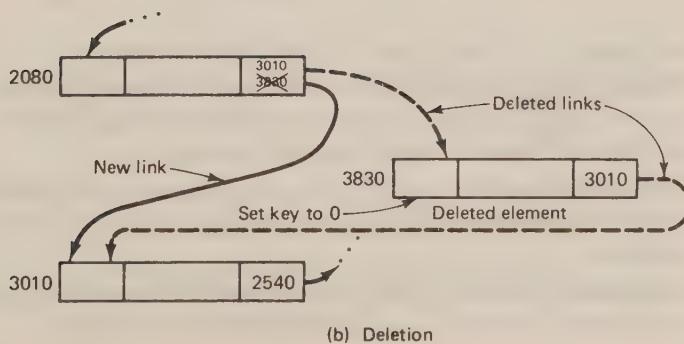
| | | | |
|---------------|------|-------------|------|
| Location 4590 | 5082 | Information | 4320 |
|---------------|------|-------------|------|

and the pointer in the element whose key is 5008 would be changed to 4590. The deletion of an arbitrary element is shown in Fig. 4-46(b). Note that the ordering

Figure 4-46 Inserting elements in and deleting them from a linked list.



(a) Insertion



(b) Deletion

is according to the keys, *not* the location in the list array. The independence of the keys from the array position is the primary advantage of a linked list.

Suppose that INPUTM, OUTPUTM, and LOCATE are library routines and that INPUTM and OUTPUTM are to receive, via a parameter table, the number of bytes to be input or output, a three-word designation of a mass storage location, and a memory address indicating where the data to be input or output are to be stored. LOCATE is to be passed the address of the three-word block in which it is to store the designation of an available mass storage location. The modules MAIN, INPUTU, UPDATE, and ERROR are to be assembled separately, but SEARCH, INSERT, DELETE, and CHANGE are to be included as separate procedures within the UPDATE source module. Therefore, UPDATE and these submodules can use the same variable names and parameter passing will be unnecessary.

A summary of the data items that are used by more than one module is:

DATA_BUFF (256 Bytes)—Memory area for storing personnel record currently being processed. Its location is passed through a parameter table.

NO_OF_BYTES (One Word)—Number of bytes to be input by INPUTM or output by OUTPUTM. Its location is to be passed using a parameter table.

NEW_REC_INFO (Three Words)—The information portion of the element in DIRECTORY that is filled by LOCATE. Its address is passed via a parameter table.

DIRECTORY (5000 Words, 1000 Elements)—Ordered linked list directory of the personnel data records. It is to be in a common area.

UPDATE_DATA (259 Bytes)—Memory area for storing update data brought in by INPUTU. It is to be in a common area.

ERR_CODE (1 Byte)—Code which indicates whether or not an error has occurred and, if so, what kind. It is to be in a common area.

SRCH_CODE (1 Byte)—Code which indicates the results of a search by SEARCH. The code definitions are:

0—key not found

1—key found

It is local to UPDATE and its immediate submodules.

PRIOR_ELE (One Word)—For temporarily storing the index to the element that was just previous to the last element examined during a search. It is local to UPDATE and its immediate submodules.

CUR_ELE (One Word)—For temporarily storing the index to the last element examined during a search. It is local to UPDATE and its immediate submodules.

KEY (One Word)—To store the key currently being searched for by SEARCH. It is local to UPDATE and its immediate submodules.

Sample module descriptions, those for UPDATE and SEARCH, are given in Fig. 4-47.

An update command, which consists of the information that INPUTU inputs to UPDATE_DATA, is to have a 1-byte command type (the letter "I", "D", "C", or "S") followed by a one-word key and the remaining information required to carry out the command. This information depends on the command and is determined as indicated below.

Insert—Complete record to be inserted, including its key.

Delete—Key for record to be deleted.

Change—Key for record to be changed and a field number-field data pair, where the field number indicates the field to be changed and the field data is the replacement data.

Stop—No other information is needed.

Figure 4-47 Sample module descriptions for the personnel records example.

```
Name: UPDATE
Common: Areas in common that may be affected or used
        DIRECTORY, UPDATE_DATA, ERR_CODE

Function: To determine the command code letter "I" (insert),
          "D" (delete), or "C" (change) and call the submodules
          SEARCH, INSERT, DELETE, and CHANGE to perform the
          requested action.. (A check for "S" (stop) is made in
          MAIN.) It first puts 0 in ERR_CODE and then, if it
          cannot identify the command letter, it changes the
          contents of ERR_CODE to 1 and returns to MAIN. Its
          submodules may change ERR_CODE to 2 if they detect an
          error.
```

(a) UPDATE

```
Name: SEARCH
Input: From UPDATE - the local variable
        KEY

Output: To UPDATE - the local variables
        PRIOR_ELE, CUR_ELE, SRCH_CODE

Common: Area in common that is used
        DIRECTORY, ERR_CODE

Function: It sets ERR_CODE to 2 and returns if (KEY) are
          outside the range 1000 to 9999. Otherwise,
          it searches DIRECTORY for the key in KEY and
          returns to UPDATE the indices to the elements
          which are such that:
```

The key is the largest key less than (KEY).
 The key is the smallest key greater than or
 equal to (KEY).

It puts 1 in SRCH_CODE if (KEY) are matched
 and 0 in SRCH_CODE if they are not matched.

(b) SEARCH

Figure 4-48 Code for the UPDATE and SEARCH modules.

```

PUBLIC  UPDATE
EXTRN  LOCATE:FAR, INPUTM:FAR, OUTPUTM:FAR
DATA SEG  SEGMENT
    SRCH_CODE  DB      ?
    PRIOR_ELE  DW      ?
    CUR_ELE    DW      ?
    KEY        DW      ?
DATA SEG  ENDS
COM SEG  SEGMENT COMMON
    D_ELEMENT  STRUC
        EMPNUM   DW      ?
        INFO     DW      3 DUP(?)
        POINTER  DW      ?
    D_ELEMENT  ENDS
    U_DATA    STRUC
        COMMAND  DB      ?
        COM_KEY  DW      ?
        COM_DATA DB      256 DUP(?)
    U_DATA    ENDS
    DIRECTORY D ELEMENT 1000 DUP(<>)
    ERR_CODE  DB      ?
    UPDATE_DATA U-DATA  <>
COM SEG  ENDS
UPDATE_SEG SEGMENT
    ASSUME CS:UPDATE_SEG, DS:DATA_SEG, ES:COM_SEG
UPDATE    PROC    FAR
    PUSH    AX          ;SAVE REGISTERS ON STACK
    PUSH    BX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSH    DI
    MOV     ERR_CODE, 0 ;ZERO ERR CODE
    MOV     AX, UPDATE_DATA.COM_KEY ;MOVE COMMAND KEY TO AX
    MOV     KEY, AX ;AND KEY
    CALL    NEAR PTR SEARCH ;SEARCH FOR KEY
    MOV     DL, UPDATE_DATA.COMMAND ;PUT COMMAND IN DL
    CMP     DL, 'I' ;IF COMMAND LETTER
    JNZ    NOT_I: ;IS NOT 'I' BRANCH TO
    CALL    NEAR PTR INSERT ;NOT_I, OTHERWISE MAKE INSERTION
    JMP    SHORT_EXIT ;AND BRANCH TO EXIT
    NOT_I:  CMP     DL, 'D' ;IF COMMAND IS NOT 'D'
    JNZ    NOT_D: ;BRANCH TO NOT D,
    CALL    NEAR PTR DELETE ;OTHERWISE MAKE DELETION
    JMP    SHORT_EXIT ;AND BRANCH TO EXIT
    NOT_D:  CMP     DL, 'C' ;IF COMMAND IS NOT 'C'
    JNZ    NOT_C: ;BRANCH TO NOT C,
    CALL    NEAR PTR CHANGE ;OTHERWISE MAKE CHANGE
    JMP    SHORT_EXIT ;AND BRANCH TO EXIT
    NOT_C:  MOV     ERR_CODE, 1 ;SET ERR CODE TO 1
    EXIT:   POP    DI          ;RESTORE REGISTERS
            POP    SI
            POP    DX
            POP    CX
            POP    BX
            POP    AX
    RET     ;AND RETURN
UPDATE    ENDP
.
.
SEARCH    PROC    NEAR
    MOV     SRCH_CODE, 0 ;ZERO SRCH CODE
    MOV     AX, KEY ;PUT KEY IN AX
    CMP     AX, 1000 ;CHECK IF KEY IS
    JB      ERROR ;IN THE RANGE 1000
    CMP     AX, 9999 ;TO 9999, IF SO
    JA      ERROR ;BRANCH TO OK
    JMP    SHORT_OK
    ERROR:  MOV     ERR_CODE, 2 ;ELSE, PUT 2 IN ERR CODE
    JMP    SHORT_NOT_FOUND ;AND EXIT TO NOT FOUND
    OK:    LEA     BX, DIRECTORY.EMPNUM ;PUT BASE ADDR IN BX

```

Figure 4-48 (Continued)

```

        MOV    SI,ES:[BX+8]      ;PUT POINTER IN SI AND
        MOV    PRIOR_ELE,0       ;PRIOR INDEX IN PRIOR_ELE
AGAIN:   CMP    AX,ES:[BX][SI]  ;COMPARE KEYS AND
        JE     FOUND           ;BRANCH TO FOUND IF EQUAL
        JL     GREATER         ;IF LESS THAN GO TO GREATER
        MOV    PRIOR_ELE,SI     ;ELSE, UPDATE PRIOR_ELE
        MOV    SI,ES:[BX][SI+8]  ;AND SI
        JMP    SHORT AGAIN
        MOV    SRCH_CODE,1
        MOV    CUR_ELE,SI
NOT_FOUND: RET
SEARCH    ENDP
.
.
.
UPDATE_SEG ENDS
END

```

Code for the source module containing UPDATE, which is based on the above command format, is given in Fig. 4-48. Only the complete code for UPDATE, SEARCH, and their data segments are included.

BIBLIOGRAPHY

1. Myers, Glenford J., *Software Reliability: Principles and Practices* (New York: Wiley-Interscience, 1976).
2. Bohl, Marilyn, *Tools for Structured Design* (Chicago: Science Research Associates, Inc., 1978).
3. Jenson, Randall W., and Charles C. Tonies, *Software Engineering* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1979).
4. *MCS-86 Macro Assembler Reference Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
5. *The 8086 Family User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).

EXERCISES

1. Consider the source modules

Source Module 1

| | | |
|-------|---------|------------|
| S_SEG | SEGMENT | STACK |
| TOP | DW | 100 DUP(?) |
| S_SEG | LABEL | WORD |
| D_SEG | ENDS | |
| VAR | SEGMENT | COMMON |
| D_SEG | DB | 50 DUP(?) |
| E_SEG | ENDS | |
| AREA | SEGMENT | AT 1000H |
| E_SEG | DW | 70 DUP(0) |
| | ENDS | |

```

C_SEG      SEGMENT      PUBLIC
          :
          } 500H bytes
          :
C_SEG      ENDS

      Source Module 2

S_SEG      SEGMENT      STACK
          DW      50 DUP(?)
S_SEG      ENDS
D_SEG      SEGMENT      COMMON
          VECT    DW      30
D_SEG      ENDS
C_SEG      SEGMENT      PUBLIC
          :
          } 1000H bytes
          :
C_SEG      ENDS

```

Assuming that the linker (or locator) places the segments in the order S_SEG, D_SEG, and C_SEG with the bottom of the stack beginning at address 20000, show exactly how the segments will be placed in memory by the loader.

2. Write a set of statements that declare the word named VAR1 and the FAR label LAB1 as being external and the variable VAR2 and label LAB2 as being local but accessible by other source modules.
3. Under what circumstances would the NEAR attribute be given to a label in an EXTRN statement?
4. Assume that:
 1. The double word named VAR1, the byte array named VAR2, and the NEAR label LAB1 are defined in source module 1 but are used by source modules 2 and 3.
 2. The word named VAR3 and the FAR label LAB2 are defined in source module 2 and VAR3 is used by source module 1 and LAB2 is used by source module 3.
 3. The FAR label LAB3 is defined in source module 3 and is used by source module 2.

Give the necessary EXTRN and PUBLIC statements for each module and illustrate how the variables and labels are matched by the linker (see Fig. 4-5).

5. Given the following set of code, indicate which segment addresses and offsets would be determined by the assembler and which would be determined by the linker, and give the reasons for your answers:

```

EXTRN      COST:WORD,ROUTINE:FAR
PUBLIC
BEGIN
DATA_1      SEGMENT
TOTAL      DW      50 DUP(?)
NUM       DW      ?
DATA_1      ENDS
DATA_2      SEGMENT
PART       DW      100 DUP(?)

```

```

DATA_2          ENDS
CODE           SEGMENT

MOV      AX,DATA_1
MOV      DS,AX

MOV      AX,SEG COST
MOV      ES,AX
MOV      AX,TOTAL
ADD      AX,ES:COST

MOV      AX,DATA_2
MOV      ES,AX
INC      ES:PART[SI]
MOV      CX,NUM
CMP      TOTAL[DII],CX
JE       NEXT
JMP      FAR PTR ROUTINE

NEXT:

```

```

CODE          ENDS

```

6. Give a sequence of code which would cause the register SI to be loaded from an externally defined variable COUNT.
7. Does an intersegment branch to an externally defined label need to be preceded by a set of instructions for loading the CS register? Why?
8. Give the code needed to permit the word variables NUM1, NUM2, NUM3, and NUM4 which are defined in source module 2 to be accessed in source module 1 as if they were defined in source module 1.
9. Assume the code:

```

S_SEG          SEGMENT      AT    1000H
                DW            200 DUP(?)
TOS            LABEL        WORD
S_SEG          ENDS

C_SEG          SEGMENT      SS:S_SEG
ASSUME         CS:C_SEG,
                MOV     AX,S_SEG
                MOV     SS,AX
                MOV     SP,OFFSET TOS

```

```

        PUSH      T_ADDR
        PUSH      AX
        PUSHF

        .
        .
        .

        POPF
        POP      AX
        POP      T_ADDR

        .
        .

C_SEG      ENDS

```

Draw an illustration of the stack that indicates its important points and the addresses of these points, and describe the actions caused by the pushes and pops.

10. Give a sequence of instructions that pushes the offsets of X, Y, and Z onto the stack.
11. Suppose that a certain set of code is reused several times by a program and that it is to use a separate stack from the remainder of the program. Give the instructions that are needed at the beginning and end of this set of code in order to switch stacks and then switch back again. Also give the code necessary to define the two stacks and to initially set (SS) and (SP).
12. Why must the NEAR or FAR attribute appear in a PROC statement?
13. Consider the following sequence of calls:
 1. MAIN calls NEAR procedure SUBA—return offset is 0400.
 2. SUBA calls NEAR procedure SUBB—return offset is 0A00.
 3. SUBB calls FAR procedure SUBC—return offset is 0100 and return segment address is B200.
 4. Return from SUBC to SUBB.
 5. SUBB calls NEAR procedure SUBD—return offset is 0C00.
 6. Return from SUBD to SUBB.
 7. Return from SUBB to SUBA.
 8. Return from SUBA to MAIN.
 9. MAIN calls procedure SUBC—return offset is 0600 and return segment address is 1000.
- Assuming that the only stack activity is due to the calls and returns, illustrate this activity by a series of diagrams as was done in Fig. 4-16.
14. Write a procedure COMPUTE for performing the computation

$$R \leftarrow X + Y - 3$$

where X, Y, and R are contained in words and COMPUTE is in the same code segment as the calling program. Also, show the definition of the data segment D_SEG that contains X and Y, the data segment E_SEG that contains R, the beginning part of the calling program, and a call being made to COMPUTE. Modify the code assuming that COMPUTE is in the same source module but is in a different code segment from the calling program. Modify the code again assuming that COMPUTE is in a different source module and in a different segment from the calling program and a parameter table is used for communication.

15. Write a procedure MULT that performs unsigned double-word binary multiplication of the contents of A and B and puts the 64-bit result in the four-word sequence beginning at PROD. MULT is the only code in its source module and A, B, and PROD are to be in a common area COM_SEG. Also, give the definition of COM_SEG in both the calling program source module and the procedure source module, but in the calling program source module the variables are to be named X, Y, and Z. Last, give a typical call to MULT.
16. Write a FAR procedure SEARCH that searches a byte array for a given byte and sets the value of a word parameter to the index of the element in the array if a match is found; otherwise, it puts a -1 in the index word parameter. The parameters are to be passed to SEARCH via a parameter address table. Also, give a sequence for calling SEARCH to search ARY for the (ID) and put the index in IDX.
17. Rewrite the procedure and calling sequence in Exercise 16 so that the parameter addresses are passed via the stack.
18. What is the primary purpose of allowing an expression to be included in the operand field of a RET instruction?
19. Write a set of procedures that will perform the unsigned binary arithmetic operations on double-word operands. The procedures are to assume that the operands have been pushed onto the stack just prior to their being called and are to return the results in the concatenation DX:AX. They are to be:
 - (a) ADDITION—Adds the two operands.
 - (b) SUBTRACT—Subtracts the two operands. The minuend is to be pushed onto the stack last.
 - (c) MULTIPLY—Multiplies the two operands and returns only the low-order 32 bits.
 - (d) DIVIDE—Divides the two operands and returns only the quotient. The dividend is to be pushed onto the stack first.
20. Write a procedure EVALUATE that will use the procedures requested in Exercise 19 to evaluate the expressions:
 - (a) $A \cdot X + B$
 - (b) $X \cdot (X + Y - Z)$For each case, the parameters are to be pushed onto the stack immediately before the procedure is called.
21. Write a procedure BCD_ADDSUB that will add or subtract two 16-digit BCD numbers. The procedure is to receive the beginning addresses of the operands and the location the result is to be put through the stack. Negative numbers are assumed to be in 10's complement format. Also to be passed via the stack is a flag for indicating whether addition or subtraction is to be performed. Subtraction is to be accomplished by adding the 10's complement of the subtrahend to the minuend. The 10's complement is to be found by a second procedure, TENS_COMP, that expects the beginning address of the number to be in the BX register. TENS_COMP is to replace the parameter passed to it with the result.
22. Write a recursive routine PEVAL that uses Horner's rule to evaluate the polynomial

$$Y = A_0 + A_1X + \dots + A_NX^N$$

The coefficients A_0, \dots, A_N are to be in successive words in memory and all parameter addresses are to be passed via the stack.

23. Given that (SP) = 0100, (SS) = 0300, (PSW) = 0240, and the contents of the following

memory locations are as indicated

| | |
|-------|------|
| 00020 | 0040 |
| 00022 | 0100 |

and that the

INT 8

instruction has an offset of 00A0 within the segment whose address is 0900, determine the contents of SP, SS, IP, CS, PSW and the top three words on the stack after the interrupt sequence caused by the INT 8 instruction is executed.

24. Discuss the difference in the debug applications of the interrupts caused by the INT, INT Type, and single-step trap. In particular, when should one of these interrupts be used rather than the remaining two types?
25. If (PSW) = 0080 just after the INTO instruction whose offset is 0500 and segment address is B100 is executed, what will be the contents of the top of the stack just after the interrupt sequence is completed?
26. Give the definition of a macro DADD that adds two three-word memory operands and stores the result back in a three-word memory location. The dummy parameters are to be associated with the least significant words of the operands and the result. Also, give the expansions resulting from the following calls:

%DADD (OPR,PRICE[SI],TOTAL)

.

%DADD (TOTAL,TOTAL,TAX + 4)

27. Give the definition of a macro RESTORE that will pop the register contents that have been pushed in the order AX, BX, CX, DX, SI, and DI.
28. In the ABSOL example in Sec. 4-5-2, could the instruction

JGE %NEXT

be replaced by

JGE \$+4

and thereby eliminate the need for a local variable? Give your reasons

29. Write a macro LINE that performs the double-word computation

$Y \leftarrow AX + B$

by calling a macro DMULT for executing the multiply and a macro DADD for executing the addition.

30. Given the code:

```
%*DEFINE (ABSDIF (V1,V2,V3)) LOCAL CONT
(
    PUSH AX
%DIF (%V1,%V2)
    CMP AX,0
    JGE CONT
    NEG AX
```

```
%CONT:      MOV    %V3,AX
            POP    AX
}
%*DEFINE (DIF (X,Y))
(
    MOV    AX,%X
    SUB    AX,%Y
)
```

determine the expansions caused by the following calls and note any invalid calls:

- (a) %ABSDIF (P1,P2,DISTANCE)
- (b) %ABSDIF ([BX],[SI],X[DI],CX)
- (c) %ABSDIF ([BX][SI],X[BX][SI],240H)
- (d) %ABSDIF (AX,AX,AX)

31. Give a set of macro definitions for performing the logical operations OR, AND, and exclusive OR on double words whose locations are to be indicated by the first two dummy parameters. The result is to be put in the concatenation DX:AX. First do this problem without nesting macro definitions and then repeat the solution using definition nesting.

32. Write a code segment that causes

```
MOV    TERMINAL,0
```

to be assembled if the dummy parameter X = 'VT55' and otherwise results in

```
MOV    TERMINAL,1
```

being assembled.

33. Write a code segment that causes

```
ADD    AX,AX
```

to be assembled 10 times if the length of the string given the name X is greater than 5.

- 34. Define a macro that produces code for adding two binary N-byte operands and storing the N-byte result beginning at an arbitrary location. N is to be the name of a constant and is to appear as the fourth dummy parameter.
- 35. Define a macro for moving an arbitrary character string that ends with an EOT character from one string of bytes in memory to another.
- 36. From the following pseudocode draw a detailed flowchart which clearly indicates the elementary constructs:

```
CODE ← 0
IF A=0 THEN
    IF B=0 THEN
        CODE ← 1
    ELSE
        CALL DIVIDE to compute X1 ← -C/B
    ENDIF
ELSE
    CALL EVAL to compute E ← -B/2A
    CALL DISCRIM to compute D ← (B/2A)2 - C/A
    IF D<0 THEN
        CODE ← 2
    ENDIF
ENDIF
```

```

ELSE IF D=0 THEN
    X1 ← E
    CODE ← 3
ELSE
    CALL SQROOT to compute F ← √D
    CALL SUBTRACT to compute X1 ← E - F
    CALL ADDITION to compute X2 ← E + F
ENDIF
ENDIF

```

37. Consider the problem of using determinants to solve the equations

$$\begin{aligned} A_{11}X_1 + A_{12}X_2 &= B_1 \\ A_{21}X_1 + A_{22}X_2 &= B_2 \end{aligned}$$

where all quantities are to be in 16-digit, 10's complement, packed BCD format. Suppose that the input, solution, and output are performed by separate modules and the solution module uses a submodule DETEVAL to evaluate the determinants. Also, assume that the arithmetic operations and the unpacked/packed conversions are performed by separate modules. Draw a hierarchical diagram of the program, give module descriptions for the modules SOLUTION, DETEVAL, and MULTIPLY, give the pseudocode and a flowchart for the SOLUTION module, and last, provide the 8086 code for the SOLUTION module.

38. Describe how software interrupts could be used to perform the arithmetic operations in situations such as that indicated in Exercise 37.
39. Give the pseudocode for the DELETE module shown in Fig. 4-44. Translate this pseudocode into 8086 code.

5

Byte and String Manipulation

One very important computer application is *text processing*, which is the manipulation of sequences of bytes that contain the alphanumeric codes for characters, i.e., character strings. In the design of a text editor or text formatting program it is necessary to have program sequences for moving and comparing character strings, and for inserting strings into and deleting them from other strings. It is often necessary to search a string for a given substring or to replace a substring with a different substring. A text editor is a requirement on any general purpose computer system, and features which improve the ability of such a system to work with character strings can be a definite advantage. This chapter is concerned primarily with those features of the 8086 that facilitate the handling of character strings.

Although most of the instructions introduced in this chapter are designed for operating on character strings, they can be used for other purposes, such as searching arrays for keys in a file system. These instructions do not perform functions that could not be done by other instructions; it's just that they are designed to do certain things more efficiently, things that could potentially be very time consuming (e.g., searching a string or array containing thousands of bytes for a given byte, word, or substring).

It is also sometimes necessary to convert from one code (set of assignments for 0-1 combinations) to another. For example, a system may be connected to a terminal that sends and receives characters in the EBCDIC code even though the system is otherwise designed to work with the ASCII code. In that case, the characters being received from or transmitted to this terminal must undergo a code conversion. The 8086 solves the code conversion problem with the XLAT instruction.

The string-oriented instructions for the 8086 are defined in the first section of the chapter. The second section introduces the repeat prefix REP, which can

be used in conjunction with the string instructions to increase the speed of some string operations that would otherwise need to be programmed with loops. Section 5-3 gives a text editing example which utilizes the ideas introduced in the first two sections. Section 5-4 examines the XLAT instruction and its applications, and the last section discusses number format conversions.

5-1 STRING INSTRUCTIONS

The string instructions are summarized in Fig. 5-1. Because the string instructions can operate on only a single byte or word unless they are used with the REP prefix discussed in Sec. 5-2, they are often referred to as *string primitives*, or simply

Figure 5-1 String primitives.

| Name | Mnemonic and Format* | Description** |
|---------------------|----------------------|---|
| Move string | MOVS DST,SRC | ((DI)) ← ((SI)) Byte operands (SI) ← (SI) ± 1, (DI) ← (DI) ± 1 Word operands (SI) ← (SI) ± 2, (DI) ← (DI) ± 2 |
| Move byte string | MOVSB | |
| Move word string | MOVSW | |
| Compare string | CMPS SRC,DST | ((SI)) - ((DI)) Byte operands (SI) - (SI) ± 1, (DI) - (DI) ± 1 Word operands (SI) - (SI) ± 2, (DI) - (DI) ± 2 |
| Compare byte string | CMPSB | |
| Compare word string | CMPSW | |
| Scan string | SCAS DST | Byte operand (AL) - ((DI)), (DI) ← (DI) ± 1 Word operand (AX) - ((DI)), (DI) ← (DI) ± 2 |
| Scan byte string | SCASB | |
| Scan word string | SCASW | |
| Load string | LODS SRC | Byte operand (AL) ← ((SI)), (SI) ← (SI) ± 1 Word operand (AX) ← ((SI)), (SI) ← (SI) ± 2 |
| Load byte string | LODSB | |
| Load word string | LODSW | |
| Store string | STOS DST | Byte operand ((DI)) ← (AL), (DI) ← (DI) ± 1 Word operand ((DI)) ← (AX), (DI) ← (DI) ± 2 |
| Store byte string | STOSB | |
| Store word string | STOSW | |

*The B suffix indicates byte operands and the W suffix indicates word operands.

**Incrementing (+) is used if DF=0 and decrementing (-) is used if DF=1.

Flags: CMPS and SCAS affect all condition flags and
MOVS, LODS, and STOS affect no flags.

Addressing modes: Operands are implied.

primitives. All of the primitives are 1 byte long, with bit 0 indicating whether a byte (bit 0=0) or a word (bit 0=1) is being manipulated.

There are five basic primitives and each may appear in one of the following three forms:

Operation Operand(s)

or

OperationB

or

OperationW

If the first form is used, whether bytes or words are to be operated on is determined implicitly by the type of the operand(s). The second and third forms explicitly indicate byte and word operations, respectively.

Regardless of the form of the primitive the actual operands are determined solely by the contents of the SI, DI, DS, and ES registers. For a source operand the address of the operand is the sum of (SI) and (DS) \times 16₁₀, unless the DS register is overridden by explicitly specifying the ES, CS, or SS register as the segment register. The address of a destination operand is always the sum of (DI) and (ES) \times 16₁₀. This register indirect addressing scheme means that the offset(s) of the source and/or destination must be put in the SI and/or DI register(s) before the primitive is executed. It should be understood that for the MOVS and CMPS primitives, if the strings are in the same segment and DS is used for the source segment address, then DS and ES must contain the same segment address. The main reasons for including an operand(s) in a primitive is that the inclusion of the source (and destination) identifier(s) tends to make the program more readable and their presence allows the assembler to check the addressability of the operand(s). Using different segment registers for the source and destination allows the two string operands to reside in separate segments. However, to avoid switching segment registers a small program can define all string operands in a single segment. In particular, this is done when string operands are used for both the source and destination.

In addition to performing the indicated operation a primitive also causes (SI) and/or (DI) to be automatically incremented or decremented. This auto-indexing allows repetitive applications of a primitive to operate on successive bytes or words in a sequence. Whether auto-incrementing or auto-decrementing is used is dictated by the DF flag in the PSW. If DF=1, the index registers are auto-decremented, and if DF=0, they are auto-incremented. For instance, if DF=0, then

MOVSB

would cause the byte in (SI) + (DS) \times 16₁₀ to be moved to (DI) + (ES) \times 16₁₀ and the contents of both SI and DI to be incremented by 1. Therefore, if this primitive were in a loop and reexecuted, its next execution would cause the next byte in sequence to be moved.

When working with strings, the advantages of the MOVS and CMPS instructions over the MOV and CMP instructions are:

1. They are only 1 byte long.
2. Both operands are memory operands.
3. Their auto-indexing obviates the need for separate incrementing or decrementing instructions, thus decreasing overall processing time.

As an example consider the problem of moving the contents of a block of memory to another area in memory. A solution that uses only the MOV instruction, which cannot perform a memory-to-memory transfer, is shown in Fig. 5-2(a). A solution

```

    MOVE:      MOV    SI,OFFSET STRING1 ; USE SI AS SOURCE INDEX
               MOV    DI,OFFSET STRING2 ; USE DI AS DESTINATION INDEX
               MOV    CX,LENGTH STRING1 ; PUT LENGTH IN CX
               MOV    AL,[SI]           ;MOVE BYTE FROM SOURCE
               MOV    [DI],AL           ;TO DESTINATION
               INC    SI                ;INCREMENT SOURCE INDEX
               INC    DI                ;INCREMENT DESTINATION INDEX
               LOOP   MOVE

```

(a) Uses the MOV instruction


```

    MOVE:      MOV    SI,OFFSET STRING1 ; ASSUME (DS)=(ES)
               MOV    DI,OFFSET STRING2
               MOV    CX,LENGTH STRING1
               CLD
               MOVS   STRING2,STRING1 ;CLEAR DF FLAG
               MOVE
               LOOP

```

(b) Uses primitive MOVS

Figure 5-2 Program sequences for moving a block of data.

that employs the MOVS instruction is given in Fig. 5-2(b). Note that the second program sequence may move either bytes or words, depending on the type of STRING1 and STRING2. In Sec. 5-2 it will be seen that this task can be performed even more efficiently by applying the REP prefix to eliminate the explicit loop.

The program sequence given in Fig. 5-3 demonstrates the use of the DF flag by showing how data can be moved from an area to an overlapping area. In the

Figure 5-3 Moving a block of data between two overlapping areas.

```

    MOVE:      MOV    SI,SRCADDR          ;ASSUME (DS)=(ES)
               MOV    DI,DSTADDR
               MOV    CX,N              ;PUT LENGTH IN CX
               CLD
               CMP    SI,DI             ;CLEAR DF FLAG
               JA     MOVE
               STD
               ADD    SI,CX
               DEC    SI
               ADD    DI,CX
               DEC    DI
               MOVS  SB
               LOOP   MOVE
               .

```

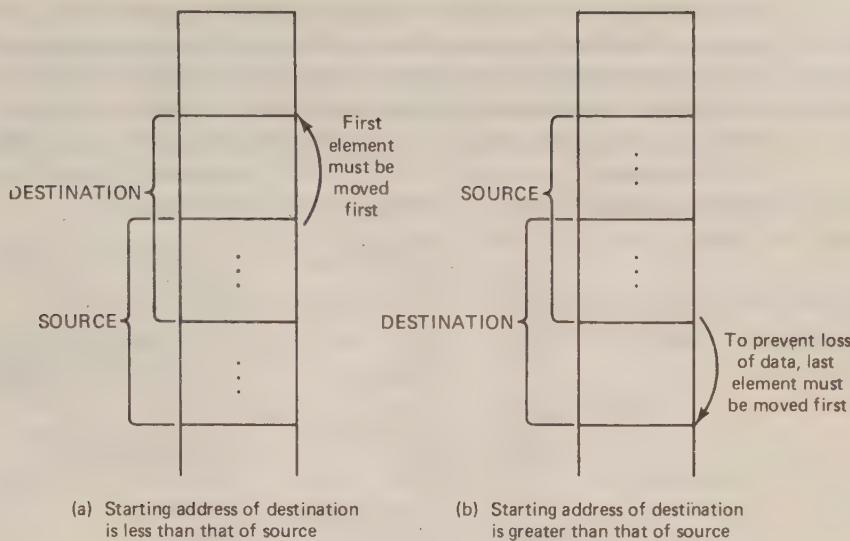


Figure 5-4 Possible cases when moving data between overlapping areas.

example, it is assumed that SRCADDR, DSTADDR, and N store the source address, destination address, and number of bytes to be moved, respectively. The two possible situations are for the source address to be greater than the destination address, in which case the first element in the source is to be moved first, or for the source address to be less than the destination address, which requires the last element of the source to be moved first. These situations are illustrated in Fig. 5-4.

The CMPS primitive can be used to compare strings of bytes or words of arbitrary length. The sequence in Fig. 5-5 causes a branch to SAME if the two strings are equal and otherwise continues at EXIT. As with the sequence for moving a string, this sequence can be significantly improved by applying the REP prefix discussed in the next section.

Note that when operands are included in the MOVS primitive the operand associated with SI appears last, but with CMPS it appears first. Also, MOVS does not affect the flags, while the purpose of CMPS is to set them so that subsequent decisions can be made (as in the program sequence in Fig. 5-5).

The other three string primitives, SCAS, LODS, and STOS, have single memory operands. Of these primitives, only SCAS affects the condition flags. A

```

MOV    SI,OFFSET STRG1
MOV    DI,OFFSET STRG2
MOV    CX,LENGTH STRG1
CLD
NEXT:  CMPS  STRG1,STRG2      ;COMPARE STRG1
       JNE   EXIT          ;WITH STRG2 AND
       LOOP  NEXT          ;BRANCH TO SAME IF
       JMP   NEAR PTR SAME ;THEY ARE EQUAL
EXIT:
       .
       .

```

Figure 5-5 String comparison.

program sequence that utilizes these primitives is given in Fig. 5-6. This sequence assumes that the string operands are defined in the same segment, which is pointed to by both DS and ES, and scans a string of 80 characters, which ends with a blank. If the string consists only of blank characters, a branch is taken to NOT_FOUND; otherwise, beginning with the first nonblank character, a substring of LINE of up to 31 characters is moved to SYMBOL. The substring may be terminated by a blank or after the first 31 characters have been moved.

```

MOV    DI,OFFSET LINE      ;PUT ADDR OF LINE IN DI
MOV    CX,80                 ;AND COUNT IN CX
MOV    AL,20H                 ;MOVE ASCII BLANK TO AL
CLD
NEXT:   SCAS    LINE          ;START FROM THE FIRST ELEMENT
       LOOPE   NEXT          ;CHECK FOR BLANK LINE
       JE     NOT FOUND      ;IF LINE IS BLANK BRANCH
       MOV    SI,DI          ;TO NOT FOUND ELSE MOVE STRING
       DEC    SI              ;PUT ADDR OF FIRST NONBLANK
       MOV    DI,OFFSET SYMBOL ;CHARACTER IN SI
       MOV    CX,31          ;AND ADDR OF SYMBOL IN DI
       FILL:   STOS    SYMBOL      ;PUT COUNT IN CX
               LOOP    FILL          ;FILL SYMBOL WITH BLANKS
               MOV    DI,OFFSET SYMBOL ;SETUP DI
               MOV    CX,31          ;AND CX
               JMP    SHORT SCANE
MOVE:   STOS    SYMBOL      ;MOVE STRING TO SYMBOL UNTIL A
SCANE:  LODS    LINE          ;BLANK IS FOUND OR 31 CHARACTERS
       CMP    AL,20H          ;HAVE BEEN MOVED
       LOOPNE MOVE
.
.
.

```

Figure 5-6 Example of the use of the SCAS, STOS, and LODS primitives.

5-2 REP PREFIX

Because string operations inherently involve looping, the 8086 machine language includes a prefix that considerably simplifies the use of string primitives with loops. This prefix has the machine code

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | Z |
|---|---|---|---|---|---|---|---|

where, for the CMPS and SCAS primitives, the Z bit helps control the loop. By prefixing MOVS, LODS, and STOS, which do not affect the flags, with the REP prefix 11110011, they are repeated the number of times indicated by the CX register according to the following steps:

1. If (CX) = 0, exit the REP operation.
2. Perform the specified primitive.
3. Decrement CX by 1.
4. Repeat steps 1 through 3.

For the CMPS and SCAS primitives, which do affect the flags, the prefix causes them to be repeated the number of times indicated by the CX register or until the

Z-bit does not match the ZF flag, whichever occurs first. The Z-bit/ZF flag comparison is made after each repetition. In any case the CX register is decremented with each repetition and, unless a Z-bit/ZF flag mismatch causes the termination, CX will be 0 when the loop is exited. Therefore, after repetitively executing the CMPS and SCAS primitives, the contents of ZF can be examined to determine the cause of termination.

In assembler language the prefix is invoked by placing the appropriate repeat (REP) mnemonic before the primitive. The REP mnemonics are defined in Fig. 5-7.

As an example of the use of the REP prefix let us reconsider the program sequence for moving a string within memory given in Fig. 5-2(b). By replacing the explicit loop

```
MOVE:    MOVS    STRING2,STRING1
        LOOP   MOVE
```

with

```
REP    MOVS    STRING2,STRING1
```

not only is the code simplified, but the execution time is reduced from

$$18 + 17 = 35 \text{ clock cycles per iteration}$$

to

$$9 + 17 = 26 \text{ clock cycles}$$

for the first iteration and 17 clock cycles for each subsequent iteration. Also, the last four instructions in the string comparison example in Fig. 5-5 could be changed from

```
NEXT:   CMPS    STRG1,STRG2
        JNE     EXIT
        LOOP   NEXT
        JMP    NEAR PTR SAME
```

Figure 5-7 REP prefix.

| Name | Mnemonic and Format | Termination Condition |
|---|---|-----------------------|
| Repeat string operation until CX = 0 | REP String Primitive* | (CX)=0 |
| Repeat string operation while equal or zero | REPE String Primitive** or REPZ | (CX)=0 or (ZF)=0 |
| Repeat string operation while not equal or not zero | REPNE String Primitive** or REPNZ | (CX)=0 or (ZF)=1 |

* MOVS, LODS, or STOS

**CMPS or SCAS

Note: In all cases, $(CX) \leftarrow (CX)-1$ with each repetition.

```

        MOV     BX, TABLE SIZE      ;LOAD NO. OF ENTRIES INTO BX
        LES     DI, TABLEPT        ;LOAD OFFSET AND SEGMENT ADDRESS
                                ;OF THE ARRAY TO BE SEARCHED
        MOV     DX, DI              ;SAVE OFFSET IN DX
        LDS     SI, SYMBOLPT       ;LOAD OFFSET AND SEGMENT ADDRESS
                                ;OF THE SYMBOL TO BE SEARCHED FOR
                                ;START FROM THE FIRST ENTRY
        CLD
LOOP1:   MOV     CX, 8             ;COMPARE TWO CORRESPONDING
        REPE   CMPSB             ;CHARACTERS UNTIL NO MATCH
        JE     FOUND             ;POINT TO THE NEXT TABLE ENTRY
        ADD    DX, 20
        MOV    DI, DX
        MOV    SI, OFFSET_SYMBOL ;POINT TO THE FIRST CHARACTER
        DEC    BX
        JNE    LOOP1
NOT_FOUND: .
.
.
FOUND:   .
.
.
```

Figure 5-8 Search a table for a given name with eight characters.

to

```

REPE   CMPS STRG1,STRG2
JNE   EXIT
JMP   NEAR PTR SAME
```

for a time savings from 43 to 22 clock cycles per iteration. This savings could be substantial if this sequence were used to search for a substring of a very long string and were, therefore, inside an outer loop.

Further examples which demonstrate the application of the string primitives and the REP prefix are given in Figs. 5-8 and 5-9. The program sequence in Fig. 5-8 searches an array TABLE consisting of 20-byte strings until the first 8 bytes of an array element are matched by the string beginning at SYMBOL. The offsets and segment addresses of TABLE and SYMBOL are contained in TABLEPT and SYMBOLPT, respectively. It is assumed that the first 8 bytes of an array element contain a symbol name and the remaining elements hold information that is related to the symbol. The sequence in Fig. 5-9 scans a string that begins at ASCII_STG and is not terminated by "\$", and replaces the dollar signs (\$) by underscores (_). In this example, it is assumed that both DS and ES contain the segment address associated with the byte array ASCII_STG.

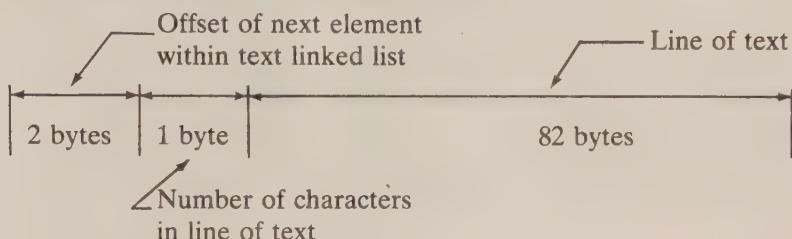
```

MOV     CX, LENGTH ASCII_STG ;INITIALIZE SEARCH
MOV     AL, '$'
MOV     DI, OFFSET ASCII_STG
CLD
LOOP1:  REPZ    SCAS ASCII_STG      ;SEARCH FOR '$' AND
        JNZ     DONE
        MOV     BYTE PTR [DI-1], '_'; REPLACE WITH '_'
        JMP     SHORT LOOP1
DONE:   .
.
```

Figure 5-9 Replace each "\$" in a character string with a "_".

5-3 TEXT EDITOR EXAMPLE

Let us consider a primitive text editor for creating, modifying, and storing up to 400 lines of text having up to 82 characters per line. The text is to be stored as a linked list with all elements having the form:



The first element in the linked list is to have zero characters and cannot be deleted. The last element is to point to the first element so that a “wraparound” effect will be provided. The last two characters in any line except the first line are to be a carriage return/line feed combination.

The hierarchical diagram of the program is given in Fig. 5-10. The EDITOR module is to use “?” as a prompting character to request a command. It is to call INPUT_COM to input the command and then execute one of the other modules, depending on the first letter of the command. At all times, one of the lines of text is to be considered the current line and the offset pointing to that line is called the *cursor*. The commands are to be:

R Filename—Reads the file *Filename* into the linked list (i.e., text buffer) in memory. The file is to have the linked list structure indicated above and the text buffer is to be a common area named TEXT_BUFF. After this command is executed, the cursor is to point to the first line of text.

/ String—Searches the text beginning with the current line until a substring that exactly matches *String* is found. The cursor is changed to point to this line and the line is printed. If *String* is not found, then the cursor is to return to its original value and NOT FOUND is to be printed by the subprogram MESSAGE.

I String—If the length of *String* is less than or equal to 80, this command is to cause *String*, concatenated with a carriage return/line feed combination, to be put in a line immediately following the current line. The cursor is to point to the new line and the new line is to be printed. If the length of *String* is greater than 80, the command is to be ignored.

D—If the current line is not the first element in the list, this command causes the current line to be deleted, the cursor to point to the next line, and the next line is to be printed; otherwise, the command is ignored.

P—Causes the current line to be printed unless the current line is the first element in the list, in which case the command is ignored.

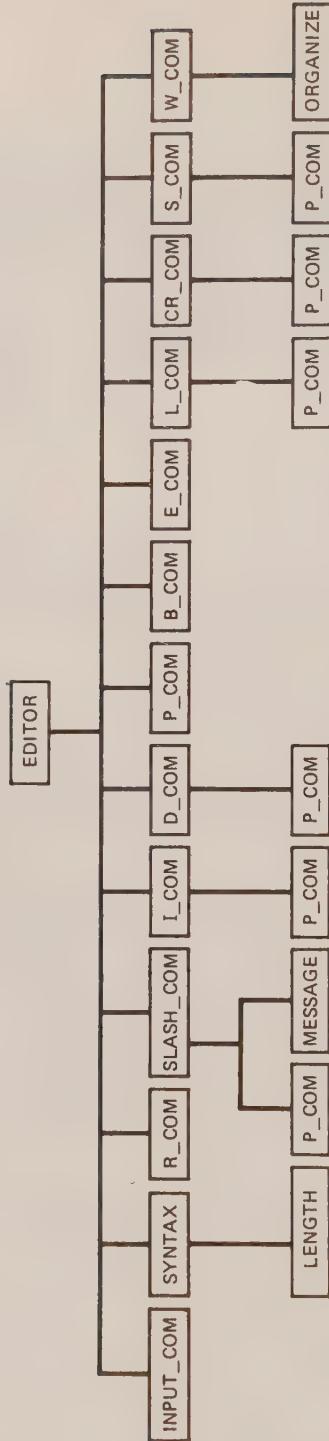


Figure 5.10 Hierarchical diagram for the text editor example.

- B**—Causes the cursor to point to the first element in the list.
- E**—Causes the cursor to point to the last element in the list.
- L**—Causes all lines to be printed and the cursor to point to the last line.
- (Carriage Return)**—Causes the cursor to move to the next line and the next line to be printed.
- S /String 1/String 2**—If the length of *String 2* minus the length of *String 1* plus the length of the current line is less than or equal to 82, this command causes the current line to be searched for *String 1* and, if it is found, to replace *String 1* by *String 2*. Also, the changed line is to be printed. If the resulting line is too long or *String 1* is not found, then no action is to be taken. In any case the cursor is to be left unchanged.
- W Filename**—Causes the linked list to be copied into an array so that the lines are in their proper order, and then outputs the ordered array to the file *Filenname*.
- Q**—Causes the program to terminate.

A blank is to appear after the command symbol in those commands that contain a filename or string. The SYNTAX module is to check the syntax of the commands. If the syntax is correct and the command symbol is I, /, or S, SYNTAX is to call the module LENGTH to determine the length(s) of the string(s) involved in the command. The ORGANIZE module is to copy the linked list into an ordered array before it is written into a file by W_COM. A flowchart of the EDITOR module is shown in Fig. 5-11.

Figure 5-12 gives the code for the S_COM module. It is assumed that this module is a separately assembled procedure that is passed the following five parameter addresses by putting the addresses in an array and putting the address of this array in the BX register:

- CURSOR**—Offset of current text element within the linked list.
- LENGTH_1**—Length of *String 1*.
- STRING_1**—Offset of first byte of *String 1*.
- LENGTH_2**—Length of *String 2*.
- STRING_2**—Offset of first byte of *String 2*.

It is assumed that the linked list of text is in a common area and the segment address of the common area is in ES when the procedure is entered. The segment address of the other parameters is to be in DS; therefore, the local variables in the procedure, which are in CODE_SEG, must be accessed using CS. The code for the EDITOR, SLASH_COM, I_COM, SYNTAX, and LENGTH modules is requested in Exercise 7.

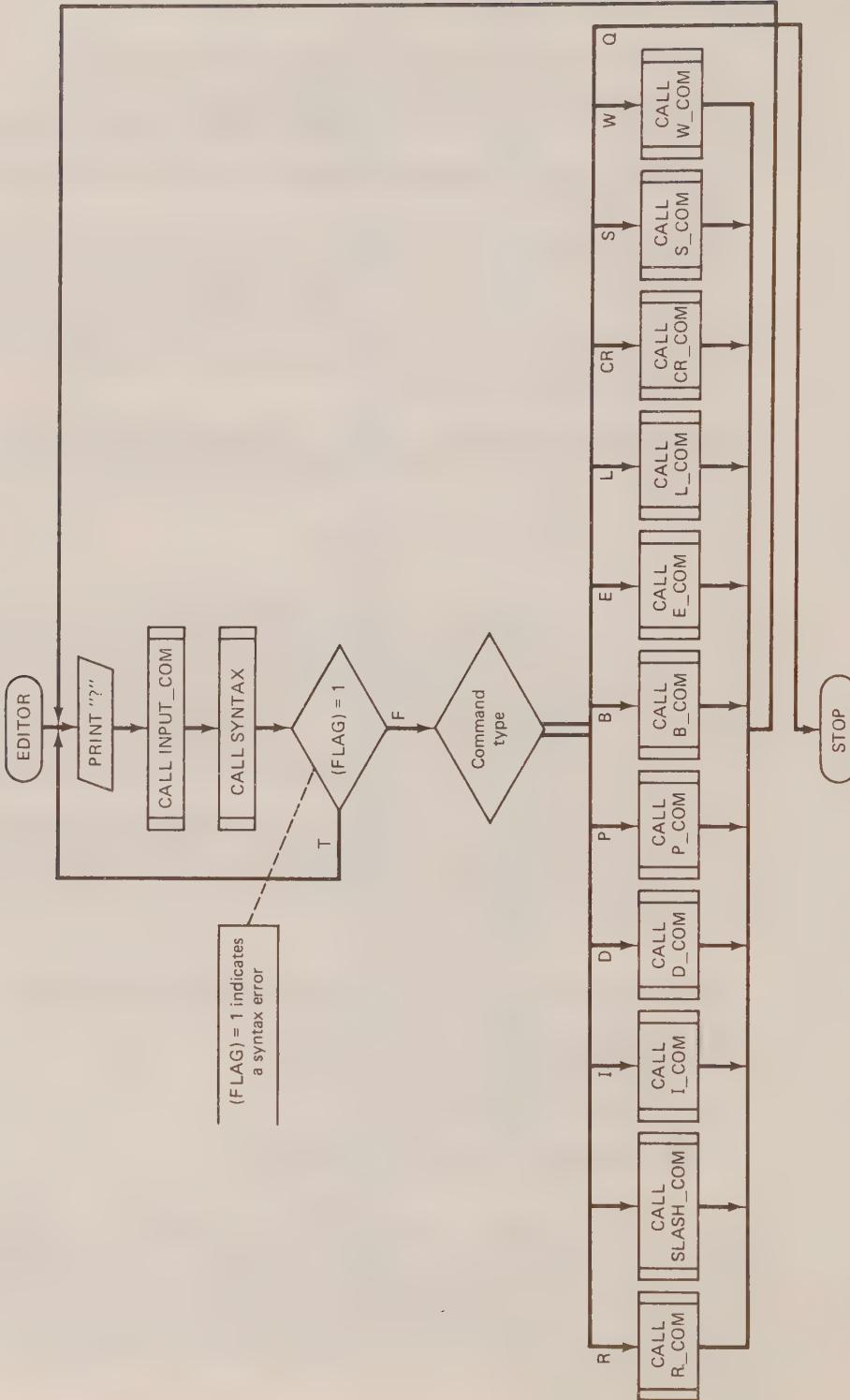


Figure 5-11 FlowChart of the EDITOR module.

Figure 5-12 Code for the module S__COM.

```

PUBLIC  S COM
EXTRN  PTCOM:FAR
CODE SEG  SEGMENT
ASSUME CS:CODE SEG
LENGTH2  DB    ?
ADLENG  DB    ?
S COM      PROC   FAR
        PUSH AX          ;SAVE THE REGISTERS
        PUSH CX
        PUSH DX
        PUSH SI
        PUSH DI
        MOV  SI,[BX]        ;PUT ADDRESS OF CURRENT
        MOV  DI,[SI]        ;TEXT ELEMENT IN DI
        MOV  DL,ES:[DI+2]   ;PUT LENGTH OF CURRENT LINE IN DL
        MOV  SI,[BX+2]
        MOV  DH,[SI]        ;PUT LENGTH OF
        MOV  SI,[BX+6]        ;STRING1 IN DH
        MOV  AL,[SI]        ;PUT LENGTH OF
        MOV  CS:LENGTH2,AL  ;STRING2 IN AL
        MOV  AL,DH          ;AND LENGTH2
        SUB  AL,DH          ;BRANCH TO EXIT IF LENGTH2
        ADD  AL,DL          ;MINUS LENGTH1 PLUS LENGTH
        CMP  AL,82          ;OF CURRENT LINE IS
        JNA  NEXT           ;GREATER THAN 82 (NOTE EXIT IS TOO
        JMP  EXIT            ;FAR FOR A CONDITIONAL BRANCH)
NEXT:   MOV  CS:ADLENG,AL  ;OTHERWISE, STORE ADJUSTED LENGTH
        MOV  SI,[BX+4]        ;PUT ADDR OF STRING1 IN SI
        ADD  DI,3             ;AND ADDR OF CURRENT LINE IN DI
        MOV  AX,DI
        XOR  CX,CX          ;AND AX
        CLD
        MOV  CL,DL          ;CLEAR CX
        MOV  CL,DH          ;AND DF FLAG
        SUB  CL,DH          ;PUT LENGTH OF CURRENT LINE
        ;MINUS LENGTH1 IN CL
SEARCH: PUSH CX          ;PUSH COUNT,
        PUSH DI          ;LINE POINTER,
        PUSH SI          ;AND STRING1 ADDR
        MOV  CL,DH          ;PUT LENGTH1 IN CL
        REPE CMPSB         ;COMPARE SUBSTRING WITH STRING1
        POP  SI
        POP  DI
        POP  CX
        JE   FOUND          ;LINE POINTER
        INC  DI
        LOOP SEARCH         ;AND STRING1 ADDR
        JMP  EXIT            ;BRANCH TO FOUND IF MATCH
        ;OTHERWISE, INCREMENT LINE POINTER
        ;DECREMENT CX, REPEAT IF NONZERO
        ;BRANCH TO EXIT IF NOT FOUND
        ;SAVE DI AND DS
FOUND:  PUSH DI
        PUSH DS
        MOV  SI,AX          ;PUT ADDR OF CURRENT LINE IN SI
        XOR  CX,CX          ;PUT NO. OF CHARACTERS
        MOV  CL,DL          ;TO BE SHIFTED INTO CX
        ADD  CX,AX
        SUB  CX,DI
        SUB  CL,DH
        MOV  AX,ES          ;MAKE DS SAME
        MOV  DS,AX          ;AS ES
        CMP  DH,CS:LENGTH2  ;COMPARE LENGTH1 AND LENGTH2
        JA   ABOVE           ;IF LENGTH1 IS BELOW OR
        STD
        XOR  AX,AX          ;EQUAL TO LENGTH2 SET DF AND
        MOV  DI,SI          ;SOURCE AND DESTINATION
        MOV  AL,DL          ;ADDRESSES TO SHIFT
        ADD  SI,AX          ;REMAINDER OF TEXT
        DEC  SI
        MOV  AL,ADLENG       ;LINE TO THE RIGHT
        ADD  DI,AX
        DEC  DI
        JMP  SHORT SHIFT     ;ELSE SET DF AND
        ;ADDRESSES FOR A
        CLD
        MOV  SI,DI          ;LEFT SHIFT
        XOR  AX,AX
        MOV  AL,DH
        ADD  SI,AX
        MOV  AL,CS:LENGTH2

```

Figure 5-12 (Continued).

```

        ADD    DI,AX
SHIFT:   REP    MOVS B
          POP    DS      ;SHIFT TEXT
          POP    DI      ;RESTORE DS AND DI
          MOV    CL,CS:LENGTH2 ;INSERT STRING2
          MOV    SI,[BX+8]
          CLD
          REP    MOVS B
          MOV    SI,[BX]    ;PUT ADDR OF CURSOR IN SI
          MOV    DI,[SI]    ;AND OF CURRENT ELEMENT IN DI
          MOV    AL,CS:ADLENG ;ADJUST LENGTH
          MOV    ES:[DI+2],AL ;OF CURRENT LINE
          CALL   FAR P_COM  ;PRINT NEW CURRENT LINE
EXIT:    POP    DI      ;RESTORE THE REGISTERS
          POP    SI
          POP    DX
          POP    CX
          POP    AX
          RET

S COM     ENDP
CODE SEG  ENDS
END

```

5-4 TABLE TRANSLATION

As mentioned in the introduction to this chapter, it is sometimes necessary to translate from one code to another. A terminal may communicate with the computer using the EBCDIC alphanumeric code even though the computer's software is designed to work with the ASCII code, or vice versa.

Code conversions involving fewer than 8 bits (which accommodates up to 256 distinct entities) can be performed most easily by storing the desired code in an array of up to 256 bytes and letting the original code be the index within the array of the desired code values. If the EBCDIC code were being converted to the ASCII code, then the EBCDIC code value for "A", which is 11000001, would be added to the address of the beginning of the array. Then, by putting the ASCII code for A, which is 01000001, in the array element having the address of the array plus 00C1, the code conversion is readily accomplished.

The 8086 has an instruction specifically designed for executing this procedure. It is the XLAT instruction and is defined in Fig. 5-13. This instruction may appear either as XLATB or XLAT OPR. Both forms produce the following 1-byte machine language instruction:

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|

In the XLAT OPR form, OPR is a dummy operand that is normally the variable name associated with the translation table and serves only to improve the readability of the program. The XLAT instruction assumes the base address of the byte array is in the BX register and the byte to be converted is in the AL register. The desired code value is taken from the array and put in the AL register. None of the flags are affected.

As an example, suppose that a string of unpacked BCD digits with zeros in the most significant 4 bits were to be converted to bit combinations to be used for lighting a seven-segment display. The arrangement of the seven segments in the display are shown in Fig. 5-14(a) along with the symbols for representing the

| Name | Mnemonic and Format | Description |
|-----------|---------------------|-------------------------------|
| Translate | XLAT OPR XLATB | (AL) \leftarrow ((BX)+(AL)) |

Flags: None of the flags are affected.

Addressing modes: The operands are implied by the op code.

Figure 5-13 XLAT instruction.

segments. Figure 5-14(b) summarizes the code conversion needed to form the various digits on the display. In the figure, bit 0 of the resulting code corresponds to segment a of the display, bit 1 to segment b, and so on. It is seen that a string such as

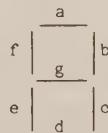
05 07 09 00 00 01 03 04

which corresponds to the number 57900134, would produce the output string

6D 07 6F 3F 3F 06 4F 66

Figure 5-15 gives a program sequence for translating the unpacked BCD string beginning at STG_BCD into the string to be output to the display which begins at

Figure 5-14 Unpacked BCD to seven-segment display conversion.



(a) Seven-segment display unit

| Decimal Digit | Display Form | Unpacked BCD Format | g f e d c b a | Code in Hex |
|---------------|--------------|---------------------|---------------|-------------|
| 0 | □ | 00000000 | 0 1 1 1 1 1 1 | 3F |
| 1 | : | 00000001 | 0 0 0 0 1 1 0 | 06 |
| 2 | 匚 | 00000010 | 1 0 1 1 0 1 1 | 5B |
| 3 | 匚 | 00000011 | 1 0 0 1 1 1 1 | 4F |
| 4 | 匚 | 00000100 | 1 1 0 0 1 1 0 | 66 |
| 5 | 匚 | 00000101 | 1 1 0 1 1 0 1 | 6D |
| 6 | 匚 | 00000110 | 1 1 1 1 1 0 1 | 7D |
| 7 | 匚 | 00000111 | 0 0 0 0 1 1 1 | 07 |
| 8 | 匚 | 00001000 | 1 1 1 1 1 1 1 | 7F |
| 9 | 匚 | 00001001 | 1 1 0 1 1 1 1 | 6F |

(b) Seven-segment code equivalent for BCD digits

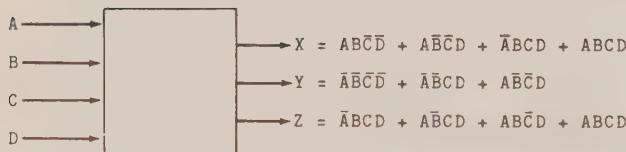
```

;SEVEN-SEGMENT CODE FOR DECIMAL DIGITS 0,1,2,3,4,5,6,7,8 AND 9
TABLE    DB 3FH
          DB 06H
          DB 5BH
          DB 4FH
          DB 66H
          DB 6DH
          DB 7DH
          DB 07H
          DB 7FH
          DB 6FH

.
.
.

LEA      BX, TABLE           ;LOAD BASE ADDR OF TRANSLATION TABLE
LEA      SI, STG_BCD         ;SET UP POINTER FOR SOURCE STRING
LEA      DI, STG_DISP         ;SET UP POINTER FOR DESTINATION
CLD
MOV     CX, LENGTH STG_BCD ;CAUSE OPERATIONS TO AUTO-INCREMENT
CONVERT: LODS   STG_BCD      ;MOVE A DIGIT TO AL
          CMP    AL, 9          ;IS IT GREATER THAN 9?
          JA     INVALID        ;IF TRUE, INVALID DIGIT IS DETECTED
          XLAT   TABLE          ;OTHERWISE, CONVERT TO 7-SEG CODE
          STOS   STG_DISP        ;STORE THE RESULT
          LOOP   CONVERT        ;REPEAT FOR NEXT DIGIT UNTIL DONE
.
.
.
```

Figure 5-15 Program sequence to convert unpacked BCD digits to seven-segment code.



(a) Logic network

Figure 5-16 Using code conversion to implement a multiple-output logic network.

| | XYZ |
|---------|---------|
| LOG_TAB | DB 010B |
| | DB 000B |
| | DB 000B |
| | DB 010B |
| | DB 000B |
| | DB 000B |
| | DB 000B |
| | DB 101B |
| | DB 000B |
| | DB 110B |
| | DB 000B |
| | DB 001B |
| | DB 100B |
| | DB 001B |
| | DB 000B |
| | DB 101B |

(b) Conversion table

STG_DISP. The sequence also includes a check for bit combinations that are not associated with a decimal digit and causes a branch to INVALID if such a combination is found.

Actually, code conversion can be viewed in a much more general light. Code conversion can be looked at as a functional mapping of the set of 8-bit 0-1 combinations into itself. The 8-bit combinations need not be alphanumeric code representations. As in the previous example, they may represent elements in any set containing fewer than 256 items. For instance, the bits being input to the conversion process could correspond to inputs to a logic network, and the resulting bits could correspond to the outputs of the network. The procedure could handle up to eight inputs and eight outputs. As an example, the array needed to provide the outputs for the logic network shown in Fig. 5-16(a) is given in Fig. 5-16(b). The conversion table assumes that the Boolean variable D corresponds to the least significant input bit and the Boolean function Z corresponds to the least significant output bit.

5-5 NUMBER FORMAT CONVERSIONS

Terminals, printers, card readers, and other devices communicate with a computer system through a code, often the ASCII code. However, the internal arithmetic operations are not normally done with numbers in their coded formats, but usually operate on numbers in their binary or packed BCD formats. Therefore, it is necessary to convert from the external form of the data which the I/O equipment is familiar with, to the internal form, which can easily be operated on by the machine language instructions of the computer.

Although the 8086 is capable of performing arithmetic on ASCII-coded numbers directly, a review of the examples in Chap. 3 reveals that anything more than simple arithmetic calculations on such numbers can be quite complicated. Arithmetic using the packed BCD format is acceptable if only addition and subtraction are involved. Binary arithmetic is, of course, the easiest and fastest of the three choices. Unfortunately, to use binary arithmetic all numbers must be converted from their ASCII form to their binary form. Therefore, there is a trade-off between taking the extra time to perform the conversions to and from binary, and using more time in carrying out the arithmetic operations. The packed BCD format is a good compromise for some problems since the conversion from ASCII to packed BCD is relatively simple and addition and subtraction on packed BCD numbers can be performed fairly quickly. The decision as to which format to use is most often arrived at after weighing the amount of I/O against the complexity of the arithmetic.

The entire process of inputting ASCII character strings, converting them to binary numbers, performing the necessary arithmetic, and reconverting the results to ASCII strings is shown in Fig. 5-17. The conversion to binary is a two-step process. First each ASCII character is converted into an unpacked BCD form with the four MSBs set to zeros, and then Horner's rule is applied to the string of

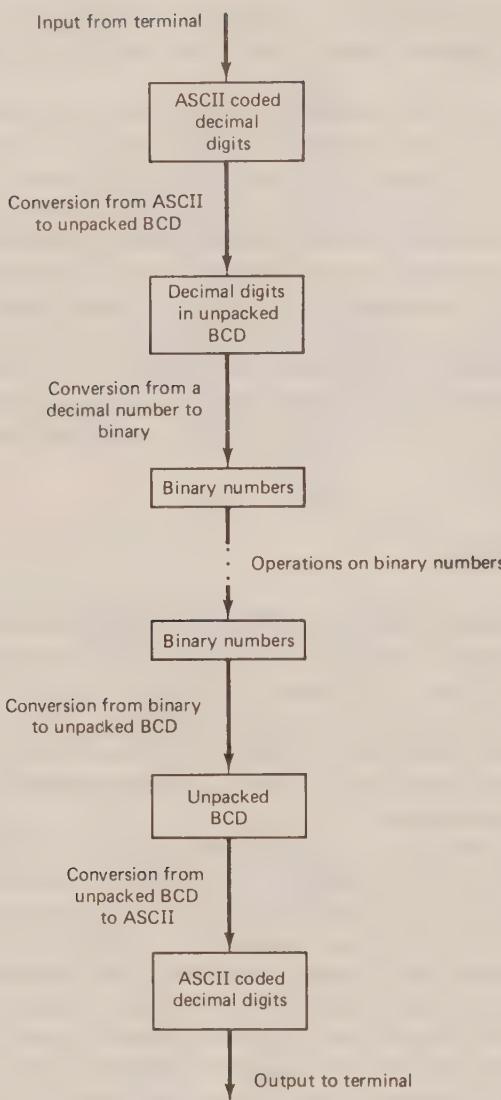


Figure 5-17 Conversion process required for inputting or outputting decimal numbers.

unpacked digits to produce the binary number. If the ASCII string is

$$d_n \dots d_1 d_0$$

where d_n, \dots, d_0 represent the decimal digits, then the binary number can be found by evaluating

$$((\dots ((10d_n + d_{n-1})10 + d_{n-2}) \dots)10 + d_1)10 + d_0$$

A program sequence for performing an ASCII to binary conversion is given in Fig. 5-18. The sequence assumes that five characters (possibly including leading

```

    MOV    CX,5          ;PUT NO. OF DIGITS IN CX
    MOV    DI,10         ;PUT BASE 10 IN DI
    MOV    BH,'9'        ;ASCII CODE 9
    MOV    BL,'0'        ;ASCII CODE 0
    MOV    SI,OFFSET ASCII_STG
    ADD    SI,LENGTH ASCII_STG-1
    STD
    XOR    AX,AX
    AGAIN: MUL   DI      ;MULTIPLY THE PARTIAL SUM BY 10
    JC    OVERFLOW       ;NUMBER IS GREATER THAN 65535
    MOV    DX,AX
    LODS  ASCII_STG     ;LOAD THE NEXT DECIMAL DIGIT
    AND   AL,7FH         ;CLEAR THE PARITY BIT
    CMP   AL,BL         ;IF IT IS LESS THAN 0
    JL    INVALID        ;AN INVALID DIGIT IS DETECTED
    CMP   AL,BH         ;IF IT IS GREATER THAN 9
    JG    INVALID        ;AN INVALID DIGIT IS DETECTED
    SUB   AL,30H         ;CONVERT TO UNPACKED BCD
    CBW
    ADD   AX,DX
    JC    OVERFLOW       ;NUMBER IS GREATER THAN 65535
    LOOP  AGAIN
    .
    .

```

Figure 5-18 Conversion of an ASCII-coded decimal number to its binary equivalent.

0s) are stored beginning with ASCII_STG, with the least significant digit appearing first. Each ASCII character is validated and a branch to INVALID is made if a nondigit character is encountered. Also, because only one word has been reserved for the result, a branch to OVERFLOW is executed if the result is greater than $2^{16} - 1$. A sequence for performing the reverse conversion is given in Fig. 5-19. ASCII zeros (hexadecimal 30) are used to pad the more significant bytes if the result is less than five digits long. These sequences do not take negative numbers into account—the extension to negative numbers is requested in Exercise 15.

Occasionally, it is necessary to input or output an ASCII string that represents a hexadecimal number. This is the case for a debugging program which is designed to modify the machine instructions of another program. The debugging program must be capable of inputting an ASCII string that represents a hexadecimal address, machine instruction, or datum and converting it to its binary form so that it can be used to directly modify the machine code or memory contents in the program

Figure 5-19 Conversion of a binary number to an ASCII character string.

```

    MOV    CX,5          ;USE DI AS INDEX REGISTER
    XOR    DI,DI
    AGAIN: MOV   AX,BINNUM
    MOV   BX,10
    XOR   DX,DX          ;ZERO DX
    DIV   BX              ;DIVIDE DX:AX BY 10
    ADD   DL,30H           ;CONVERT THE REMAINDER TO ASCII
    MOV   ASCII_BCD[DI],DL ;STORE CONVERTED DECIMAL DIGIT
    INC   DI              ;POINT TO THE NEXT LOCATION
    LOOP  AGAIN
    .
    .

```

that is being debugged. For example, the hexadecimal numbers

01A25 1B3D

might be the response to a query for changing the contents of an address (the first number) to a specified value (the second number). The debug program would need to convert the ASCII strings '01A25' and '1B3D', which are input from the terminal, to their equivalent hexadecimal numbers. Similarly, ASCII strings for hexadecimal addresses, instructions, and memory contents may need to be output.

The procedure for inputting hexadecimal numbers consists of two steps. The first step converts the ASCII hexadecimal digits to their binary equivalents and puts them into bytes which are padded on the left with 0s, and the second step simply packs these binary equivalents together. Because the binary equivalent of a hexadecimal number is obtained by expressing each hexadecimal digit by its binary equivalent, the result of the input conversion process is the binary equivalent of the input hexadecimal number. The output process involves the same two steps, but they are reversed.

The program sequence in Fig. 5-20 converts a four-digit hexadecimal number in ASCII code to its binary equivalent. It is assumed that the hexadecimal number is stored beginning at ASCII_STG, with the least significant digit appearing first. The sequence compares each digit with the ASCII codes for 0 through 9 and A through F to determine whether it is a valid hexadecimal digit. If so, it is converted to its binary equivalent and then combined with the other digits to form the binary equivalent number. A sequence for performing the reverse process is requested in Exercise 16.

Figure 5-20 Conversion of a four-digit ASCII-coded hexadecimal number to its binary equivalent.

```

MOV CL,4 ;USE CL AS SHIFT COUNTER
MOV CH,CL ;USE CH AS DIGIT COUNTER
MOV SI,OFFSET ASCII_STG;USE SI AS POINTER
CLD ;CAUSE STRING OPERATIONS TO AUTOINC
XOR AX,AX ;CLEAR AX
XOR DX,DX ;CLEAR DX
AGAIN: LODS ASCII_STG ;LOAD AN ASCII-CODED DIGIT
AND AL,7FH ;CLEAR THE PARITY BIT
CMP AL,'0' ;IS IT < 30H?
JL INVALID
CMP AL,'9' ;IS IT > 39H?
JG A_TO_F
SUB AL,30H ;CONVERT TO DIGIT 0-9
JMP SHORT_ROTATE
A_TO_F: CMP AL,'A' ;IS IT < 41H?
JL INVALID
CMP AL,'F' ;IS IT > 46H?
JG INVALID
SUB AL,37H ;CONVERT TO DIGIT A-F
OR DL,AL ;COMBINE WITH PREVIOUS DIGITS
ROR DX,CL ;TO FORM BINARY EQUIVALENT
DEC CH
ROTATE: JNZ AGAIN ;IF NOT THE LAST DIGIT, REPEAT
MOV INTER,DX ;STORE THE BINARY EQUIVALENT

```

EXERCISES

1. Assume a 5-MHz clock and estimate the time needed to execute the sequence in Fig. 5-2(a) if the length of STRING1 were 10; in Fig. 5-2(b) if the REP prefix were used.
2. Rewrite the sequence in Fig. 5-5 using the CMP instruction instead of the CMPS instruction.
3. Write a program sequence that will search a character string STRING and move a substring of digits to NUM, but does not use the REP prefix. If a nonblank character that is not a digit occurs, a branch is to be taken to MESSG1. STRING is to be searched until a digit is found and the digits are to be moved to NUM until a blank is found or the total number of characters exceeds 20, at which time the sequence is to continue at EXIT.
4. Give a program sequence that compares the first 10 bytes beginning at CHAR_1 with the first 10 bytes beginning at CHAR_2 and branches to MATCH if they are equal, but otherwise continues in sequence.
5. Use the REP prefix and rewrite the solution to Exercise 3.
6. Give the machine language code for
 - (a) AGAIN: MOVS STRING1,STRING2
LOOP AGAIN
 - (b) REP MOVS STRING1,STRING2where STRING1 and STRING2 are beginning addresses of word arrays.
7. Consider the text editor example discussed in Sec. 5-3 and assume that:

TEXT_BUFF—The common area used to store the linked list of text. This area is to be defined as a structure with each element corresponding to a line of text.

CURSOR—The offset of the current line within TEXT_BUFF.

FLAG—Used to indicate a syntax error. It is set to 1 if an error has been detected by SYNTAX.

FILENAME—The location used to store the *Filename* appearing in an R or W command.

LENGTH_1—Used to store the length of the *String* in a / or I command and *String 1* in an S command.

STRING_1—The offset of the first byte of *String* in a / or I command and *String 1* in an S command.

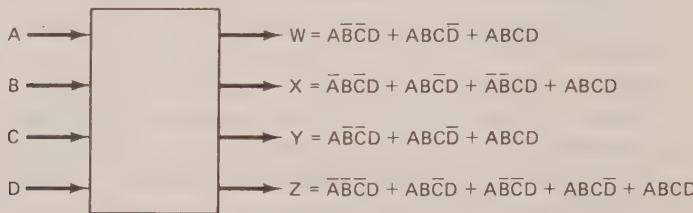
LENGTH_2—Used to store the length of *String 2* in an S command.

STRING_2—The offset of the first byte of *String 2* in an S command.

Write the code for each of the following program modules assuming that they are separately assembled:

- (a) EDITOR (b) SLASH_COM (c) I_COM
- (d) SYNTAX and LENGTH are assembled together but LENGTH is in a separate segment and is called as a procedure.

8. Expand the table in Fig. 5-14 to one that will convert hexadecimal digits to seven-segment display code. The digits A, C, E, and F are to be displayed as uppercase letters and B and D as lowercase letters (b and d).
9. Assume the following logic network:



Determine the contents of an array IN_OUT that will produce the correct outputs for all possible inputs. Suppose that 10 consecutive inputs are stored in INPUT through INPUT+9 and write a program sequence that will put the corresponding outputs in OUTPUT through OUTPUT+9.

10. Rewrite the solution to Exercise 3-30 using string primitives.
11. Rewrite the solution to Exercise 4-35 using string primitives.
12. Write a program segment that will examine an ASCII character string of 100 characters and replace each decimal digit by a %. Assume that the character string starts at STG.
13. Write a program segment that will copy all ASCII characters in STG that are enclosed by a pair of parentheses to the string MESSAG and store the number of characters moved in COUNT.
14. Assume that a symbol table starting at location TABLE consists of 100 entries; each entry has 80 bytes with the first 8 bytes representing the name field and the remaining 72 bytes representing the information field. Write a program sequence to search this table for a given name of eight characters stored in NAME. If this name is found, copy the associated information to INFO; otherwise, fill INFO with null characters.
15. Modify the sequences in Figs. 5-18 and 5-19 so that they will assume leading blanks instead of leading zeros. Further modify Fig. 5-18 so that a leading minus sign will cause the 2's complement of the number indicated by the string of digits to be stored. Keep in mind that the numbers will now be limited to -32768 to 32767 and that the input validation must be changed accordingly.
16. Write a sequence that will convert the integer in BINARY to four ASCII-coded hexadecimal digits and put the result in ASCII_STG.

6

I/O Programming

In the previous chapters it has been assumed that the programs and data being operated on by the computer were already in memory and no consideration was given to how they got there. All examples included only the movement of information between the CPU and memory. The purpose of this chapter is to discuss the ways in which information can be moved between peripheral or mass storage devices and the CPU or memory.

Figure 6-1 shows the basic architecture of a single-bus computer system. Because multiple-bus systems are more complicated and their architecture may vary considerably, this chapter will assume only a single bus, leaving the more complicated configurations to Chap. 11. As seen from the figure, all peripheral and mass storage devices are connected to the system bus through interfaces. Each interface contains a set of registers, called *I/O ports*, through which the CPU and memory communicate with the interface's external device. Some of the ports are for buffering data to and from the CPU and memory, some are for holding status information about the device and interface so that it can be examined by the CPU, and some are for retaining the commands sent from the CPU to control the actions taken by the interface and device. All communication with the external world and mass storage is channeled through the I/O ports in the interface. Therefore, the CPU must have a means of transferring information to and from these ports as well as memory.

Some computers encompass both the memory and port addresses in a single address space and allow all instructions that are capable of accessing memory to access the I/O ports. Others, such as the 8086-based systems, permit the establishment of two address spaces, an I/O space and a memory space. The latter is done by including control lines in the control bus that indicate whether the address

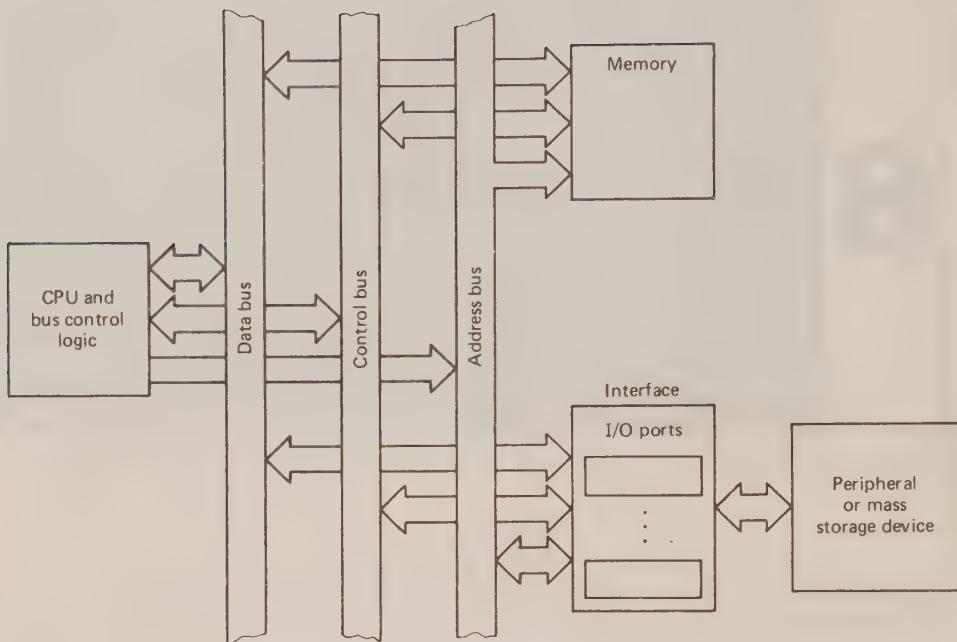


Figure 6-1 Single-bus system.

on the address bus is in the I/O space or the memory space. In order to send the correct signals over the control lines, a system that has separate I/O and memory address spaces must have different instructions for communicating with the I/O ports.

An output from the CPU to a control or buffer port is made by putting the address on the address bus and the proper signals on the control bus, and then putting the data on the data bus. An input from an input port is accomplished by putting the address and control signals on their respective buses and waiting for the interface to respond by placing the contents of the addressed port on the data bus. It should be emphasized that the addresses are associated with the ports, not the interfaces. If an interface has four ports, it must be designed to accept four addresses. (However, it is permissible to design an interface so that two or more registers share the same port, provided that the design is such that the interface can properly direct its internal traffic.)

This chapter begins with a discussion of the basic issues relating to I/O transfers and I/O programming. Among other things, it introduces the principal types of I/O:

1. Programmed I/O.
2. Interrupt I/O.
3. Block transfers.

Although the first section gives only a brief introduction to several important topics, the next three sections in the chapter, which are primarily to give a detailed treatment of the I/O types, also reexamine these topics. The last section provides a complete I/O programming design example.

6-1 FUNDAMENTAL I/O CONSIDERATIONS

The transfer of data to or from a port can be done in two ways. One way is to execute an instruction that causes a single byte or word to be transferred and the other is to execute a sequence of instructions that causes a special system component associated with the interface to transfer a sequence of bytes or words to or from a predesignated block of memory locations. The latter is referred to as a *block transfer* or a *direct memory access (DMA)* and the special component is called a *DMA controller*. Byte or word transfers are between the port and the CPU, but block transfers are made directly with memory.

Of the three principal types of I/O, programmed I/O and interrupt I/O are based on byte or word transfers and, if the data are to be moved between memory and a port, they must be transferred via a CPU register. For instance, if a word is to be input from a port to a memory location it must first be input to a CPU register and then moved to the memory location. To transfer a stream of bytes or words being sent to memory from an I/O or mass storage device, the program would need to successively, one by one, input the bytes or words to the CPU and then put them in consecutive memory locations. This could be done with a program loop. A block input transfer is supervised by the interface and DMA controller and is such that each byte or word is moved directly from the port to the memory as soon as it arrives at the port. The program need only give the required commands to the interface and DMA controller to initiate the transfer. Similarly, the interface and its DMA controller can take consecutive bytes or words from memory and send them to an external device.

Two questions that arise are: How does the system know when an interface has data available or is ready to receive data; and if more than one interface simultaneously needs attention, which should get the attention first? In the case of programmed I/O the program determines which interfaces need servicing by testing the "ready" bits in their status registers. Programmed testing of ready bits or signals is known as *polling*. For interrupt I/O, an external interrupt is sent to the CPU from the interface when the interface has data to input or is ready to accept data, and the I/O operation is performed by an *interrupt routine*. For a DMA operation the interface requests the use of the bus by sending a signal over a control line and makes the necessary transfer without the help of the CPU or software being executed. Exactly how the three types of I/O are performed is discussed in detail in Secs. 6-2, 6-3, and 6-4.

If two or more interfaces request service simultaneously, the order in which the requests are satisfied depends on the *priority* arrangement built into the system. There are a variety of ways the priorities of the interfaces can be designated; some

are based on software, some on hardware, and some on a combination of the two. Although the priority arrangement may differ according to the type of I/O, there must always be an unambiguous method for resolving conflicts. Because priority is often determined by hardware, it is not only considered in each of the remaining sections in this chapter, but is also discussed in Chap. 8.

On the 8086, all programmed communication with the I/O ports is done by the IN and OUT instructions defined in Fig. 6-2. Both instructions may transfer either a byte or a word and both have a long form and a short form. The first operand in the IN instruction, the destination operand, must be either AL (for a byte transfer) or AX (for a word transfer). As with memory, a word transfer is made from two consecutive addresses with the low-order byte in AX being moved from the port with the lower address. If the second operand in the IN instruction evaluates to a constant, then the constant is used as the address of the port whose contents are being input. In this case the instruction is 2 bytes long with the port address occupying the second byte. If the second operand is DX, then there is only one byte in the instruction and the contents of DX are used as the port address. Unlike memory addressing, the contents of DX are not modified by any segment register. This form allows variable access to I/O ports in the range 0 to 64K. The machine language code for the IN instruction is:

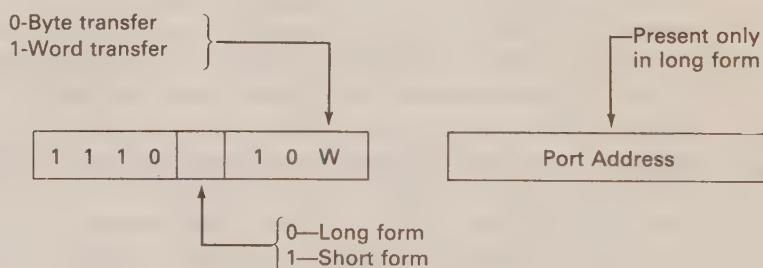


Figure 6-2 IN and OUT instructions.

| Name | Mnemonic and Format | Description |
|------------------|---------------------|---------------------------------|
| Input | | |
| Long form, byte | IN AL,PORT | (AL) \leftarrow (PORT) |
| Long form, word | IN AX,PORT | (AX) \leftarrow (PORT+1:PORT) |
| Short form, byte | IN AL,DX | (AL) \leftarrow ((DX)) |
| Short form, word | IN AX,DX | (AX) \leftarrow ((DX)+1:(DX)) |
| Output | | |
| Long form, byte | OUT PORT,AL | (PORT) \leftarrow (AL) |
| Long form, word | OUT PORT,AX | (PORT+1:PORT) \leftarrow (AX) |
| Short form, byte | OUT DX,AL | ((DX)) \leftarrow (AL) |
| Short form, word | OUT DX,AX | ((DX)+1:(DX)) \leftarrow (AX) |

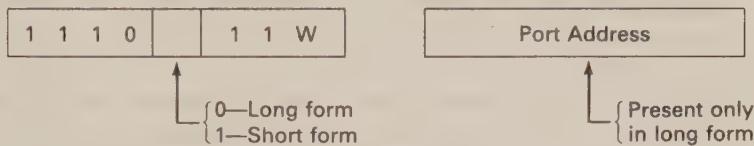
Note: PORT is a constant ranging from 0 to 255.

Flags: No flags are affected.

Addressing modes: Operands are limited as indicated above.

Although AL or AX is implied as the first operand in an IN instruction, either AL or AX must be specified so that the assembler can determine the W-bit.

Similar comments apply to the OUT instruction except that for it the port address is the destination, and is therefore indicated by the first operand, and the second operand is either AL or AX. Its machine code is:



Note that if the long form of the IN or OUT instruction is used the port address must be in the range 0000 to 00FF, but for the short form it can be any address in the range 0000 to FFFF (i.e., any address in the I/O address space). Neither IN nor OUT affects the flags.

The IN instruction may be used to input data from a data buffer register or the status from a status register. The instructions

```
IN      AX,28H
MOV    DATA_WORD,AX
```

would move the word in the ports whose addresses are 0028 and 0029 to the memory location DATA_WORD. The instructions

```
IN      AL,27H
TEST   AL,00000100B
JNZ    ERROR
```

would input the byte from port 27H and branch to ERROR if bit 2 of this byte is set.

Likewise, the OUT instruction is intended for outputting data or sending a command to a specified I/O port. If bit 7 of the port 26H causes a block transfer to begin through the associated interface, then the transfer could be initiated by the sequence:

```
COM_REG EQU 26H
BEGIN_BIT EQU 10000000B
.
.
.
MOV      AL,OTRCNTBITS
OR       AL,BEGIN_BIT
OUT      COM_REG,AL
```

where OTRCNTBITS contains other bits in the command to be sent to the command register.

Sometimes it is sufficient to bring in only one byte or word and use or operate on it by itself, but most of the time a sequence of bytes or words is needed before the data become meaningful. If several bytes or words are required, then it is

necessary to transfer them to a set of consecutive locations in memory before they are used. A set of locations used for such a purpose is known as a *memory buffer*, or simply *buffer*. For example, the characters in a typed line may represent a command to the operating system. Because the command could not be interpreted until the entire line has been input, the string of characters being input would need to be stored in a memory buffer until a carriage return/line feed combination is encountered.

Also, it is often necessary to temporarily store input data until it can be converted into a suitable form or other operations can be performed on it. Some operations may employ more than one memory buffer so that one buffer can be input to while the other buffer is being operated on; then, when the buffer being filled is full, it can be operated on and the input can be directed to the other buffer. This mode of operation is referred to as *double buffering*. Some situations may require more than two buffers and the term *multiple buffering* is used to describe them. Although this discussion of buffering has been centered on inputting, similar comments apply to outputting. Triple buffering is required when inputting, processing, and outputting must be carried on simultaneously.

The conversion of numerical data from its input or output format to the format used by the computer is a very important part of an I/O process. Simple conversions were considered in Chap. 5, but some conversions can be much more complex. A string of characters may need to be converted to a floating point format or a line of characters may be partitioned and a conversion made for each partition. Normally, a high-level language I/O statement will set up the pattern that is to be used in such a partitioning process. The pair of FORTRAN statements

```
      WRITE (6,20) X,N  
20      FORMAT (T2,F10.2,I5)
```

would cause the variable X, whose contents are in floating point format, and the variable N, whose contents are in integer format, to be converted to the correct character strings and output to a line of print. The pattern of characters is determined by the FORMAT statement. The first 10 characters would be formed according to the contents of X and the next 5 according to N.

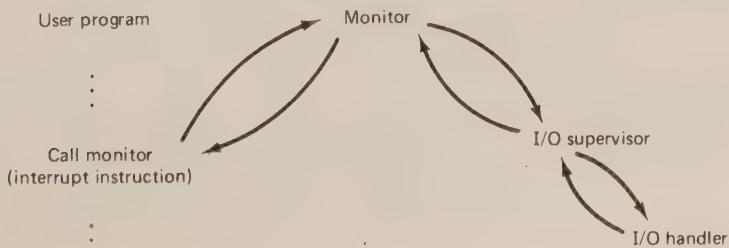
Another basic consideration with regard to I/O is how the data are passed between the procedures. Normally, the procedures that perform the I/O are separate from those that operate on the data. There are two basic situations. A simple microcomputer, one used for a low-end application, may be designed to execute only one program. When this is true the programmer can put the components of the program, including the data areas, in fixed areas of memory and assume complete control over the linking process. Thus, the programmer can ascertain that each procedure knows exactly where all other components in the program are located in memory. This means that an input procedure will place the input data in an area that is directly accessible by the other procedures in the program and an output procedure will have a fixed area to go to when retrieving the data to be output. Programs and data areas are not dynamically moved around within memory. It is this static situation that will be assumed in this chapter.

At the other extreme is a system that is directed by a complex operating system that is capable of controlling a multiprogramming environment. Under such circumstances, the I/O is done by a set of I/O procedures, called *I/O drivers* or *I/O handlers*, and interrupt routines. Together these drivers and interrupt routines comprise the I/O subsystem of the operating system. Unlike the simple situation described above, a multiprogramming environment may require that programs and data areas be continually moved about. Because the memory may not be large enough to hold all of the programs at once, user programs are shuffled between memory and mass storage devices and I/O drivers are brought in and executed as they are needed. As one might expect, the communication between the operating system, interrupt routines, I/O drivers, and user programs can be quite complicated.

In a typical system, the input and output transfers required by a user's program are actually performed by the operating system. This arrangement provides two obvious advantages. First, it releases the user from the details of I/O processing. Because I/O routines are device dependent, a user might not have a thorough enough knowledge of the devices to design the necessary routines. By including them in the operating system, users can share these routines and thereby reduce the amount of duplicated effort. Second, system resources are better utilized. If I/O processing is handled through the operating system, the monitor is informed of any I/O request by the user's program. The monitor can then initiate the I/O and let the CPU execute another job while the current one is waiting for the I/O to be completed.

Figure 6-3 shows a possible sequence of events when I/O is handled by the operating system. In a user's program, whenever I/O is needed the user calls the monitor through a type of software interrupt, or trap, which passes the control to the monitor along with a function code, indicating the I/O operation to be performed, and the necessary parameters required to carry out that operation. By examining the function code, the monitor, which handles other services in addition to I/O, knows the request is for an I/O operation and, consequently, dispatches the task to the I/O supervisor. The I/O supervisor further determines which device is involved in the I/O operation so that the corresponding I/O driver, which is typically an interrupt-driven or DMA routine, can be initiated. The I/O handler uses a buffer in the system area for temporarily storing input and output data. The I/O supervisor moves the data to be output (or the input data) from the user's area

Figure 6-3 I/O handling by the operating system.

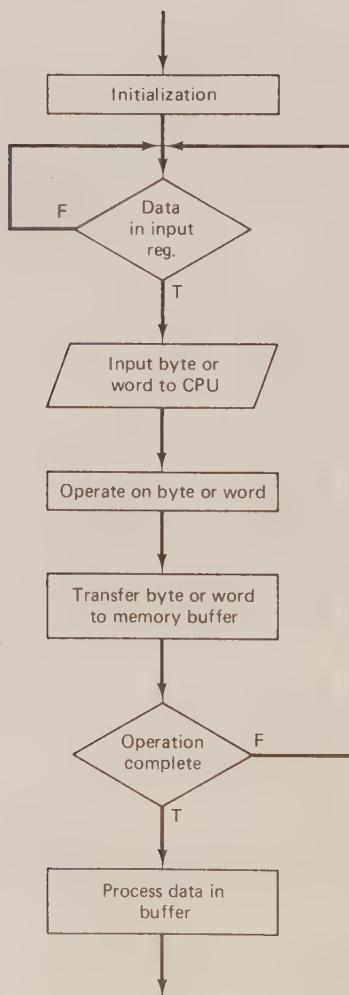


to the buffer area (or vice versa). Upon completion of the I/O operation, control is transferred back to the user's program through the I/O supervisor and the monitor.

6-2 PROGRAMMED I/O

Programmed I/O consists of continually examining the status of an interface and performing an I/O operation with the interface when its status indicates that it has data to be input or its data-out buffer register is ready to receive data from the CPU. A typical programmed input operation is flowcharted in Fig. 6-4. The flowchart assumes that a sequence of bytes or words is to be input and as each byte or

Figure 6-4 Programmed input.

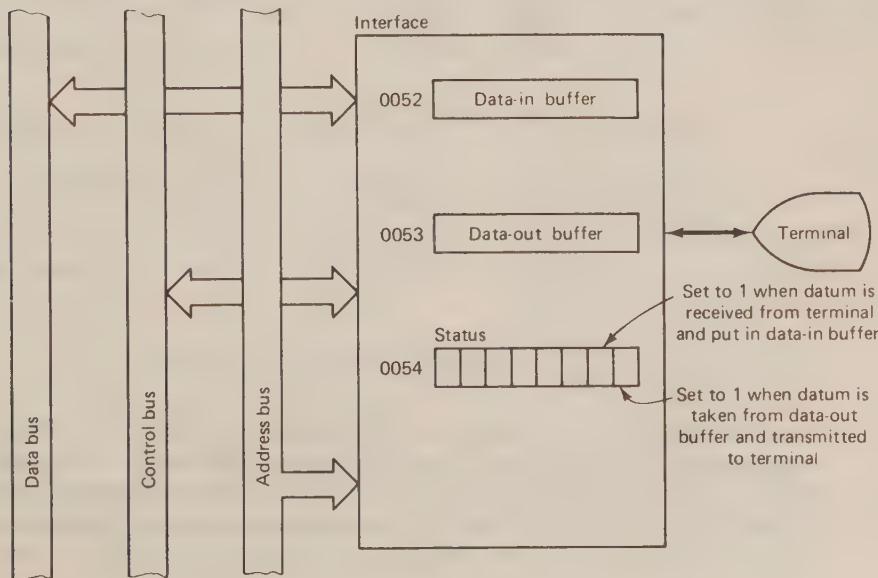


word is brought into the CPU it is modified and transferred to a memory buffer. After all of the data have been brought in and put in the buffer, the buffer is processed.

As a more complete example, suppose that a line of characters is to be input from a terminal to an 82-byte array beginning at BUFFER until a carriage return is encountered or more than 80 characters are input. If a carriage return is not found in the first 81 characters then the message "BUFFER OVERFLOW" is to be output to the terminal; otherwise, a line feed is to be automatically appended to the carriage return. Because the ASCII code is a 7-bit code, the eighth bit, bit 7, is often used as a parity bit during the transmission from the terminal. Let us assume that bit 7 is set according to even parity and if an odd parity byte is detected, a branch is to be made to ERROR. If there is no parity error, bit 7 is to be cleared before the byte is transferred to the memory buffer. The terminal's interface is to be as shown in Fig. 6-5, with the address of the data-in buffer register being 0052, of the data-out buffer being 0053, and of the status register being 0054. A 1 in bit 1 of the status register indicates that the data-in buffer register contains a byte to be input and a 1 in bit 0 indicates that the data-out buffer register is empty. The interface is designed so that bit 1 is set when a byte is input from the device and cleared when the byte is input from the interface. Conversely, bit 0 is cleared when a byte is output to the interface and set when it is output to the device.

The program for solving the above problem is given in Fig. 6-6. The registers ES and DI are used as the segment and index registers for storing the characters as they are input and CX is used as the counter to limit the looping process. The first four executable instructions that are shown initialize the DS and ES registers

Figure 6-5 Interface for the programmed I/O example.



```

DATA SEG      SEGMENT
MESSAGE      DB      'BUFFER OVERFLOW',ODH,0AH
.
.
.
DATA SEG      ENDS
COM_SEG      SEGMENT COMMON
    -BUFFER     DB      82 DUP(?)
    COUNT       DW      ?
;RESERVE BUFFER AREA
;AND COUNT
COM_SEG      ENDS
.
.
.
IN_BUFF      EQU      52H      ;ASSIGN NAMES TO
OUT_BUFF     EQU      53H      ;INTERFACE REGISTER
STATUS       EQU      54H      ;ADDRESSES
RRDY         EQU      00000010B ;AND READY BITS
TRDY         EQU      00000001B ;IN STATUS REGISTER
.
.
.
ASSUME DS:DATA_SEG, ES:COM_SEG
MOV AX, DATA_SEG      ;INITIALIZE THE DS
MOV DS, AX            ;AND ES REGISTERS
MOV AX, COM_SEG
MOV ES, AX
.
.
.
MOV DI, OFFSET BUFFER ;INITIALIZATION NEEDED
MOV COUNT, DI          ;FOR INPUT
MOV CX, 81              ;CLEAR DF FOR AUTOINCREMENT
CLD
NEXT_IN:   IN AL, STATUS      ;IDLE UNTIL CHARACTER
TEST AL, RRDY        ;IS PUT IN INPUT
JZ NEXT_IN          ;BUFFER REGISTER
IN AL, IN_BUFF      ;INPUT CHARACTER
OR AL, 0             ;CHECK PARITY AND
JPE NO_ERROR        ;BRANCH TO ERROR
JMP NEAR PTR ERROR ;IF PARITY IS ODD
NO_ERROR:  AND AL, 7FH      ;ELSE, CLEAR PARITY BIT
STOSB             ;MOVE CHARACTER TO BUFFER
CMP AL, ODH        ;CHECK FOR CARRIAGE RETURN
LOOPNE NEXT_IN      ;LOOP IF NO CR OR OVERFLOW
JNE OVERFLOW        ;BRANCH ON OVERFLOW
MOV AL, OAH          ;APPEND LINE FEED
STOSB             ;STORE NO. OF CHARACTERS
SUB DI, COUNT
MOV COUNT, DI
.
.
.
OVERFLOW:  MOV SI, OFFSET MESSAGE ;INITIALIZATION NEEDED
MOV CX, 17            ;FOR OUTPUT
NEXT_OUT:  IN AL, STATUS      ;IDLE UNTIL OUTPUT
TEST AL, TRDY        ;BUFFER REGISTER
JZ NEXT_OUT          ;IS EMPTY
LODSB             ;OUTPUT CHARACTER
OUT OUT_BUFF, AL      ;LOOP UNTIL MESSAGE COMPLETE
LOOP NEXT_OUT
.
.
.
```

Figure 6-6 Programmed I/O example.

and the next four executable instructions put the offset of BUFFER into the DI register and COUNT, set CX to the maximum line length, and clear the DF flag. Next, a two-level nest of loops is entered which inputs the line of characters. The first three instructions in the nest comprise the inner loop, which causes the computer to idle until the data input status bit becomes 1. Then a character is input

to the AL register, where its parity is checked and, if it is even, it is cleared. After clearing the parity bit the input character is stored and then compared with the ASCII code for a carriage return. The LOOPNE instruction will cause another character to be input if neither a carriage return is detected nor the count in CX is decremented to 0. If the last character is a carriage return, a line feed is appended and the total number of characters is put in COUNT. In the event that the count reaches 0 without a carriage return being encountered a branch is made to a program sequence that causes the overflow message to be output to the terminal. This sequence also includes a two-level nest of loops, with the inner loop simply waiting for the data output status bit to become 1.

If the line being input were filled with numbers that must be converted to integer and other formats before they are used, then the processing of the line would have to include a sequence for performing the conversions. The conversions could be done as described in Sec. 5-5 and could be accomplished by calling a conversion procedure, which may be a system procedure. If the line needs to be divided into fields and the fields may contain numbers to be put into different formats, the conversion process could be quite complicated.

If there is more than one device using programmed I/O, it is necessary to poll the ready bits of all of the devices. Figure 6-7 shows a program sequence for inputting data from three devices. The addresses of their status registers have been equated to STAT1, STAT2, and STAT3 and the procedures PROC1, PROC2, and PROC3 are called upon to perform the input. Bit 5 is taken to be the input ready bit in all three of the status registers. The variable FLAG is for terminating the input process and is initially set to 0. It is assumed that the first input procedure will check a termination condition and set FLAG to 1, thereby causing the input process to cease after all currently pending inputs have been completed. The program sequence gives the devices a priority. The device corresponding to STAT1 has the highest priority and the other two devices must wait until this device is

Figure 6-7 Priority polling.

```

INPUT:    MOV    FLAG,0          ;CLEAR FLAG
          IN     AL,STAT1        ;CHECK STAT1
          TEST   AL,20H          ;AND IF NO INPUT IS
          JZ    DEV2             ;READY, GO TO DEV2,
          CALL   FAR PTR PROC1   ;ELSE INPUT FROM DEVICE 1
          CMP    FLAG,1           ;IF FLAG IS CLEAR
          JNZ    INPUT            ;INPUT ANOTHER DATUM
          IN     AL,STAT2        ;CHECK STAT2
          TEST   AL,20H          ;AND IF NO INPUT IS
          JZ    DEV3             ;READY, GO TO DEV3
          CALL   FAR PTR PROC2   ;ELSE, INPUT FROM DEVICE 2
          CMP    FLAG,1           ;IF FLAG IS CLEAR
          JNZ    INPUT            ;INPUT ANOTHER DATUM
          IN     AL,STAT3        ;CHECK STAT3
          TEST   AL,20H          ;AND IF
          JZ    NO_INPUT         ;INPUT IS AVAILABLE,
          CALL   FAR PTR PROC3   ;INPUT FROM DEVICE 3,
          CMP    FLAG,1           ;ELSE CHECK FLAG, IF CLEAR
          JNZ    INPUT            ;INPUT ANOTHER DATUM,
                               ;ELSE CONTINUE
          .
          .
          .

```

```

    INPUT:   MOV    FLAG,0      ;CLEAR FLAG
             IN     AL,STAT1   ;INPUT FROM DEVICE 1
             TEST   AL,20H     ;IF INPUT IS READY
             JZ    DEV2
             CALL   FAR PTR PROC1
             IN     AL,STAT2   ;INPUT FROM DEVICE 2
             TEST   AL,20H     ;IF INPUT IS READY
             JZ    DEV3
             CALL   FAR PTR PROC2
             IN     AL,STAT3   ;INPUT FROM DEVICE 3
             TEST   AL,20H     ;IF INPUT IS READY
             JZ    NO_INPUT
             CALL   FAR PTR PROC3
             CMP    FLAG,1      ;REPEAT LOOP IF FLAG
             JNZ    INPUT      ;IS STILL CLEAR
             .
             .

```

Figure 6-8 Round-robin polling.

idle. The device corresponding to STAT3 cannot be serviced until neither of the other two devices is ready.

Figure 6-8 shows how the devices could be serviced in turn. This is referred to as a *round-robin arrangement*. Such an arrangement essentially gives all three devices the same priority. In this example, FLAG is checked only at the bottom of the loop and, if it is 1, the loop is exited without testing for additional inputs.

6-3 INTERRUPT I/O

Even though programmed I/O is conceptually simple, it can waste a considerable amount of time while waiting for ready bits to become active. In the above example, if the person typing on the terminal could type 10 characters per second and only 10 μ s is required for the computer to input each character, then approximately

$$\frac{99,990}{100,000} \times 100\% = 99.99\%$$

of the time is not being utilized. This is all right if no other processing could be done while the input is taking place, but if other processing is unnecessarily delayed, then a different approach is needed.

As was seen in Sec. 4-4 an *interrupt* is an event that causes the CPU to initiate a fixed sequence, known as an *interrupt sequence*. Before an 8086 interrupt sequence can begin, the currently executing instruction must be completed unless the current instruction is a HLT or WAIT instruction. (The WAIT instruction is primarily used to wait for the completion of a coprocessor instruction and is discussed in Chap. 11.)

For a prefixed instruction, because the prefix is considered as part of the instruction, the interrupt request is not recognized between the prefix and the instruction. In the case of the REP instruction, the interrupt request is recognized after the primitive operation following the REP is completed, and the return address

is the location of the REP prefix. For MOV and POP instructions in which the destination is a segment register, an interrupt request is not recognized until after the instruction following the MOV or POP instruction is executed. Provided that the segment register is filled first, this allows the contents of both a segment register and a pointer to be changed without interruption.

For the 8086, once the interrupt request has been recognized, the interrupt sequence consists of:

1. Establishing a type N.
2. Pushing the current contents of the PSW, CS, and IP onto the stack (in that order).
3. Clearing the IF and TF flags.
4. Putting the contents of memory location 4^*N into IP and the contents of $4^*N + 2$ into CS.

Thus, an interrupt causes the normal program sequence to be suspended and a branch to be made to the location indicated by the double word beginning at four times the type (i.e., the *interrupt pointer*). Control can be returned to the point at which the interrupt occurred by placing an IRET instruction at the end of the *interrupt routine*.

It was mentioned that there are two classes of interrupts, internal and external interrupts, with external interrupts being caused by a signal being sent to the CPU through one of its pins, which for the 8086 is either the NMI pin or the INTR pin. An interrupt initiated by a signal on the NMI pin is called a *nonmaskable interrupt* and will cause a type 2 interrupt regardless of the setting of the IF flag. Nonmaskable interrupt signals are normally caused by circuits for detecting catastrophic events such as imminent power failure, or by external events or clock pulses that must be processed immediately. An interrupt on the INTR pin is masked by the IF flag so that if this flag is 0, the interrupt is not recognized until IF returns to 1. When IF = 1 and a maskable external interrupt occurs, the CPU will return an acknowledgment signal to the device interface through its INTA pin and initiate the interrupt sequence. The acknowledgment signal will cause the interface that sent the interrupt signal to send to the CPU (over the data bus) the byte which specifies the type, and hence the address of the interrupt pointer. The pointer, in turn, supplies the beginning address of the interrupt routine. As indicated in Fig. 4-26, the type of a maskable external interrupt must be in the range 5 to 255, inclusive. The entire process is diagrammed in Fig. 6-9. For a nonmaskable external interrupt (type 2), the INTA signal is not sent. Unless specifically stated to the contrary, maskable interrupts are assumed in the examples that follow.

Returning to the example involving the input of a line of characters from a terminal, it is seen that if interrupt I/O is used, the system does not have to wait between keystrokes on the terminal. By designing the interface so that it sends an interrupt to the CPU whenever its input data buffer register receives data, the CPU can perform other tasks while waiting for an interrupt. When a character arrives

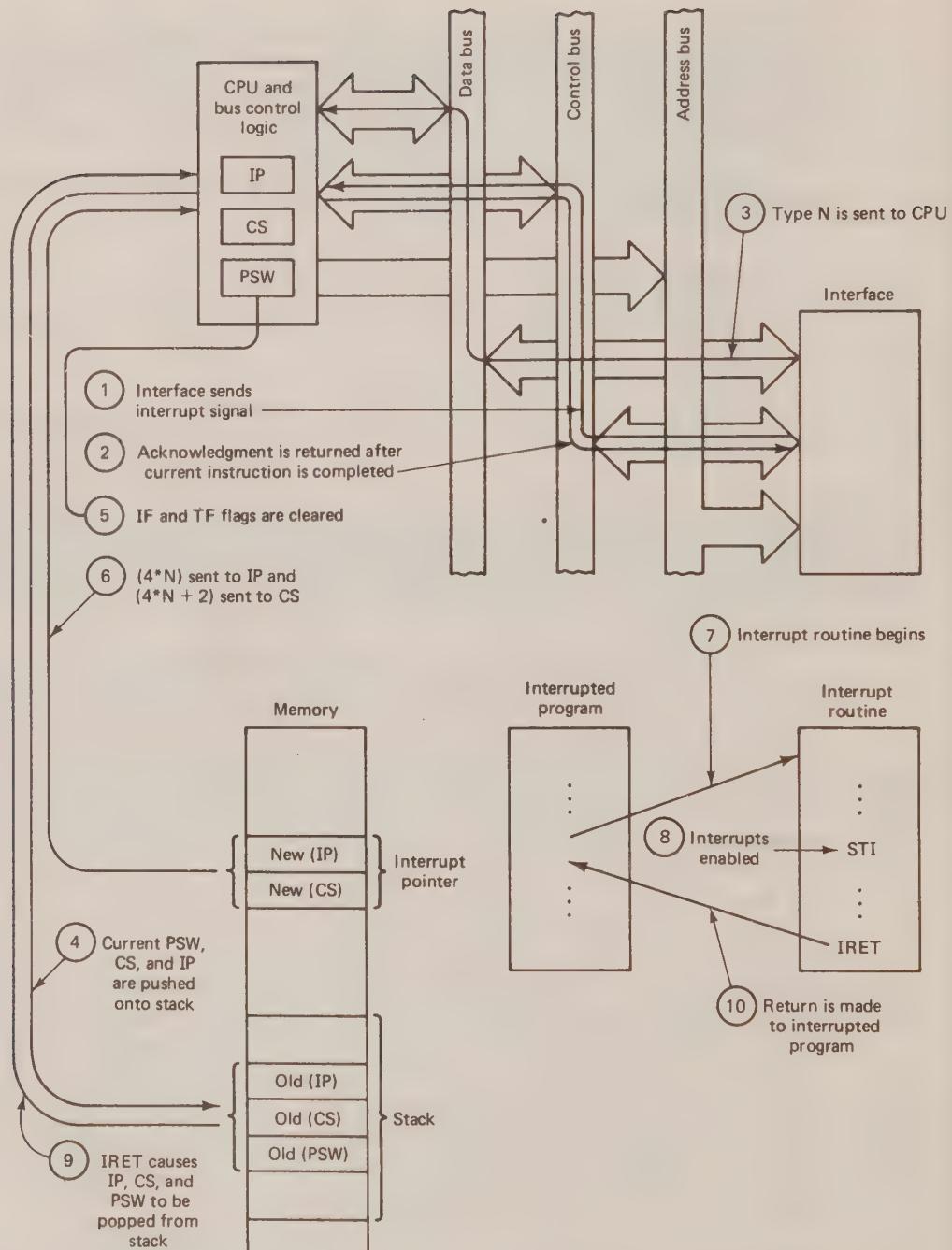


Figure 6-9 Sequence of events during a maskable interrupt and subsequent return.

at the interface, the CPU is notified by the interrupt signal, the interrupt routine can input the character, and then the CPU can return to its other processing.

As an example, consider the processing components and their relationships illustrated in Fig. 6-10. The main program is to initialize the necessary interrupt pointers and then begin its normal processing. As it is executing, interrupt I/O is used to input a line of characters to a buffer that is pointed to by BUFF_POINT. It is assumed that all variables are defined in a segment DATA_SEG whose segment address has been stored in DS. The location CODE, which is initially set to 0, is used to indicate when a complete line has been input (CODE = 2) or the input buffer has overflowed (CODE = 1). An overflow occurs when 81 characters are received without a carriage return being detected. In the event of an overflow, input interrupts are disabled, output interrupts are enabled, and interrupt I/O is used to output an error message from MESSAGE. After a line has been input a subprogram LINE_PROC is called to process the line. This processing could consist of moving the data to a larger text buffer area, converting the data in the line to a different format, or performing a variety of other tasks.

The interrupt routine for performing the input and (in the event of an overflow) the output is given in Fig. 6-11. The input routine appears first and the

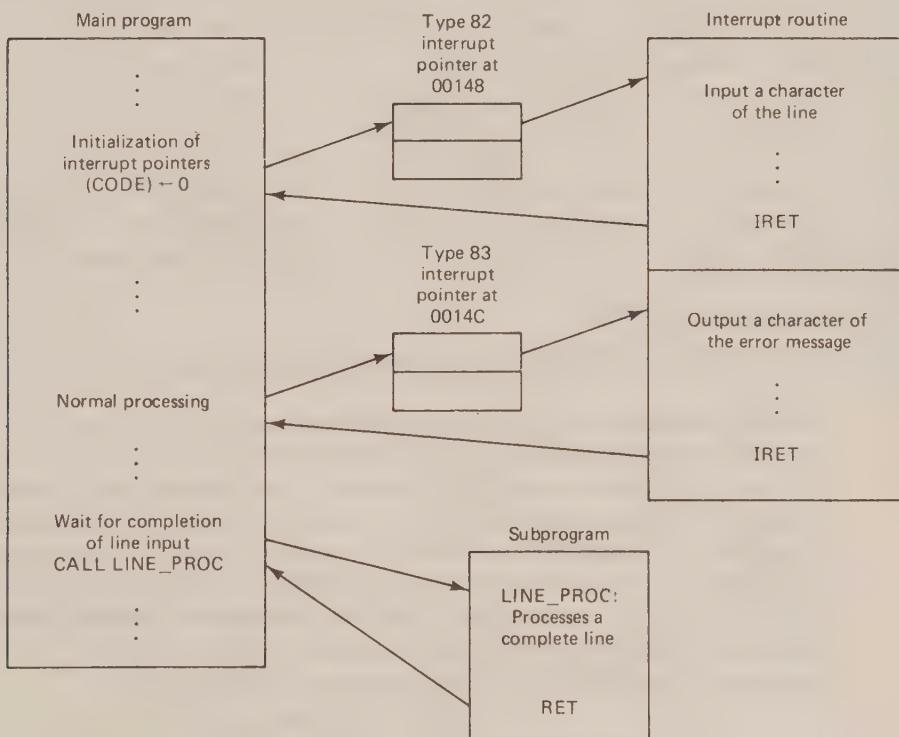


Figure 6-10 Processing component relationships for the line input example involving interrupt I/O.

```

INT SEG      SEGMENT
ASSUME CS:INT SEG, DS:DATA SEG           ;PARAMETERS ARE ACCESSIBLE VIA DS
    IN BUF     EQU 52H
    OUT BUF    EQU 53H
    CONTROL   EQU 54H
    ENABLE OUT EQU 00000001B
    INT_ROUT: PUSH AX
                ;SAVE REGISTERS
                PUSH BX
                IN AL, IN_BUF          ;INPUT CHARACTER
                MOV BX, BUFF POINT     ;AND STORE IN
                MOV [BX], AL           ;MEMORY BUFFER
                INC BX                 ;INCREMENT BUFFER POINTER
                INC COUNT              ;AND COUNT
                MOV BUFF POINT, BX     ;STORE BUFFER POINTER
                CMP AL, ODH             ;CHECK FOR CARRIAGE
                JNZ NO CR               ;RETURN AND
                MOV BYTE PTR [BX+1], OAH ;APPEND A LINE FEED
                INC COUNT              ;SET CODE TO 2 SO MAIN ROUTINE
                MOV CODE, 2              ;MAY CALL PROCEDURE LINE PROC
                XOR AL, AL               ;ALSO, DISABLE INPUT DEVICE
                OUT CONTROL, AL         ;CONTINUE
                JMP CONT
    NO_CR:    CMP COUNT, 81            ;CHECK FOR OVERFLOW
                JB CONT                ;IF NO, RETURN
                MOV CODE, 1              ;OTHERWISE, SET CODE TO 1,
                MOV MSGCOUNT, 0           ;ZERO MSGCOUNT
                MOV AL, ENABLE_OUT        ;DISABLE INPUT AND ENABLE
                OUT CONTROL, AL         ;OUTPUT
                POP BX                 ;RESTORE REGISTERS
                POP AX
                IRET

;THE FOLLOWING INTERRUPT SERVICE ROUTINE OUTPUTS ONE CHARACTER
;FROM MESSAGE WHEN INTERRUPT FROM OUTPUT DEVICE OCCURS

OVERFLOW:  PUSH AX                  ;SAVE REGISTERS
            PUSH BX
            MOV BX, MSGCOUNT
            MOV AL, MESSAGE[BX]       ;OUTPUT A 'CHARACTER
            OUT OUT BUF, AL
            INC MSGCOUNT             ;INCREMENT COUNTER
            CMP AL, OAH               ;LAST CHARACTER IN MESSAGE?
            JNE RETURN               ;NO, RETURN. OTHERWISE,
            XOR AL, AL                ;DISABLE FURTHER INTERRUPT
            OUT CONTROL, AL          ;FROM OUTPUT
            RETURN:    POP BX
                        ;RESTORE REGISTERS
                        POP AX
                        IRET

INT SEG      ENDS

```

Figure 6-11 Interrupt routine for inputting a line of characters.

output routine second. In both cases the registers that are used by the routine are pushed onto the stack and then popped from the stack immediately prior to the return. As indicated by the EQU directives, it is assumed that the addresses of the data-in and data-out buffer registers are 0052 and 0053, the address of the control register is 0054, and bits 1 and 0 of the control register are for enabling and disabling the input and output interrupts, respectively. The variables COUNT and MSGCOUNT are defined as word variables and both are initially set to 0. COUNT keeps a running count of the characters that have been input to the line buffer. In particular, note that if an overflow occurs, the instructions

```

MOV AL,ENABLE_OUT
OUT CONTROL,AL

```

cause the input interrupts to cease being generated by the interface and permit

output interrupts to be generated. This allows the overflow message to be sent to the user's terminal.

A sequence for initializing the interrupt pointers, which assumes the input interrupt has type 82 and the output interrupt has type 83, is given in Fig. 6-12. The last two instructions are for enabling input interrupts. The code for loading the interrupt pointers can appear in the user's program, but it is part of the initialization performed by the operating system if interrupts are handled through the operating system. If the interrupt pointers are to be set by the user's program, they could alternatively be set when the program is loaded by inserting the following directives at the beginning of the program:

```
INT_VEC SEGMENT AT 0
ORG 148H
DD INT_ROUT
DD OVERFLOW
INT_VEC ENDS
```

In the above example only a single memory buffer has been included and when the buffer has received a line of text, LINE_PROC is called to move the data elsewhere or process it in place. It is assumed that input to the buffer will be suspended until LINE_PROC has completed its work, and that when LINE_PROC is finished the buffer will be available to receive a new line. At that time BUFF_POINT would be reset to the beginning address of the buffer (presumably by LINE_PROC).

On the other hand, for a program that continually receives new data and cannot suspend the input while processing a buffer, at least two buffers are needed. Figure 6-13 gives a flowchart of how LINE_PROC could be structured when double buffering is required. In the flowchart BUFFER1 and BUFFER2 represent the beginning addresses of the two buffer areas. It is assumed that the normal processing initially sets FLAG to 0 and puts the address BUFFER1 in BUFF_POINT. When the first line is ready to be processed, LINE_PROC sets FLAG to 1 and switches the contents of BUFF_POINT to point to BUFFER2. It puts the address BUFFER1 in BX and the character count in DI, and then the line processing uses the contents of these two registers. With each succeeding line, the contents of FLAG, which is

Figure 6-12 Program sequence for initializing the interrupt pointers.

```
PUSH DS ;SAVE DS
XOR AX,AX
MOV DS,AX ;CLEAR DS SO AN ABSOLUTE LOCATION
MOV AX,OFFSET INT_ROUT ;MAY BE ADDRESSED
MOV BX,148H
MOV [BX],AX ;MOVE OFFSET OF INT_ROUT TO 148H
MOV AX,OFFSET OVERFLOW ;MOVE OFFSET OF OVERFLOW TO 14CH
MOV [BX+4],AX ;MOVE OFFSET OF OVERFLOW TO 14CH
MOV AX,INT_SEG
MOV [BX+2],AX ;MOVE SEGMENT BASE TO 14AH
MOV [BX+6],AX ;MOVE SEGMENT BASE TO 14EH
POP DS ;RESTORE DS
MOV AL,00000010B ;ENABLE INPUT DEVICE
OUT CONTROL,AL
.
.
.
```

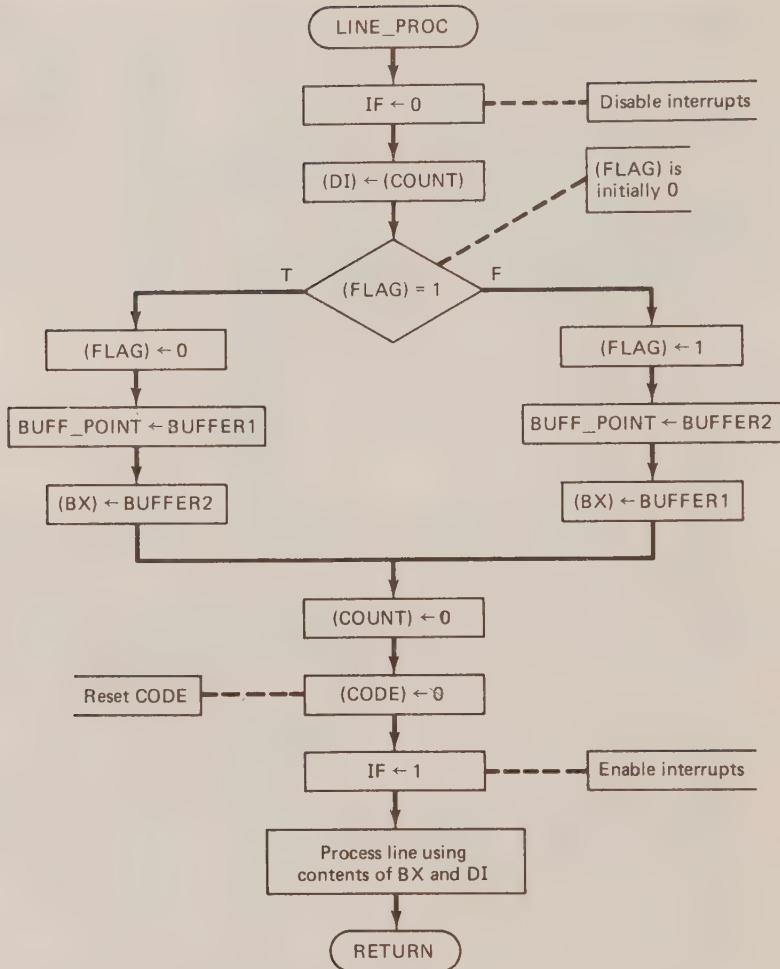


Figure 6-13 Flowchart of `LINE_PROC` when double buffering is used.

logically complemented with each call to `LINE_PROC`, causes the uses of the addresses `BUFFER1` and `BUFFER2` to be switched. Therefore, the buffers are alternately being filled and processed. While one buffer is being filled, the other is being processed, and vice versa. Note that the time required to process a line must be less than the time needed for inputting a line. In order to ensure that the initialization is complete before inputting the next line, it is necessary to disable interrupts while the buffers are switched. Once `IF` is set to 1, characters sent by

the terminal will be able to cause interrupts and be input to the buffer currently being filled.

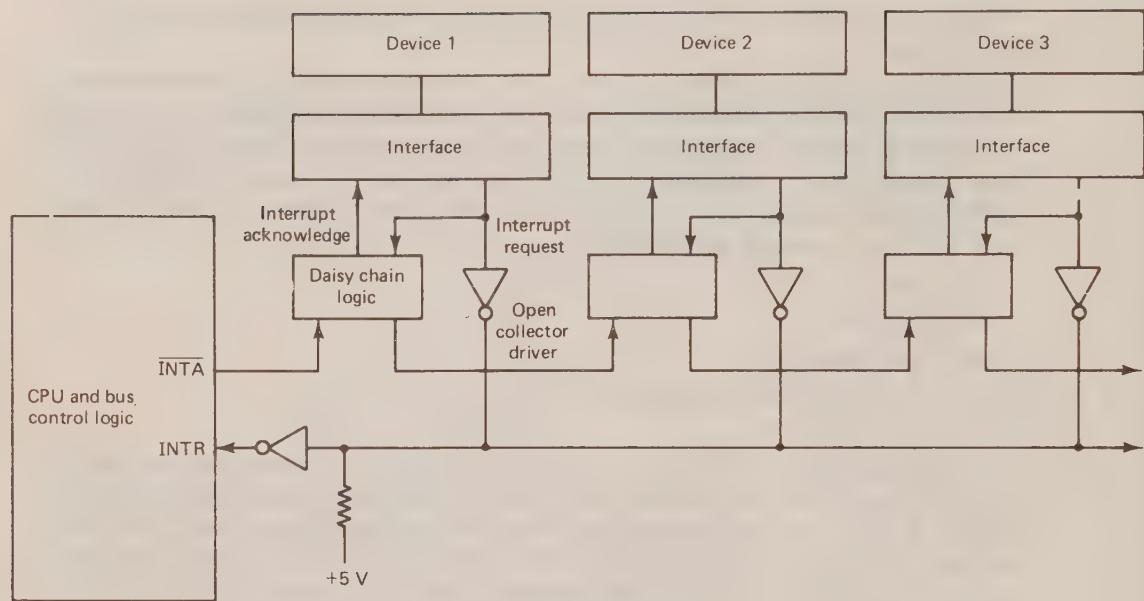
In the above discussion the possibility of interrupts arriving simultaneously (or at least within the same instruction cycle) was tacitly overlooked. So, nothing has been said about incorporating a priority scheme into an interrupt system. There are several ways of combining priority with interrupt I/O, some involving only software, some only hardware, and some a combination of the two. Let us consider the following means of giving priority to an interrupt system:

1. Polling.
2. Daisy chaining.
3. Interrupt priority management hardware.

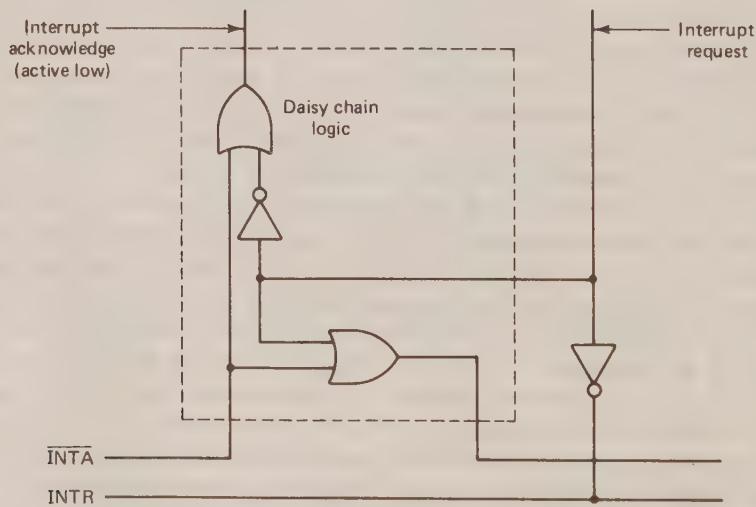
Although there are numerous external interrupt types available for use by the interfaces in an 8086 system and it is seldom necessary to assign two interfaces the same type, a set of interfaces may be given the same type so that polling can be used to assign them priorities. By putting a program sequence (similar to the one in Fig. 6-7) at the beginning of the interrupt routine, the priority of the interfaces could be established by the order in which they are polled by the sequence.

Daisy chaining is a simple hardware means of attaining a priority scheme. It consists of associating a logic circuit with each interface and passing the interrupt acknowledge signal through these circuits as shown in Fig. 6-14(a). The details of a daisy chain logic circuit are shown in Fig. 6-14(b). If an interface has made a request, the active low signal INTA causes an active low interrupt acknowledge signal to be sent to the interface and INTA is blocked from passing on to the next interface. If an interface has not made a request, then the INTA signal is allowed to pass through its daisy chain logic unchanged. Therefore, as the INTA signal proceeds along the daisy chain, the requesting interface that is closest to the CPU will intercept the acknowledgment signal, send the type, and complete the interrupt sequence, eventually dropping its request as it is serviced. Other interfaces farther down the daisy chain that have made requests will not receive the acknowledgment signal and will maintain their requests. After an STI instruction reenables the IF flag or IF is reenabled by the PSW being restored by an IRET instruction, the CPU will recognize further requests and send out another INTA signal. By this time the interface whose request has been serviced will have dropped its request and will not compete for the INTA signal. Therefore, the priority of an interface is determined by its position on the daisy chain. The closer it is to the CPU, the higher its priority.

A more flexible hardware/software priority arrangement can be had by designing a programmable interrupt priority management circuit and including it in the bus control logic. Typically, such a circuit would be designed-and inserted in the system as shown in Fig. 6-15. The INTR and INTA pins would not be connected to the interfaces, but would be connected only to the management circuit. The



(a) Daisy chain



(b) Logic

Figure 6-14 Daisy chain arrangement.

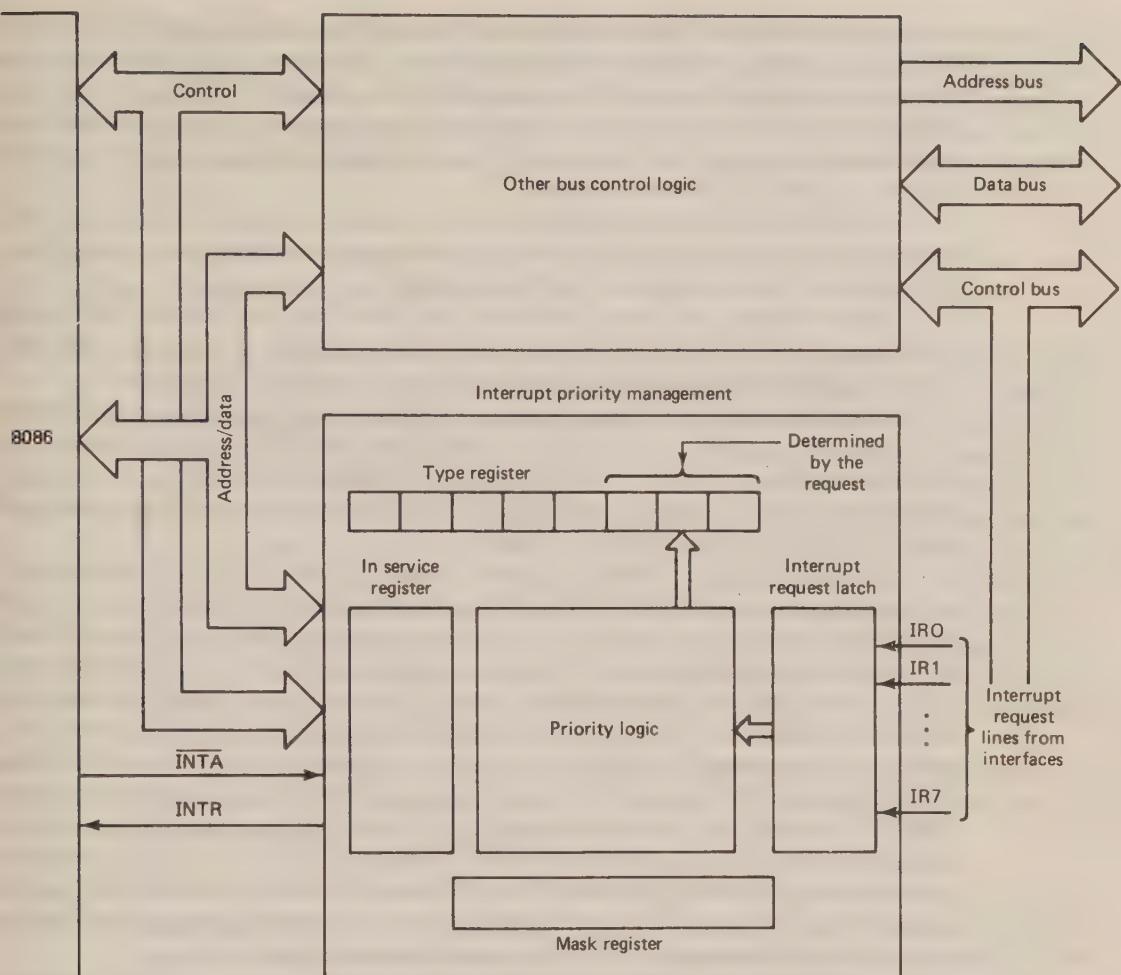


Figure 6-15 Representative interrupt priority management design.

interrupt request lines emanating from the interfaces would be connected directly to the management circuit and would not be tied together. The management circuit would contain the logic needed to assign priorities to the incoming requests. For example, the highest priority could be given to IR0, the next highest priority to IR1, and so on. When an interrupt request is recognized by the priority logic as having the highest priority, then the three LSBs of the type register are set to the number of the request line, a bit is set in the in-service register, and an interrupt is sent to the CPU. If IF = 1, then the CPU returns an acknowledge signal and the management circuit sends the CPU the type. All requests having lower priority are blocked until the bit in the in-service register is cleared, an action which is normally done by the interrupt routine. Therefore, when IF is reenabled by an STI instruction, higher-priority requests may interrupt the currently executing interrupt rou-

tine, but lower-priority requests will be blocked by the priority logic until the bit that was set in the in-service register is cleared. This gives the interrupt routine control over when lower-priority requests will be recognized. In order for the program to be able to clear bits in the in-service register, this register must be programmable, i.e., it must have an I/O port address so that it can be accessed using the IN and OUT instructions.

In addition to the built-in priority, a 1-byte mask register is included to allow masking of the individual requests. Bit n in this register would be for masking IR n . It is assumed that this register is programmable.

In the example, the least significant 3 bits of the type register are determined by the request selected by the priority logic. If this register is programmable, the 5 most significant bits could be initialized when the system is turned on.

Many microprocessor manufacturers produce interrupt priority management IC devices to supplement their CPU devices. The Intel 8259A programmable interrupt controller is designed to work with the 8086 and 8088 CPUs. It is similar to the management circuit shown in Fig. 6-15, but has many features not considered above. It is discussed at length in Chap. 8.

It should be apparent that for a system that involves several I/O devices that may be inputting and/or outputting while processing is taking place, the interrupt processing hardware and software may be very complex. This is particularly true for multiprogramming systems. Whether a software system is a special purpose system serving only one application or a general purpose system capable of satisfying numerous applications, there is generally one main program component that controls all of the others. In a general purpose environment this component is the resident monitor. In either case, it is this main component that initializes the system when it is brought up. Among other things it inserts the interrupt pointers in the low part of memory (i.e., addresses 00000 through 003FF). In a special purpose system setting the interrupt pointers would normally remain fixed until the system is shut down, but in a multiprogramming situation the operating system may change some of the interrupt pointers as the system's needs change.

Complicating the software system is the problem of communicating with interrupt routines. Although passing parameter addresses via the stack or a table is easily accomplished when procedures are called, this method does not apply to interrupt routines associated with external interrupts. Because external interrupts occur at random points within a program, there is no way of knowing where to insert the linkage code needed for passing parameters. Therefore, parameters must be passed through common areas and/or absolute locations. For a special purpose system in which only one program is run it is possible for all programming components to be compiled and linked so that they can communicate without passing parameters.

On the other hand, if an operating system controls the activity it is given the responsibility of keeping track of all the program components currently in memory, and data references between the interrupt routines and the other program components are made with the help of the operating system. Ordinarily, the operating system monitors and updates several tables, called *system tables*, which contain the

vital information about the state of the system. Some of the tables, such as the one that contains the important physical characteristics of the system I/O devices, are static, while others, including the table for storing beginning addresses of routines, may be dynamic. Multiprogramming systems include queues for I/O requests waiting to be serviced, and certainly the tables associated with these queues are dynamic.

6-4 BLOCK TRANSFERS AND DMA

If the data transfer rate to or from an I/O device is relatively low, then the communication can be performed using either programmed or interrupt I/O. But executing instructions and performing interrupt sequences take more time than is sometimes available. Some devices, such as magnetic tape and disk units and analog-to-digital (A/D) converters, may operate at data rates that are too high to be handled by byte or word transfers. Data rates for I/O and mass storage devices are often determined by the devices, not the CPU, and the computer must be capable of executing I/O according to the maximum speed of the device. For a disk unit the data rate is determined by the speed with which data pass under the read/write head and quite often this rate exceeds 200,000 bytes per second. Thus, there is less than 5 microseconds to transfer each byte to or from memory. For data rates of this magnitude, block transfers, which use DMA controllers to communicate directly with memory, are required.

The activity involved in transferring a byte or word over the system bus is called a *bus cycle*. The execution of an instruction may require more than one bus cycle. For example, the instruction

```
MOV AL,TOTAL
```

would use a bus cycle to bring in the contents of TOTAL in addition to the cycles needed to fetch the instruction. A block transfer of 100 bytes would require 100 bus cycles if only one byte is moved at a time and 50 bus cycles if words beginning at an even address are moved.

During any given bus cycle, one of the system components connected to the system bus is given control of the bus. This component is said to be the *master* during that cycle and the component it is communicating with is said to be the *slave*. The CPU with its bus control logic is normally the master, but other specially designed components can gain control of the bus by sending a *bus request* to the CPU. After the current bus cycle is completed the CPU will return a *bus grant* signal and the component sending the request will become the master. Taking control of the bus for a bus cycle is called *cycle stealing*. Just like the bus control logic, a master must be capable of placing addresses on the address bus and directing the bus activity during a bus cycle. The components capable of becoming masters are processors (and their bus control logic) and DMA controllers. Sometimes a DMA controller is associated with a single interface, but they are often designed to accommodate more than one interface.

The 8086 receives bus requests through its HOLD pin and issues grants from its hold acknowledge (HLDA) pin. A request is made when a potential master sends a 1 to the HOLD pin. Normally, after the current bus cycle is complete the 8086 will respond by putting a 1 on the HLDA pin. When the requesting device receives this grant signal it becomes the master. It will remain master until it drops the signal to the HOLD pin, at which time the 8086 will drop the grant on the HLDA pin. One exception to the normal sequence is that if a word which begins at an odd address is being accessed, then two bus cycles are required to complete the transfer and a grant will not be issued until after the second bus cycle. Other exceptions are discussed in the later chapters.

When a DMA controller becomes master it places an address on the address bus and sends the interface the necessary signals to cause it to put data on or receive data from the data bus. Since the DMA controller determines when the bus request is dropped, it can return control to the CPU after each datum is transferred and then request control again when the next datum is ready, or it can retain control until the entire block is moved. The former is the usual case because this allows the CPU to continue its work until the next datum is available. The sequence of actions taken during a single output DMA is illustrated in Fig. 6-16.

A block transfer is a succession of the datum transfers described above. Each successive DMA uses the next consecutive memory location. Although DMA controllers could be designed around a variety of configurations, all of these configurations must satisfy certain requirements. Since the DMA controller must supply the address when it is master, it must be capable of storing the next memory address that is to be accessed. It must also be able to know when to stop the block transfer. A minimal DMA controller/interface design is shown in Fig. 6-17. The controller design contains two registers in addition to whatever status and control registers are required, one for holding the address of the next memory location and one for noting the number of bytes yet to be transferred. These registers are I/O ports, so that they can be initialized before the block transfer begins. After each datum is moved the address register is incremented (or perhaps decremented, depending on the design) and the byte count register is decremented. The incrementing and decrementing are by 1 for byte transfers, and by 2 for word transfers. During a block input byte transfer, the following sequence occurs as the datum is sent from the interface to the memory:

1. The interface sends the controller a request for DMA service.
2. The controller gains control of the bus.
3. The contents of the address register are put on the address bus.
4. The controller sends the interface a DMA acknowledgment which tells the interface to put data on the data bus. (For an output it signals the interface to latch the next data placed on the bus.)
5. The data byte is transferred to the memory location indicated by the address bus.
6. The controller relinquishes the bus.

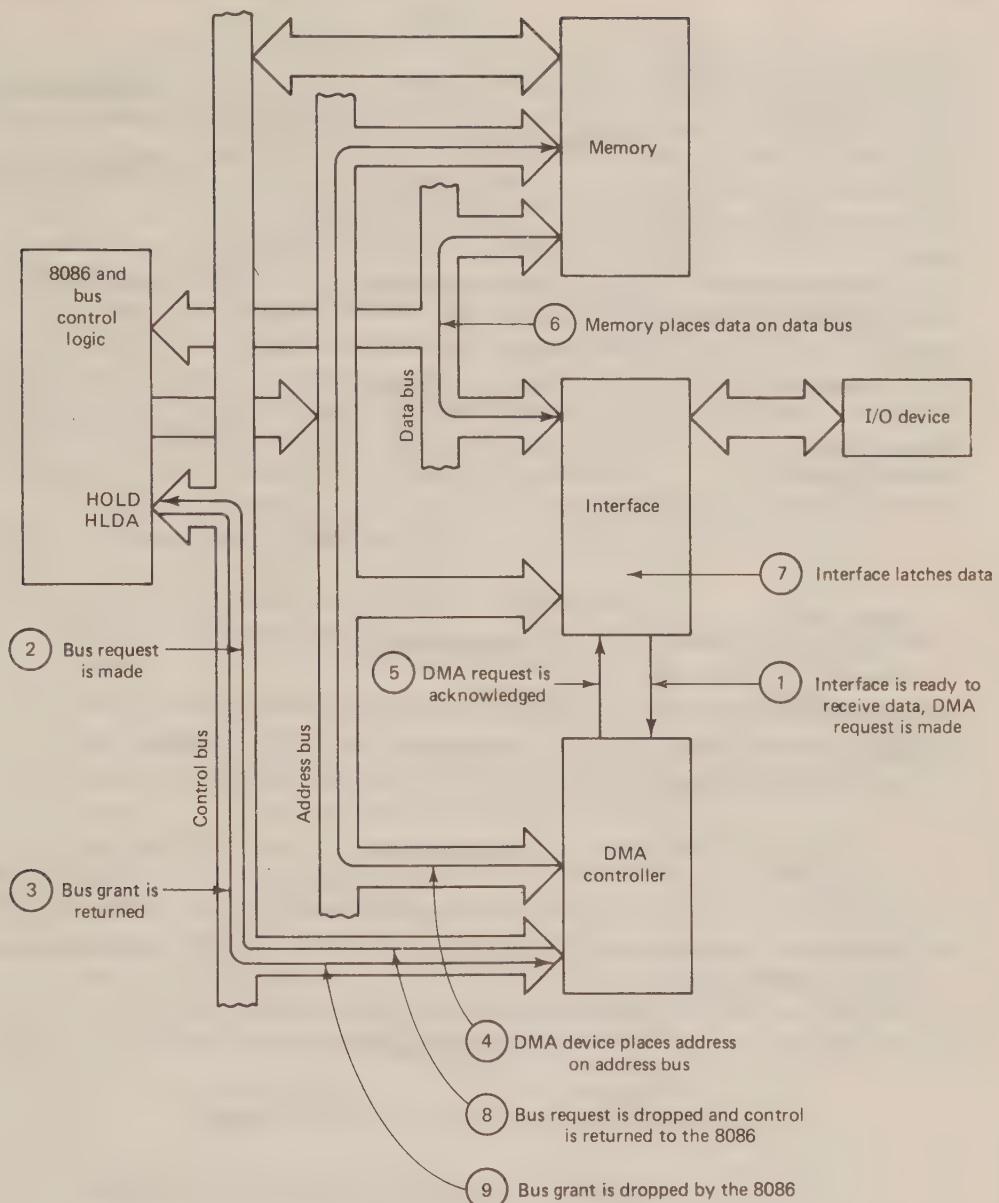


Figure 6-16 Single datum output transfer during a block transfer.

7. The address register is incremented by 1.
8. The byte count register is decremented by 1.
9. If the byte count register is nonzero, return to step 1; otherwise, stop.

The controller/interface design shows bidirectional address lines connected to the controller and only unidirectional address lines going to the interface. This

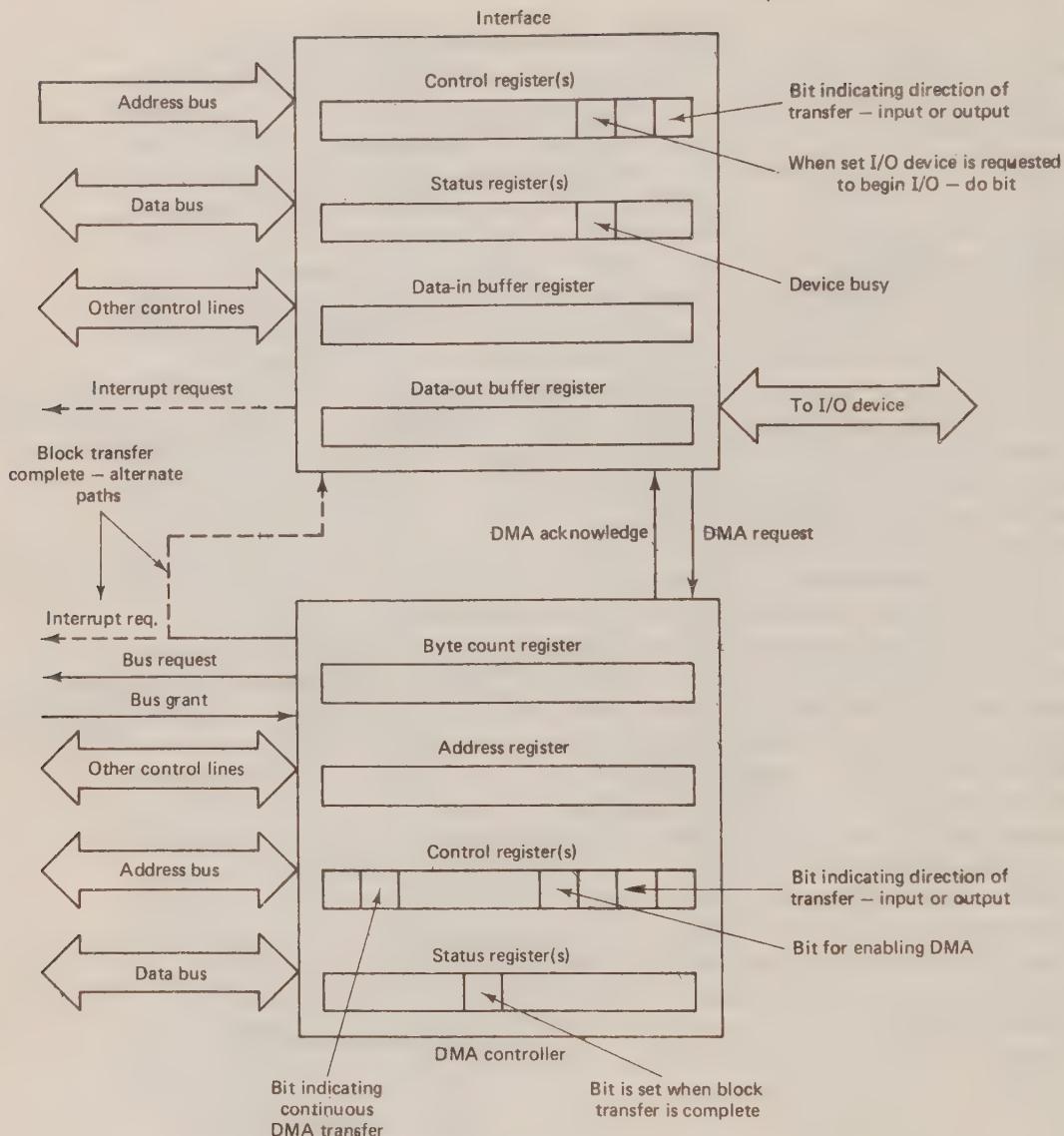


Figure 6-17 Minimal DMA controller/interface configuration.

is, of course, because the controller can become master and place addresses on the bus while the interface can only receive addresses. Bidirectional data lines are shown going to both the interface and the controller even though only the interface can transfer data between the data bus and I/O device. As with an interface, the CPU must be able to communicate with the registers in a controller and this must be done over the data lines.

If an interface is connected to an I/O device that can perform both input and output, there must be a bit in its status and control registers which indicates the type of transfer to be conducted. In addition, these registers must contain a "do" bit for initiating the I/O activity (a bit that can be sensed by the I/O device) and a bit for indicating whether or not the device is currently busy. The controller status and control registers must contain an enable bit which controls whether or not it will recognize DMA requests from the interface and a bit that determines whether the bus is to be relinquished between transfers or the bus request is held high for the entire transfer. It must also include a data direction bit so that when the controller becomes master it knows whether to supervise an input or output transfer. To initiate a block transfer these interface and DMA control bits would need to be set as well as having to put beginning values in the address and byte count registers. The bit in the interface control register that directs the I/O device to perform the I/O, the do bit, would be set last.

A typical sequence for starting a block input transfer is given in Fig. 6-18. This sequence assumes the following bit definitions:

Bit 2 of INTSTAT—Busy bit for the I/O device.

Bit 1 of DMACON—Informs the controller of the transfer direction; 1 is for input.

Bit 3 of DMACON—Enables the controller so it will accept DMA requests.

Bit 6 of DMACON—Clear when bus is to be relinquished between transfers.

Bit 0 of INTCON—Informs the interface of the transfer direction; 1 is for input.

Bit 2 of INTCON—Do bit which starts the I/O activity.

After the sequence in Fig. 6-18 is executed the I/O device will begin inputting data, and the DMA controller will steal a bus cycle and transfer a byte from the interface to memory each time a byte is placed in the interface's data-in buffer register.

If a DMA controller connected to an 8086 system only outputs the lower 16 bits of an address, then it can only supervise a block transfer to one 64K segment

Figure 6-18 Typical sequence for initiating a block transfer.

```

IDLE:    IN      AL, INTSTAT      ;LOOP UNTIL DEVICE IS FREE
        TEST   AL, 4
        JNZ    IDLE
        MOV    AX, BYTE COUNT    ;PUT BLOCK SIZE IN
        OUT    BC REG, AX       ;BYTE COUNT REGISTER
        LEA    AX, BUFFER        ;PUT BEGINNING ADDRESS
        OUT    ADDR REG, AX     ;IN ADDRESS REGISTER
        MOV    AL, DMAC          ;SET DIRECTION AND ENABLE
        OR     AL, OAH           ;BITS IN CONTROL BYTE AND
        OUT    DMACON, AL        ;OUTPUT TO CONTROLLER
        MOV    AL, INTC          ;SET DIRECTION AND DO BIT
        OR     AL, 5              ;IN THE INTERFACE COMMAND
        OUT    INTCON, AL        ;REGISTER
        .
        .

```

of memory. To have access to all of memory the controller must be capable of generating 20-bit addresses and the 8086 must be able to communicate the high-order 4 bits of the initial address to the controller as well as the low-order 16 bits. Therefore, at least one more address byte than is indicated in Fig. 6-16 would need to be output to the controller. Normally, this byte would be output to a separate I/O port which could not be incremented during the block transfer. This implies that the transfer would have to be to or from a single 64K segment, even though the segment could differ from one block transfer to the next.

Upon completion of a block transfer a "done" bit will be set in one of the controller's status registers and a signal will be placed on one of its pins. The program that is currently executing could detect the end of the block transfer by periodically checking the done bit, or the signal sent through one of the pins could be used to cause an interrupt. In the latter case, the signal could be sent directly to the bus control logic or it could be sent to the interface and the interface could relay the request and handle the interrupt. Normally, a DMA controller is not designed to provide the interrupt type; therefore, if the signal is sent directly to the bus control logic, this logic should include interrupt management circuitry. The completion signal is often sent to the interface because an interrupt request may need to be made for reasons other than completion of the block transfer, e.g., error conditions that only the interface is aware of and that are not relayed to the controller.

Both the interface and the controller include status registers that record vital information during a transfer. Just what this information is depends on the design, but such things as the occurrence of transmission errors or running out of data before the byte count reaches zero (e.g., end of tape) certainly need to be noted. After the transfer is complete or has been abnormally halted the executing program or interrupt routine must check the status registers and take the appropriate actions. These actions could consist of reinputting the same information and/or printing out an error message. A procedure or interrupt routine for performing these checks and actions is called a *completion routine*.

Figure 6-19 is a flowchart of a completion routine that is designed to follow a disk input transfer. The routine checks for abnormal end and transmission errors. If the transfer is abnormally ended, an error message is printed and the memory location CODE is set to 1. When there is no abnormal end but there is a transmission error, the variable ERRCNT, which is assumed to have been set to 0 before the transfer started, is compared with 5. If ERRCNT is less than 5, it is incremented by 1 and the transfer is reinitiated; otherwise, CODE is set to 2 and an error message is printed. This means that transmission errors must occur in five consecutive block transfers before the attempt to execute the transfer is dropped. In the event that the transfer is completed without an error, the completion routine's only action is to set CODE to 0.

If more than one interface is connected to a DMA controller, then some of the registers must be duplicated. Clearly, if the interfaces are to be able to perform block transfers at the same time, then the controller has to contain a byte count

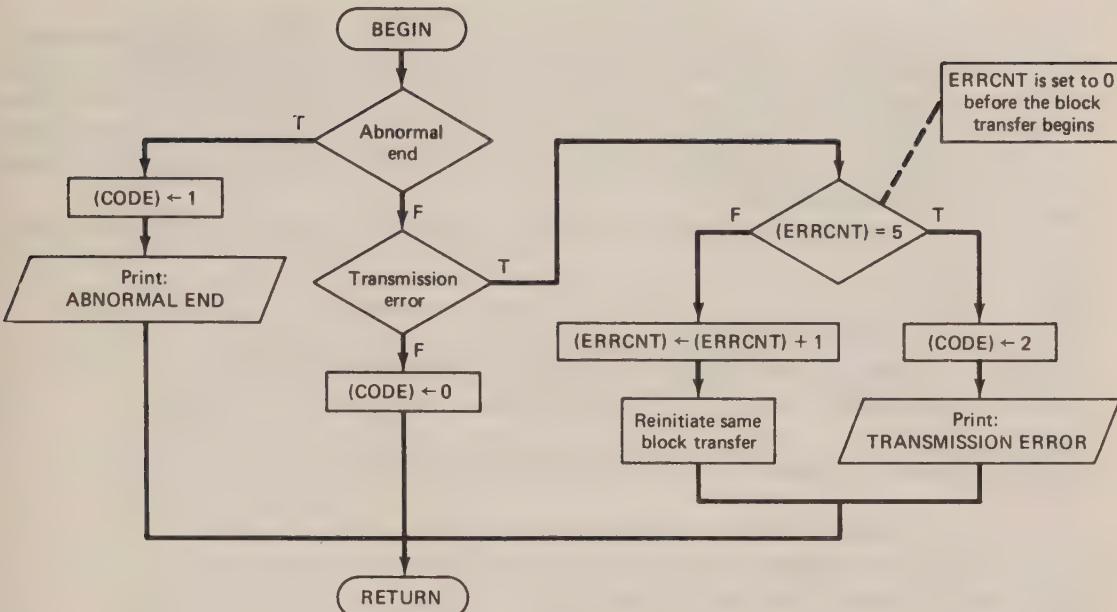


Figure 6-19 Flowchart of a completion routine.

and an address register for each interface. Also, the controller must be capable of prioritizing the requests from the interfaces. The controller's control registers need to include bits for individually enabling the incoming request lines and otherwise controlling the separate block transfers. Its status registers must contain bits for individually indicating which interfaces have completed their transfers. If the controller only has one pin to signal a completion, then the signal could be used only to indicate that some transfer is complete, and it would be left to the completion routine to determine which one by checking the controller's status bits.

Although the configuration of the DMA controller is independent of the I/O device it services, the design of the interface must closely reflect the characteristics of the I/O device associated with it. If an interface is connected to a nonstorage device, then the minimal configuration shown in Fig. 6-17 may be adequate, but for a storage device the interface needs to communicate search and address information to the device. The interface for a single-channel A/D conversion subsystem does not need to contain more than 2 or 3 bytes of control and status information, but it would need to contain bits for:

1. Enabling the interrupt capability.
2. Indicating errors.
3. Specifying the sample rate.
4. Enabling the DMA capability.
5. Initiating the input (i.e., setting the do bit).

With regard to item 4, some interfaces such as those connected to A/D subsystems, are designed to perform both byte or word transfers and block transfers. Even though these interfaces are connected to DMA controllers they may choose not to use them. If the A/D subsystem accommodates more than one channel, then interface registers would need to be included for specifying a channel number and a multiplexing mode.

In a magnetic tape subsystem the control bits would need to specify such things as:

1. Tape drive selected.
2. Information density in bits per inch (bpi) and tape speed.
3. Tape motion (rewind, forward, etc.).

Status bits should be included for indicating such things as:

1. Storage and transmission errors.
2. End of file encountered.
3. End of tape encountered.
4. Selected unit is not turned on.
5. Selected unit is busy.

Information is stored on most magnetic tapes by putting it in files and records that are accompanied by identifiers. It is retrieved by searching for the identifiers, not by addressing it. Therefore, magnetic tape subsystem interfaces do not need to include address registers for indicating the physical position of the information.

In addition to many of the status and control bits required for A/D and magnetic tape subsystems, interfaces associated with disks, magnetic bubble memory, and other types of addressable mass storage systems must include registers for designating the physical location of the information. For example, to access a disk a surface, track, and sector may need to be specified. Consequently, this information would have to be specified along with the other control information during the initialization phase of the block transfer.

Although the above discussion gives a brief introduction to what is involved in communicating with mass storage and high-speed devices, a detailed examination of these devices and their interfaces is delayed until Chap. 9.

6-5 I/O DESIGN EXAMPLE

To bring together some of the ideas presented in the preceding sections, let us consider a reasonably complex example. Suppose that data are to be continuously input to memory from an A/D converter, processed, and then output to a disk. To accomplish this, block transfers are to be used in conjunction with triple buffering. Both the disk interface and the A/D interface are to be connected to the same

DMA controller. A diagram of the hardware system is given in Fig. 6-20. To simplify the discussion it is assumed that all three buffers are in the lower 64K segment of memory so that the controller needs only a 16-bit address.

The required program is broken into tasks and the relationships among these tasks are shown in Fig. 6-21. There are a main program, an interrupt routine, and five procedures. Four of the procedures are called by the main program and one of them, the input procedure, is also called by the interrupt routine. The fifth procedure, ERR_ROUT, is called when an error occurs in any one of the other components of the program. The procedure INPUT is to initiate a block input transfer from the A/D subsystem, PROCESS is to perform the processing, OUTPUT is to initiate a block output transfer to the disk, and OUTCOMP is to do the necessary completion tasks after each block output. An interrupt occurs whenever an input transfer is finished, but not after an output transfer. The interrupt routine calls the procedure INPUT to start a new input transfer, provided that a new one is needed, and then performs the completion tasks for the input transfer that has just been completed.

A flowchart of the main program is shown in Fig. 6-22 and a summary of the important variables is given in Fig. 6-23. The main program first inputs from a terminal the number of blocks of data to be input and the number of bytes in each block. These quantities are put in COUNT and BUFF_SIZE, respectively. It then decrements COUNT, adjusts the necessary control bits in the DMA controller, including bits for allowing interrupts at the end of transfers from the A/D subsystem and disallowing them at the end of transfers to the disk, and starts the first input transfer by calling INPUT. Next, it enters a loop which inputs, processes, and outputs successive blocks of data by inputting to one memory buffer while processing another buffer and outputting from a third. The first locations in the three memory buffers have been given the names BUFF1, BUFF2, and BUFF3. The loop is exited after the COUNT is decremented to 0 and the blocks that have been input have been processed and output. Following the loop the procedure OUTCOMP completes the final output and the termination processing is performed.

As is often the case, special steps must be taken when starting and terminating a process. In order to skip the output completion procedure the first time around the loop (i.e., before the first output is initiated) a first-time flag FLAGF is used. FLAGF is preassigned to 0 and is set to 1 by the procedure OUTPUT after it begins the first output transfer. After the final input is initiated the loop must be repeated until all of the blocks have been processed and output. To make certain that this is done, a variable FLAGL is used to control the loop. FLAGL is incremented each time an input is begun and is decremented at the beginning of each output. Because FLAGL can only return to 0 after all of the inputting and processing is finished and the last output has started, the loop is repeated as long as FLAGL is nonzero and is exited when FLAGL returns to zero.

Each time through the loop the program must check the next block to be processed to make sure that its input is complete, and then process and begin the output of the next buffer. Before starting the output, however, the program must determine whether or not the disk is still busy with the previous output. If the disk

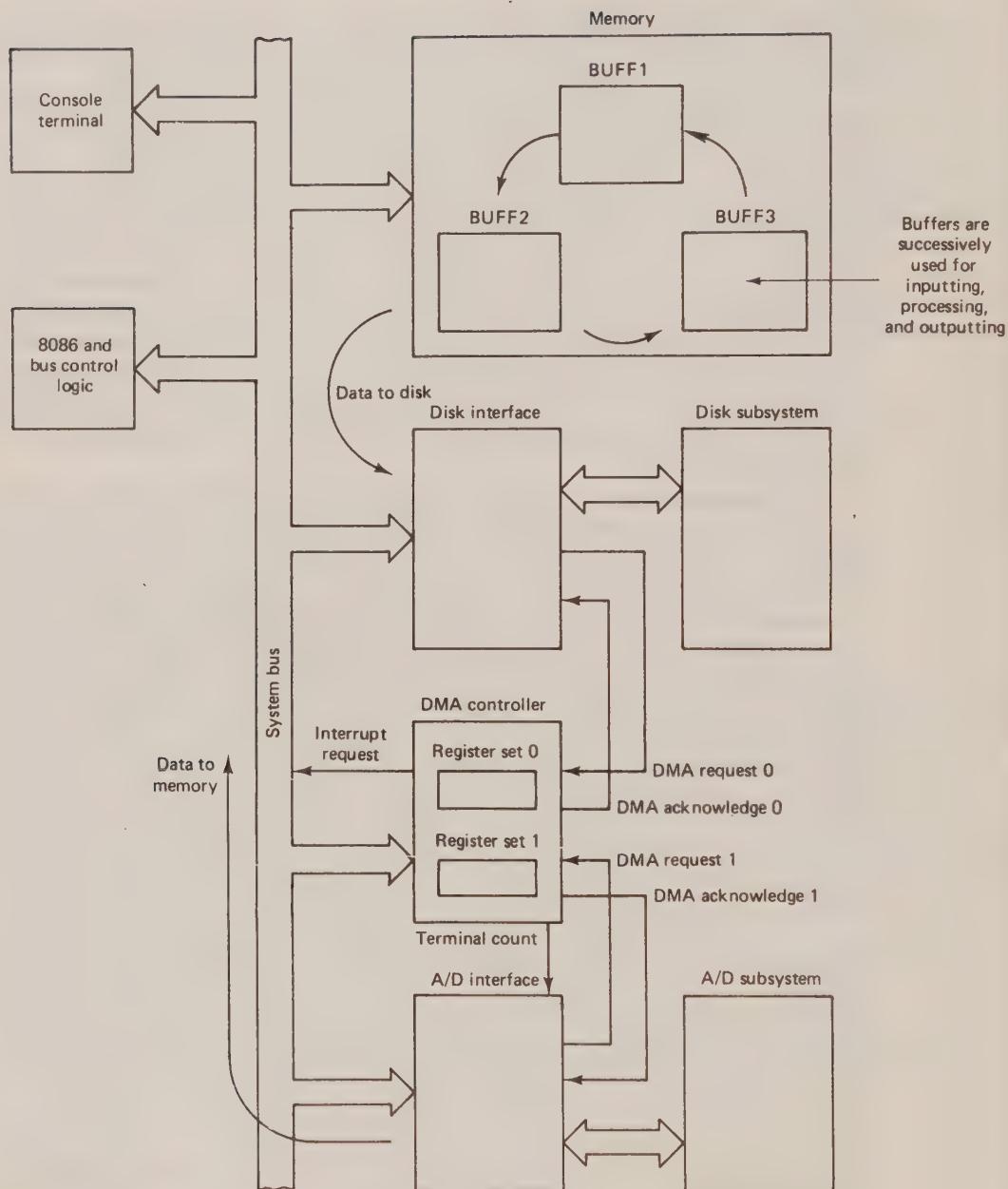


Figure 6-20 Hardware configuration for a triple buffering example.

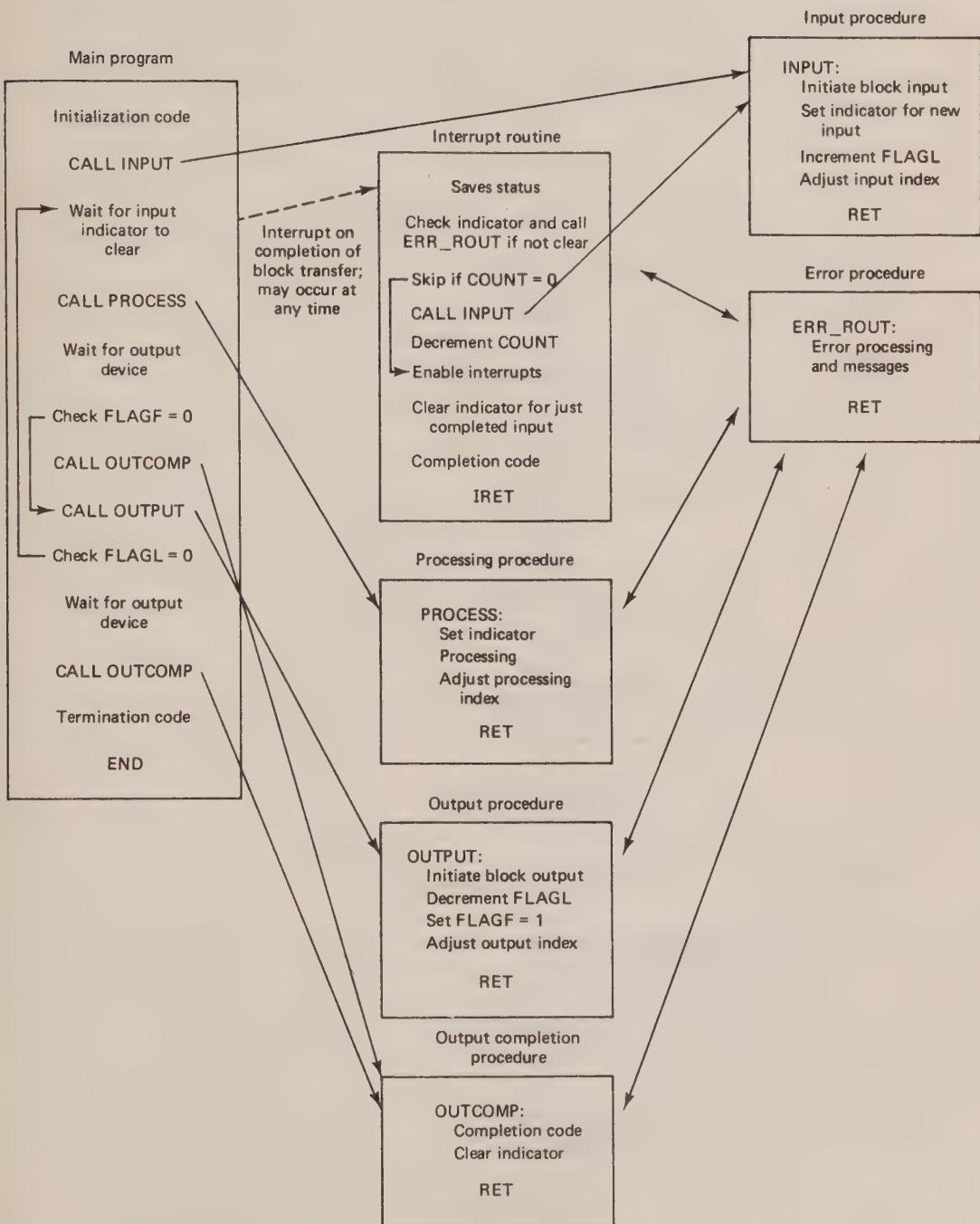


Figure 6-21 Diagram of component relationships.

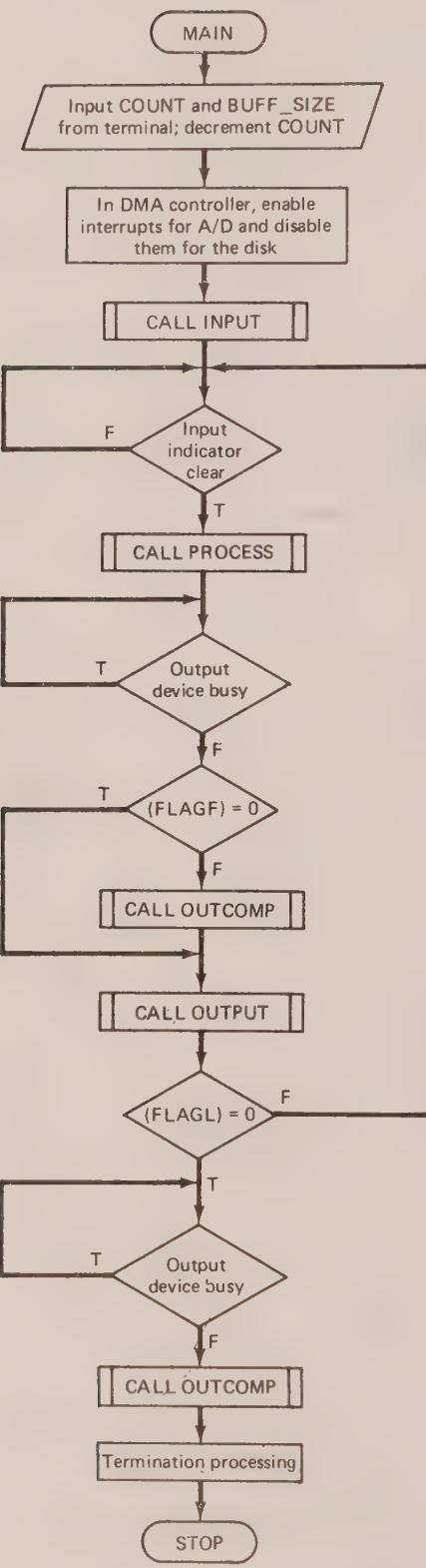


Figure 6-22 Flowchart of the main program.

Preassignment

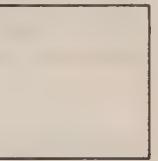
| | | |
|----------------|--|---|
| COUNT | ? | No. of blocks to be processed |
| BUFF_SIZE | ? | No. of bytes in each block |
| FLAGF | 0 | First-time flag |
| FLAGL | 0 | Last-time flag |
| ERR_CODE | 0 | Error code for determining action by error routine |
| INDEX_I | 0 | Index to BUFF_TAB for inputting |
| INDEX_P | 0 | Index to BUFF_TAB for processing |
| INDEX_O | 0 | Index to BUFF_TAB for outputting |
| Buffer pointer | | Indicator |
| BUFF_TAB | Addr. of BUFF1 | 0 |
| | Addr. of BUFF2 | 0 |
| | Addr. of BUFF3 | 0 |
| Input buffer 1 | | Input buffer 2 |
| BUFF1 |  |  |
| Input buffer 3 | |  |

Figure 6-23 Summary of the important variables for the triple buffering example.

is busy, the program has to mark time until it becomes available. A new input is begun each time an interrupt occurs at the end of the previous input; therefore, the input procedure is not explicitly called from within the loop.

To rotate the use of the memory buffers among the inputting, processing, and outputting tasks while guaranteeing that no two tasks will simultaneously use the same buffer, a table beginning at BUFF_TAB is established. The table consists of three pairs of words, with the first word in each pair containing the address of a buffer and the second containing an indicator. The indicator for a buffer is set to 1 when input to the buffer is begun and is reset to 0 upon completion of the input. Also, an indicator is set when the processing of its buffer is begun and is cleared after the buffer is output. Before input to or processing of a buffer is started the indicator is checked and if the buffer is currently being used, further use is

blocked. This prevents the inputting from overrunning the processing and outputting, and vice versa. Note that, unlike the input task, which may be entered through an interrupt that may occur at any time, the processing and output tasks must follow a fixed sequence. This assures that the processing and outputting of the same block of data cannot overlap.

Associated with the table BUFF_TAB are the three indices INDEX_I, INDEX_P, and INDEX_O, which point to the pairs in BUFF_TAB. While an input is being initiated, INDEX_I gives the relative position of the address/indicator pair corresponding to the buffer to which the input is to be made. It is adjusted by 4 immediately after an input is started so that it will be correctly set for the next input. Likewise, the INDEX_P and INDEX_O indices are to point to the currently employed processing and output buffer pairs. These pointers rotate as the buffer usage rotates.

The input and output transfers are followed by sets of completion code which check for errors that may have occurred during the previous transfer and ready the system for the next input or output. Errors are detected by examining the status bits in the proper interface. Associated with each type of error is a unique code, and if an error is found, its code is put in ERR_CODE and a call to ERR_ROUT is made. ERR_ROUT will process the error and return, or for uncorrectable errors, called *fatal errors*, will cause the program to terminate. For example, the A/D subsystem must be able to input data continuously. Because the interrupt sequence and the processing that must take place before the next input can be started takes time, it is possible for data to be missed during the transition between buffers. This could normally be detected by examining certain status bits in the A/D interface after the new input is started. If data have been missed a unique integer, say 3, would be put in ERR_CODE. The procedure ERR_ROUT would determine the integer in ERR_CODE and, upon finding a 3, could branch to code that causes

DATA MISSED IN BLOCK (No. of block)

to be printed on the terminal.

Detailed flowcharts of the interrupt routine and the procedure INPUT are given in Figs. 6-24 and 6-25. When an interrupt occurs, the status of the A/D subsystem is input from the A/D interface. To minimize the chance of missing data, this status is not checked immediately, but is pushed onto the stack and then examined after the next input transfer is begun. At the time INPUT, PROCESS, and OUTPUT are entered it is assumed that they expect the BX register to contain the address of BUFF_TAB and the SI register to contain the appropriate index. Therefore, the interrupt routine or main program must fill these registers before it calls INPUT. Also, before calling INPUT the indicator for the next input buffer and the value of COUNT are tested. If the indicator is set, a -1 is put in ERR_CODE and ERR_ROUT is called. ERR_ROUT prints out a message and sets COUNT to 0. COUNT is tested after the indicator and, consequently, there is no more input if either the next buffer is still being processed or output, or COUNT has decremented to 0 naturally (i.e., the intended number of blocks have been input). If the COUNT is nonzero, INPUT is called and COUNT is decremented. An STI

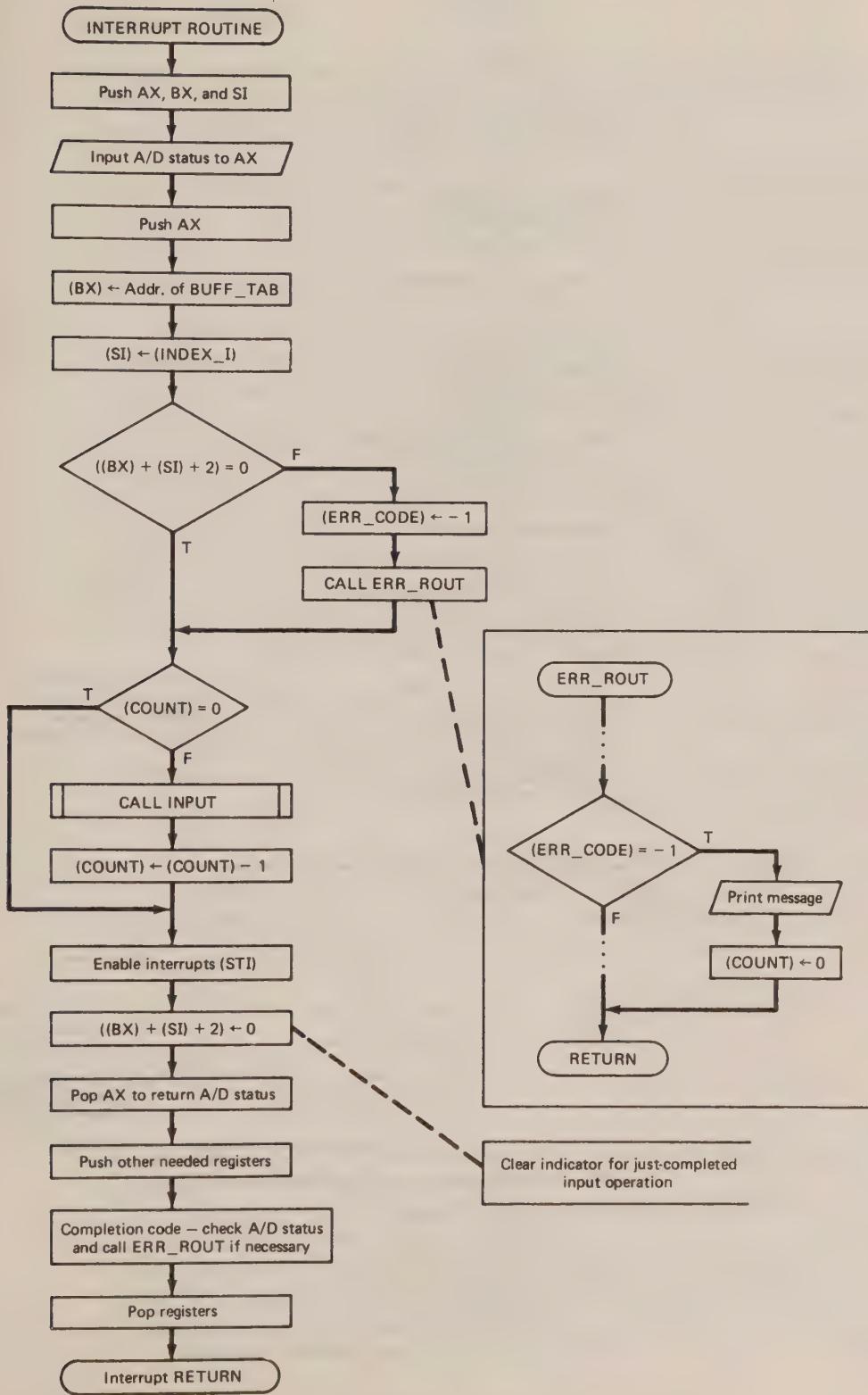


Figure 6-24 Flowchart of the interrupt routine.

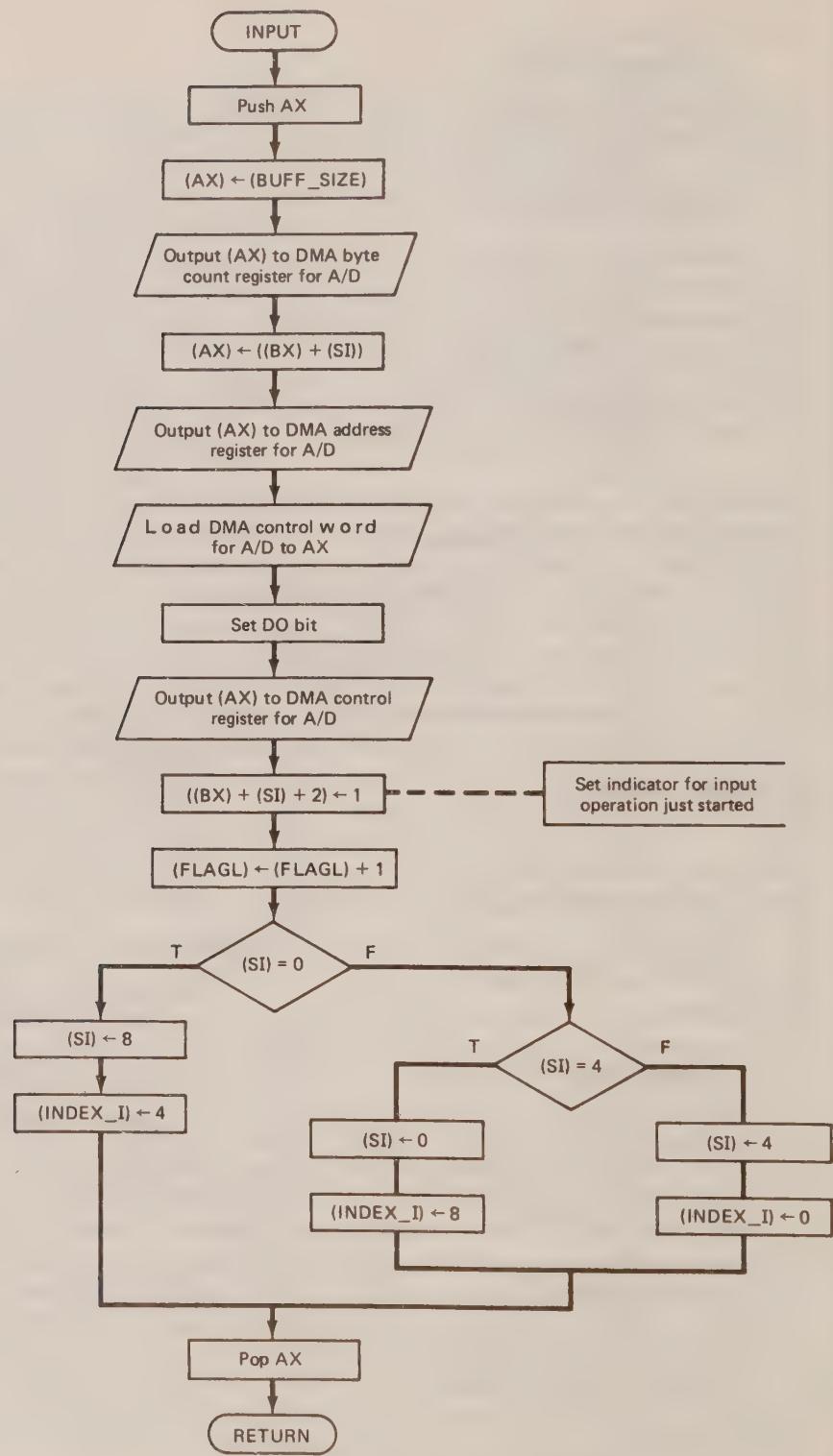


Figure 6-25 Flowchart of the INPUT procedure.

instruction then enables the interrupts and the indicator for the previous input is cleared. Last, the status corresponding to the previous input is examined by the completion code and an IRET instruction is executed.

The procedure INPUT puts the contents of BUFF_SIZE and the address of the next buffer in the byte count and address registers of register set 1 in the DMA controller, and then sets the do bit that causes the input transfer to begin. Next, it sets the buffer's indicator to 1, increments FLAGL, adjusts INDEX_I to point to the succeeding buffer, and sets register SI so that the interrupt routine can use it to clear the indicator for the buffer that was just filled.

To avoid the loss of data, the amount of processing between the time an interrupt occurs and the do bit is set should be made as small as possible. At the expense of complicating the program and degrading its control structure, the amount of this processing shown in the flowcharts could be reduced. However, even if the input initiation code is incorporated at the very beginning of the interrupt routine, the interrupt sequence and buffer switching times would still limit the maximum sample rate of the A/D converter. If only one buffer were to be filled so that buffer switching were not needed, the sample rate would be primarily limited by the time required to send a datum over the bus. It is seen from this and the previous discussion that timing is exceedingly important and must be studied carefully when interrupts, block transfers, and multiple buffering are involved in a process.

Three cases could be handled by the above program:

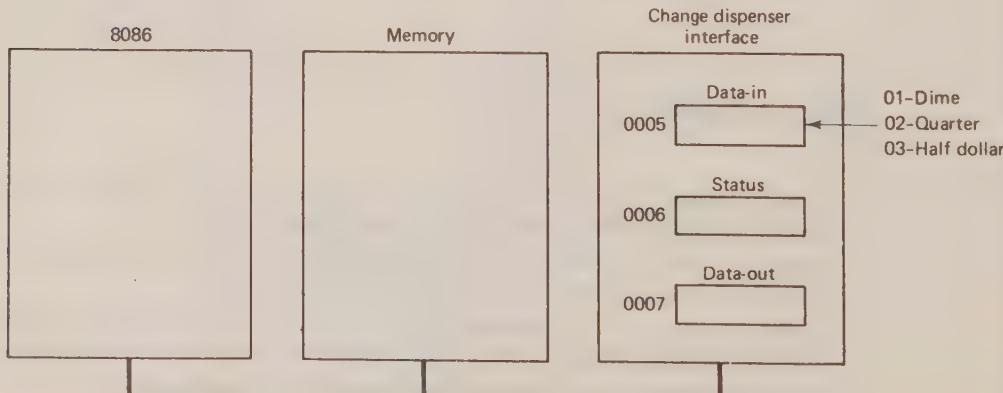
1. Data could be put in a single buffer at a rate limited by the bus cycle time. The processing and output time could be unlimited.
2. Data could be put in a limited number of (at least three) buffers at a rate limited by the buffer switching time. The processing and output times could be unlimited.
3. Any number of blocks could be input at a rate limited by the buffer switching time. Neither the block processing time nor block output time could exceed the block input time.

BIBLIOGRAPHY

1. Gibson, Glenn A., and Yu-cheng Liu, *Microcomputers for Engineers and Scientists* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980).
2. Andrews, Michael, *Programming Microprocessor Interfaces for Control and Instrumentation* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982).
3. Morse, Stephen P., *The 8086 Primer: An Introduction to Its Architecture, System Design, and Programming* (Rochelle Park, N.J.: Hayden Book Company, Inc., 1980).
4. 1982 *Data Components Catalog* (Santa Clara, Calif.: Intel Corporation, 1982).

EXERCISES

1. Draw a flowchart showing how a block of N bytes are input to memory using programmed I/O. Using a block transfer.
2. Give the machine language code for each of the following instructions:
 - (a) IN AL,52H
 - (b) OUT 0CH,AL
 - (c) OUT DX,AX
 - (d) IN AX,DX
3. Write a program sequence that will alternately test the status registers of two devices. When bit 0 of a status register is found to be 1, then a byte of data is to be brought in from the corresponding device, and if bit 3 of either status register is 1, then the inputting process is to cease. The status registers are to have the port addresses 0024 and 0036 and the corresponding data-in buffer registers are to have the addresses 0026 and 0038. The inputs are to be put into memory buffers beginning at BUFF1 and BUFF2.
4. Parity checking of the incoming bytes is often done by the interface and, if a parity error is found, a status bit is set. For the example in Fig. 6-6, assume that the interface automatically clears the parity bit but that bit 4 of the status register is set if the incoming byte has odd parity. Modify the code in Fig. 6-6 accordingly.
5. Suppose that the line input by the code in Fig. 6-6 is a five-digit address in hexadecimal and the computer is to respond with the hexadecimal contents of the byte at that address. Append to the example the code for performing the necessary conversions and providing the output. Let the input and output conversions and the output sequence be in separate NEAR procedures.
6. The software priority scheme built into the example shown in Fig. 6-7 is inflexible because the program has no control over the priority of the devices. Suppose that the three-word array STAT_TAB contains the addresses of the status registers and the three-double-word array PROC_TAB contains offsets and segment addresses of the input procedures. The order the addresses appear in these arrays is to determine the priority. Rewrite the code in Fig. 6-7 accordingly.
7. Given the following block diagram of a nickel dispenser, write the software for operating the system using programmed I/O. The program is to wait in an idle loop until a coin, which is to set bit 2 of the status register at port 0006, is detected. Then it is to bring in the indicated code for the coin and, after waiting for bit 3 (the ready bit) of port 0006 to become 1, output the number of nickels to port 0007.



8. Repeat Exercise 7, but this time assume that there is a switch on the dispenser that sets bit 6 of the status register when it is closed. If the switch is open, the change is to be dispensed as before. If it is closed, a quarter is to cause the input port to be set to 12_{16} and the output port to 21_{16} (which indicates two dimes and a nickel); and a half dollar is to cause the input to be 13_{16} and the output to be 50_{16} (which indicates five dimes).
9. Repeat Exercise 7 using interrupt I/O (i.e., the input of a coin is to result in an interrupt). The main program can be a “do nothing” loop. The interrupt type is to be 10.
10. Given that 0132 is a port that receives its settings from a device that contains two relays, write an interrupt routine that inputs the port to AL and:
 1. Prints “NO EVENT” if neither relay is closed.
 2. Calls PROC_A if relay A is closed and relay B is open.
 3. Calls PROC_B if relay A is open and relay B is closed.
 4. Prints “EMERGENCY” if both relays are closed.The state of relay A is to be indicated by bit 3 of the port and the state of relay B by bit 4. If the bit is 0, then the corresponding relay is open; otherwise, it is closed. The ready bit for the printer is to be bit 4 of the port 0096 and the data-out buffer register for the printer is to have the port address 0098. First use programmed I/O to do the printing from within the interrupt routine, and then modify the code to use a second interrupt routine.
11. Write an interrupt routine that includes a software priority scheme. The routine is to handle up to eight devices and is to contain an array ADDR_TAB of eight addresses. The i th element in the array is to provide the offset of the NEAR processing routine of the i th device. Before checking the status registers, the base address of the array is to be put in BX and SI is to be set to 0. Each time a ready bit is found to be 0, SI is to be incremented by 2. When a ready bit containing a 1 is detected, an indirect branch using the BX and SI registers is to be made to the appropriate processing routine. The code in the processing routines does not need to be included.
12. Suppose that the type 9 interrupt routine whose beginning address is labeled INT_ROUT is included in the same source module as the main program, i.e., the program that may be interrupted. Write a program sequence to be included in the main program that will properly initialize the interrupt pointer.
13. Implement the code for LINE_PROC, which is flowcharted in Fig. 6-13, using the 8086 assembler instructions.
14. If devices 0, 1, 2, 3, 4, and 5 are connected along a daisy chain in that order, and interrupts arrive as listed below, give the order in which the interrupt routines will be completed. All interrupt routines are to begin with an STI instruction.
 1. Devices 3 and 4 simultaneously send interrupt requests.
 2. Device 5 sends an interrupt request before the interrupt routine for device 4 is finished.
 3. Device 2 sends an interrupt request after a return is made to the interrupt routine for device 3, but before this routine is finished.
 4. While the interrupt routine for device 2 is executing, device 3 sends another request.
 5. After a return is made to the main program, devices 1, 3, and 5 make simultaneous interrupt requests.
15. Repeat Exercise 14 but this time assume that there are no STI instructions in the interrupt routines and that the IRET instructions enabled the IF flag by popping the PSW.

16. Repeat Exercises 14 and 15, but this time assume the interrupt priority management system in Fig. 6-15 instead of a daisy chain. First assume that no interrupts are blocked by the mask register and then that requests from devices 0, 1, and 4 are blocked until immediately after step 5 in Exercise 14.
17. Given that a block transfer of 256 bytes from a magnetic tape unit to a memory buffer beginning at 12A0 has been initialized, give a step-by-step account of the actions taken during the input of the first 3 bytes. For each step, note the contents of the byte count and address registers after the step is complete.
18. Consider an interface/controller combination for a magnetic tape unit with the following control and status definitions:

Interface Control Register (0032)

Bit 0—Do bit.

Bit 4—Set for input, clear for output.

Bit 6—Enables interrupts on completion of transfer when set.

Interface Status Register (0033)

Bit 4—Set when tape unit is busy.

Bit 5—Set when parity error is detected during a transfer.

Bit 6—Set when an end of tape is detected during a transfer.

DMA Control Register (0034)

Bit 1—Set for input, clear for output.

Bit 2—DMA requests are enabled when set.

Bit 3—Interrupt on completion is sent to interface when set.

DMA Status Register (0035)

Bit 5—Set when transfer is complete.

Byte Count Register—16 bits (0036).

Address Register—16 bits (0038).

Address Port—upper 4 bits of address (003A).

Write a program sequence that initiates a 200-byte output transfer from a memory buffer beginning at MT_OUT to the tape interface such that an interrupt will occur at the end of the transfer. (Remember that the transfer cannot begin until the tape unit is not busy.) Also write a completion routine that branches to PAR_ERR if there has been a parity error and to EOT_ROUT if an end of tape has been detected; otherwise, it is to simply put a 0 in STAT_CODE and return.

19. Implement the flowchart of the main program for the triple buffering example shown in Fig. 6-22 using the 8086 instruction set. Assume that COUNT and BUFF_SIZE are input using a procedure TERM_IN. Bit 4 of the DMA control register, whose address is 0064, is to control A/D block completion interrupts, and bit 5 is to serve the same purpose for disk interrupts. These bits are set when the corresponding interrupts are to be enabled. Bit 2 of the disk interface status register, whose address is 0080, is set when the disk is busy. The termination processing is to consist of printing the message

OPERATION COMPLETE

All of the software components are to be assembled together so that no parameter passing is needed, but at the time INPUT, PROCESS, or OUTPUT is entered BX is to contain the address of BUFF_TAB and SI the appropriate index to BUFF_TAB.

20. Assume that the addresses of the byte count and address registers in register set 1 of the DMA controller are 0048 and 004A, respectively, and bit 0 of the A/D interface control register, whose address is 006C, is the do bit for A/D input. Write the procedure INPUT flowcharted in Fig. 6-25.
21. Suppose that in the triple buffering example in Sec. 6-5 the processing is to consist of replacing each byte of data with the difference between it and the previous byte of data, and write the procedure PROCESS. For the first byte of data in the first block the previous byte is to be taken to be 0. If at any time the difference falls outside the range -128 to 127, then ERR_CODE is to be set to 2 and ERR_ROUT is to be called.

7

Introduction to Multiprogramming

A *process* is a programming unit which performs an independent task. When processes are executed in a serial fashion, the system is called a *uniprogramming* system. In such systems, normally only one process is stored in the memory at a time and the next process is not loaded for execution until the current one is terminated. This is not adequate for a large number of applications, those in which the ability to respond to events in real time is required or which must be able to execute numerous programs rapidly and efficiently. For example, suppose that a microprocessor-based system is to collect and process data from two independent data acquisition devices. If operated in a uniprogramming environment, either process may miss some of its input data while the other process is executing. This can happen even when the incoming data rate for both processes is low. The problem is not caused by the lack of processing power or by the speed of the interfaces, but is a result of the sequential nature of the overall processing scheme. In addition, for a medium-size to large system, serial execution of a set of processes could waste much of the system's time, particularly if the design involves a high-performance microprocessor.

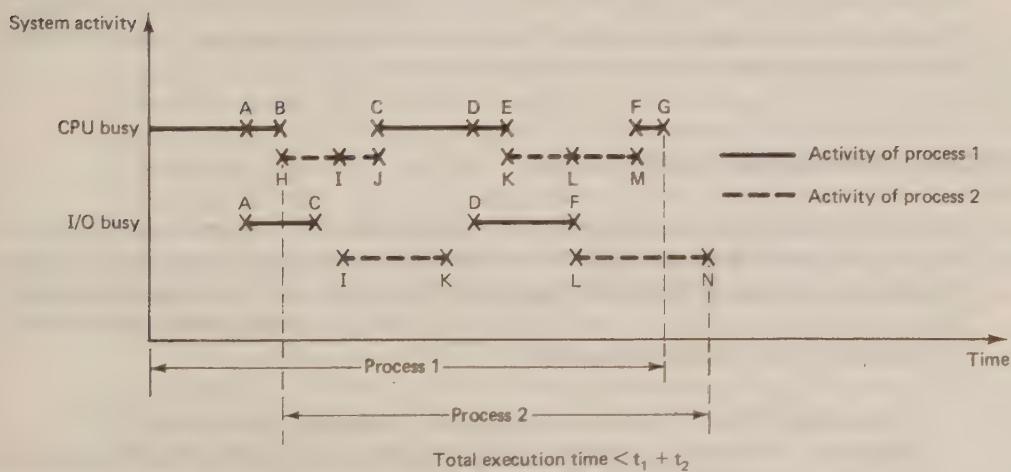
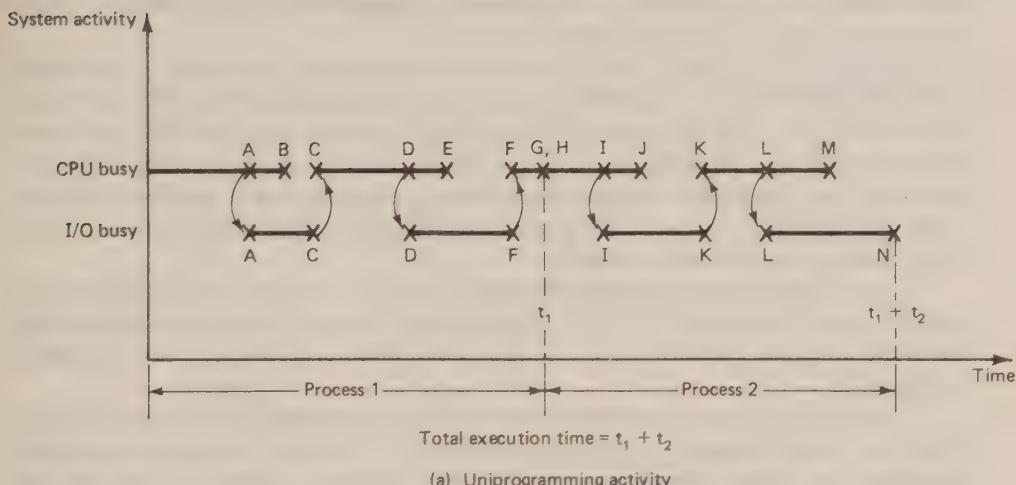
In contrast, for a *multiprogramming* environment, the code for two or more processes is in memory at the same time and is executed in a time-multiplexed fashion. If, in the above example, the processes were executed in a multiprogrammed system, they could take turns using the CPU and one process could be performing its computations while the other is doing I/O, and vice versa. Assuming reasonable data rates, the switching back and forth between the processes could be done using interrupts and both processes could execute satisfactorily.

Even though multiprogramming involves the interleaving of more than one process by a CPU, a single CPU can still execute only one instruction at a time.

However, through resource sharing a multiprogrammed single CPU system appears as if it is executing several processes simultaneously. Even for general purpose applications which do not involve real-time data, multiprogramming is desirable because it can significantly improve the system performance by overlapping the I/O processing and the CPU operations.

The performance of a system is often measured by the number of jobs finished in a period of time, a quantity that is referred to as the *system throughput*. To demonstrate how overlapping processing and DMA I/O can improve system throughput, let us consider two processes, say 1 and 2. If uniprocessing is used,

Figure 7-1 System performance of a single CPU system.



(b) Multiprogramming activity utilizing a single CPU and a single DMA channel

the activity is typically as shown in Fig. 7-1(a). Here process 1 begins and continues until I/O is needed (point A), then the I/O is initiated and the processing continues in parallel with the I/O until the processing needs the input data. At this time it must wait until the I/O is completed (point B); When the I/O is finished (point C) the processing is resumed. A similar description applies to points D, E, and F. At the end of process 1, process 2 can begin and its operation is basically the same as that of process 1.

On the other hand, if multiprogramming is applied, then the activity is as illustrated in Fig. 7-1(b). It is seen that instead of the CPU being idle while it is waiting for process 1 I/O, in a multiprogramming system process 2 can be begun and can utilize the CPU until it needs to wait for I/O. At that time, if process 1 has finished its I/O operation, it can resume its use of the CPU. Thus the overall processing time can be significantly reduced.

In addition to being able to intersperse several I/O activities, a multiprogramming system may be capable of accommodating several users at the same time. When this is the case the users take turns getting control of the CPU, with each user being allocated a *time slice*, and the system is said to be *time-shared*. Typically, time-sharing systems are connected to numerous terminals and each user interacts with the system as if it were dedicated to him or her (although too many users may cause noticeable time delays).

As a further extension of multiprogramming, it can be used in systems that include more than one processor. Such systems, called *multiprocessing systems*, have the ability to execute more than one instruction at a time and are considered in Chap. 11.

This chapter is a discussion of some of the major programming considerations that arise when multiprogramming a single CPU system. Although this chapter confines itself to an introduction to multiprogramming, several of the more important topics and how they relate to 8086/8088 systems are discussed in detail. Process management is discussed in Sec. 7-1, the accessing of shared data in Sec. 7-2, and procedure sharing in Sec. 7-3. The last two sections are concerned with memory management and the use of virtual memory.

7-1 PROCESS MANAGEMENT AND IRMX 86

In a single-processor multiprogramming system, two or more processes reside in the memory and share the CPU, but the CPU can execute only one of these processes at a time. In a simple multiprogramming system there are three states that the processes can be in, with each process being in exactly one of these states at any given time. These states are:

1. **Running**—When the process is currently being executed by the CPU.
2. **Blocked**—When the execution of the process cannot be continued because it is waiting for an event to occur, e.g., it is waiting for the completion of an I/O operation.

3. Ready—When the execution of the process can be resumed any time. For example, the I/O the process has been waiting for has finished and the processing is able to continue.

As time passes each process will rotate among these states as shown in Fig. 7-2. At the time a process is started it is placed in a waiting queue of ready processes. When the process that is currently executing switches from the running state to the blocked state because it must wait for I/O, the CPU becomes available and a process in the ready queue is selected by the process scheduler and is changed from ready to running. While a process is in the blocked state its execution is suspended. After its I/O is completed, a blocked process becomes ready and it is placed in the ready queue.

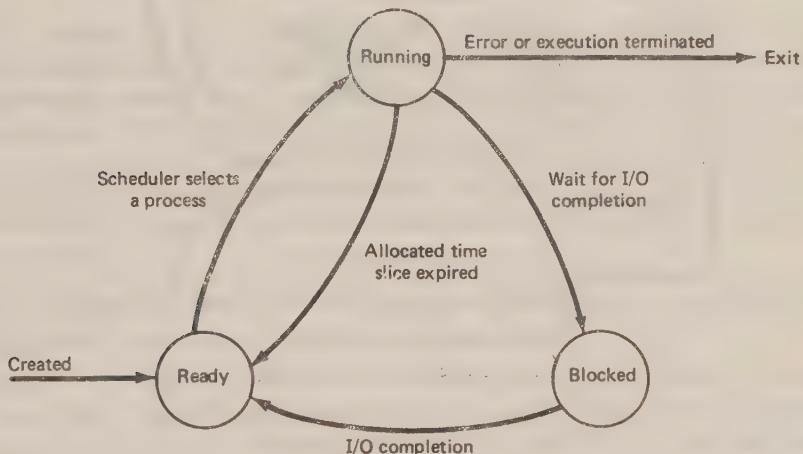


Figure 7-2 Process states and state changes.

In the simplest multiprogramming arrangement a process is allowed to execute until it is terminated or must wait for I/O. However, to prevent a process from dominating the CPU usage, a system can be designed so that a running process must return to the ready state after a specified period of time, thus giving the other processes a chance to execute. These are time-sharing systems and are regulated by programmable external timing devices known as *real-time clocks* (see Sec. 9-3). In a time-sharing system, time is measured from the beginning of the running state and the real-time clock sends an interrupt when the allotted time, i.e. time slice, is used up. The interrupt routine, which is part of the resident monitor, then initiates the change of states.

Implementation of the ready queue varies according to the scheduling policy. The easiest such policy to design into a system is the first-in/first-out (FIFO) policy, which gives each process equal priority and always selects the process that has been in the queue the longest. In effect, processes are added to the bottom of the queue and taken from the top. A structure of a ready list that is based on FIFO scheduling and a linked list is shown in Fig. 7-3. The first-element pointer indicates which

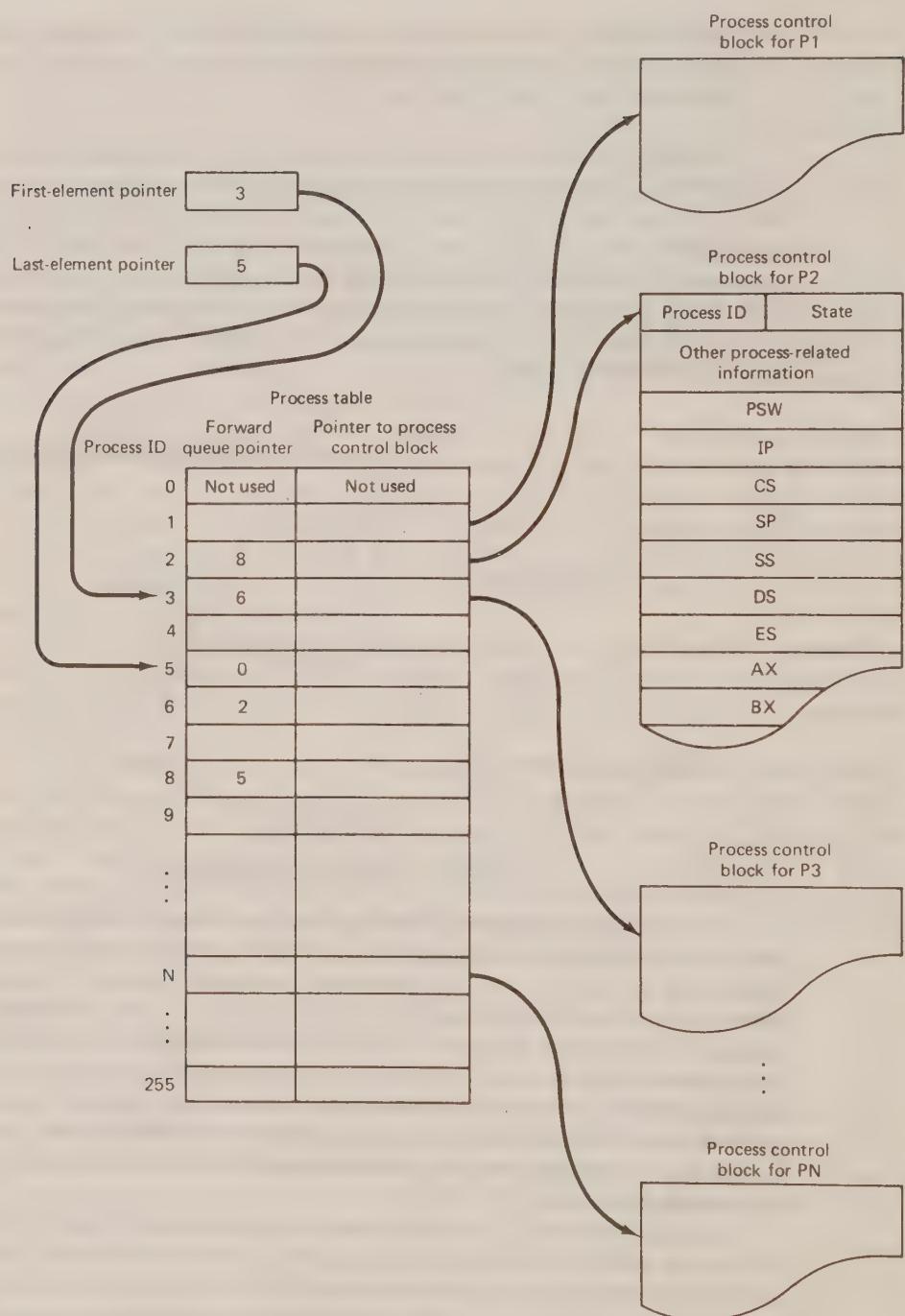


Figure 7-3 Structure of a ready queue.

process is at the top of the queue and the last-element pointer indicates which process is at the bottom. The last-element pointer is needed so that new processes can be added to the queue without having to search through the entire list. In the example, it is assumed that each process has an ID that is the same as a position in the array containing the linked list. It is also assumed that the queue presently consists of the processes 3, 6, 2, 8, and 5, in that order. Therefore, 3 is in the first-element pointer, 5 is in the last-element pointer, and the processes 2, 3, 5, 6, and 8 are currently in the ready state. The remaining entries in the process table are occupied by running and blocked processes or are not currently being used. If the number 0 is used to indicate the end of the list, it cannot be used as a process ID. If the process IDs are to be represented by 1 byte and there can be no process 0, then the maximum number of processes that can be in the system at one time is 255.

Each element in the list contains a forward queue pointer (fqp), which indicates the next element in the list, and a process control block pointer for locating a block of memory that is used to store important information related to the process. The process control block serves as the process's save area for storing the machine status as well as for storing such things as the process ID, the process state, a code indicating how and why the process entered its current state, and so on.

When the CPU switches from one running process to another, it is necessary for the monitor to:

1. Save the machine status of the running process in the process control block.
2. Update the remainder of the process control block.
3. Get the ID of the next process to be run from the first-element pointer.
4. Delete the process that has just entered the running state by setting the first-element pointer equal to the fqp of the deleted element. If the current running program is to enter the ready queue, it should be inserted at the bottom of the queue, and the last-element pointer must also be changed.
5. Change the state of the process that was just removed from the ready list to running and restore the machine status of this process.

As a result of step 5 the newly selected process will continue from the point at which it was suspended. A process that is currently in the system can be added to the ready list by:

1. Storing its process ID into the fqp of the current last element (the element pointed to by the last-element pointer) and into the last-element pointer.
2. Clearing the fqp in its process table entry.
3. Updating its process control block.

For many applications, especially those involving real-time data processing, it is necessary to assign priorities to the various processes by giving the process requiring the fastest service the highest priority. Processes may still be given the same priority, with the FIFO policy being applied within each priority level. If

priorities are used, the system must include a priority table. As shown in Fig. 7-4, each element in this table would represent a priority level and have two fields, with the first field containing the first-element pointer for the priority level and the second field containing the last-element pointer for the level. If the priority of a process is changed, it must be deleted from its current priority chain and appended to the bottom of the chain having its new priority. This action is aided by the inclusion of a backward queue pointer (bqp) so that priority changes can be made quickly. The priority scheduling system illustrated in Fig. 7-4 assumes the following ready processes:

| |
|---------------------|
| Priority 0: None |
| Priority 1: 9-4-1-3 |
| Priority 2: 2-7-5 |
| Priority 3: 10 |
| Priority 4: None |
| Priority 5: None |
| . |
| . |
| . |

To select the next running process, the scheduler would search the priority table starting with the highest priority until a nonzero first-element pointer is found, and would then delete that process from the ready queue.

The advantage of using the bqp is the ability to change the priority of a given process dynamically. Suppose that the priority of process i is to be changed to priority x . First, process i is deleted from its current priority chain with the following operations:

$$\begin{aligned} \text{fqp(bqp}(i)\text{)} &\leftarrow \text{fqp}(i) \\ \text{bqp(fqp}(i)\text{)} &\leftarrow \text{bqp}(i) \end{aligned}$$

(Of course, if process i is the top or bottom element in its current priority chain, additional adjustment to the priority table is required.) Next, process i is added to the bottom of the chain corresponding to priority level x . This requires the updating of the last-element pointer (and the first-element pointer if the chain was empty), the fqp of the current last element in the chain, and the fqp and bqp of the element being added.

Some multiprogramming operating systems are more complex than the one depicted in Fig. 7-2, and include more than the three basic states. When this is the case, the additional states are normally alternate forms of the blocked state. As an example, let us consider Intel's iRMX 86 operating system, whose states are summarized in Fig. 7-5. In iRMX 86 there are three blocked states: asleep, suspended, and asleep-suspended. iRMX 86 is purely an event-driven system and its only time dependence is due to the fact that a process can be put to sleep for a specified period of time. State changes can be caused by program commands, which are referred to as *system calls* and are implemented using software interrupts, or

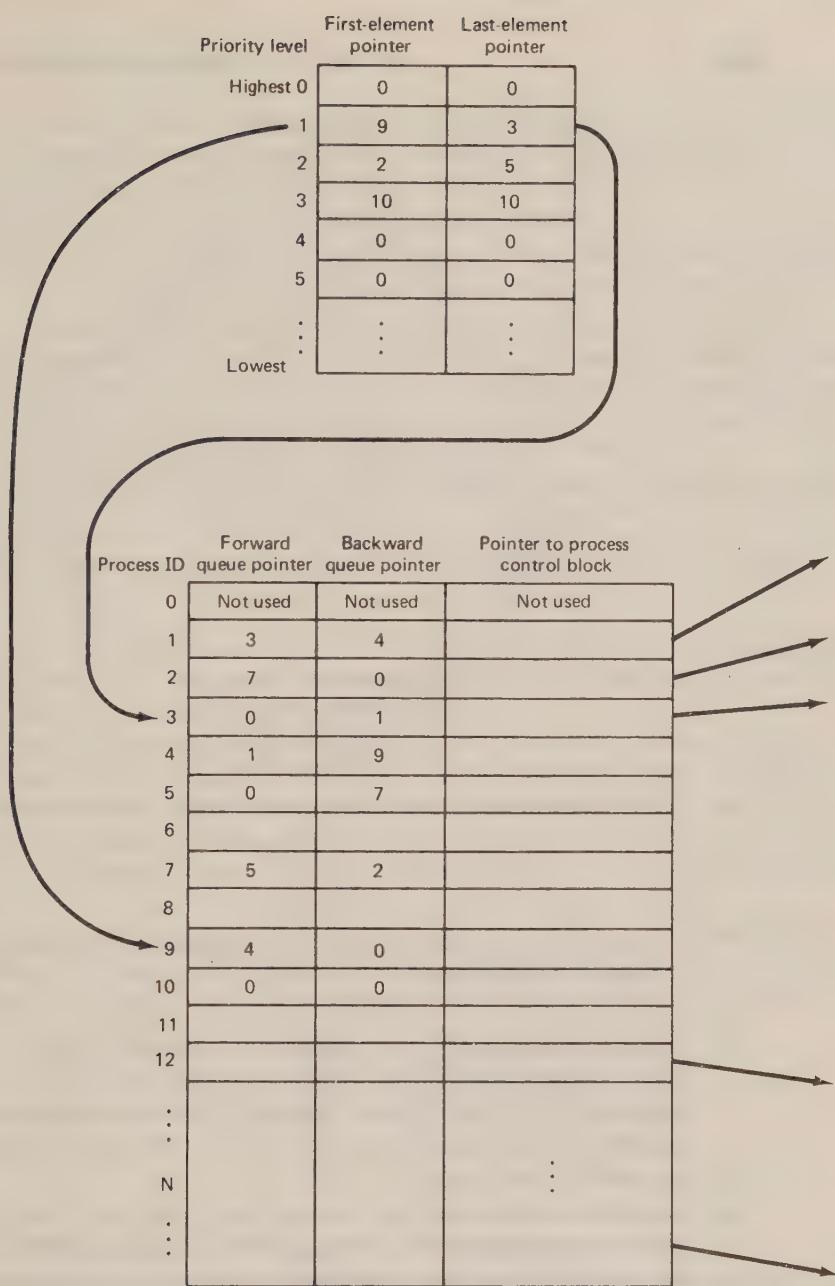


Figure 7-4 Structure of a prioritized ready queue.

automatically following certain events. The possible changes are indicated by the arrows and are defined as follows, where the arrow numbers in the figure correspond to the following statement numbers:

1. A process enters the system when it is created by a CREATE TASK system call.

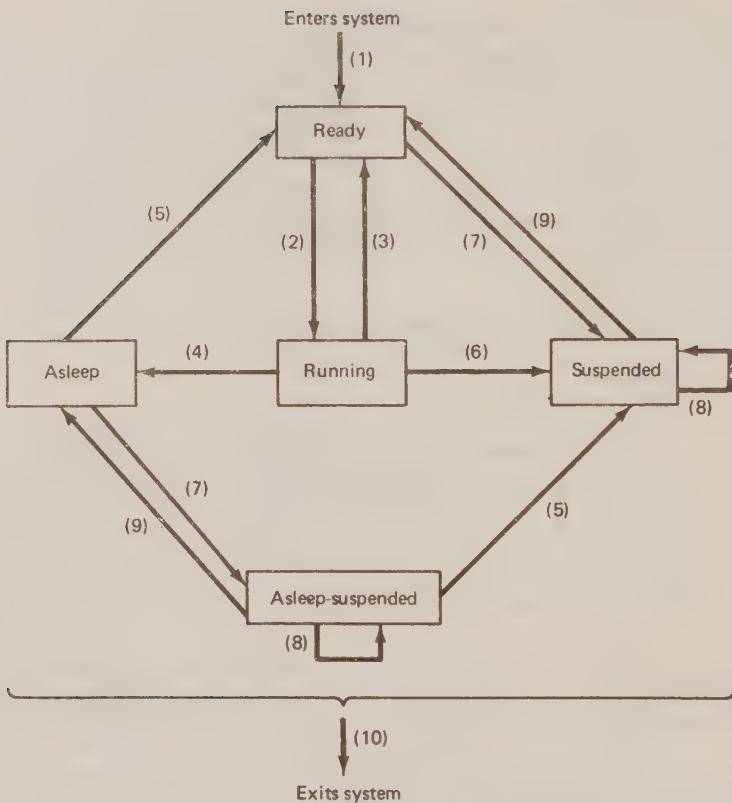


Figure 7-5 Process states for iRMX 86. (Reprinted by permission of Intel Corporation. Copyright 1981.)

2. A ready process enters the running state when its priority is higher than that of the running process, or when the running process is put to sleep or is suspended, the ready process's priority is at least as high as any other ready process, and the ready process has been in the ready queue longer than any process of equal priority.
3. A transfer is made from the running state to the ready state when the running process is preempted by a ready process of higher priority.
4. A transfer is made from the running state to the asleep state if the running process executes a SLEEP system call or must wait for information that it has requested and is not yet available. In the former case, a time limit is specified in the SLEEP command and, in the latter case, the process must express a willingness to wait when the information is requested (via a system call).
5. A process goes from the asleep state to the ready state or from the asleep-suspended state to the suspended state when the waiting period designated in the SLEEP command expires or the process's request is granted.

6. A transfer from the running state to the suspended state occurs when the running process suspends itself by executing a SUSPEND TASK system call. Also, the suspension depth for the process is set to 1.
7. A process is moved from the ready state to the suspended state or from the asleep state to the asleep-suspended state when another process specifies it in a SUSPEND TASK system call. Also, its suspension depth is set to 1.
8. The state of a suspended or asleep-suspended process remains unchanged but its suspension depth is incremented when the running process executes a SUSPEND TASK command which specifies the process. Similarly, the suspension depth of a suspended process is decremented when the running process executes a RESUME TASK command.
9. A suspended or asleep-suspended process goes to the ready or asleep state, respectively, if its suspension depth becomes 0.
10. A process exits the system if it is designated in a DELETE TASK system call.

Note that in iRMX 86, not only do the processes in the ready state have a priority, but also those in the suspended and asleep-suspended states are assigned a form of priority according to their suspension depths.

Figure 7-6 shows how the execution of the processes 1, 2, and 3 might proceed under the iRMX 86 operating system. All three processes are assumed to be in the system, with process 1 having the highest priority and 2 and 3 having the same, but lower, priority. It is also assumed that, initially, process 2 has been in the ready state longer than process 3. The solid lines indicate which process is in the running state.

As indicated above, the iRMX 86 system calls are implemented through software interrupts and the parameters are passed to and from the called interrupt routine by means of the stack. A program sequence for deleting a task is:

| | |
|------|----------------|
| PUSH | TASK_NO |
| PUSH | EXCPT_PTR |
| MOV | BP,SP |
| LEA | SI,SS:[BP + 2] |
| MOV | AX,201H |
| INT | 184 |

where 201H and 184 are the entry code and interrupt type corresponding to the DELETE TASK system call, respectively. The 184 designates the interrupt routine to be executed and the entry code 201H indicates to that routine that a delete is to be performed. TASK_NO is assumed to contain the number of the process to be deleted and EXCPT_PTR is to contain a pointer to a routine that is to be executed if an exception occurs while the delete is being made. BP and SI are to be used by the interrupt routine to access the stack.

Although the iRMX 86 system calls can be implemented in assembler language, Intel has also designed the system calls into its high-level PL/M and Pascal

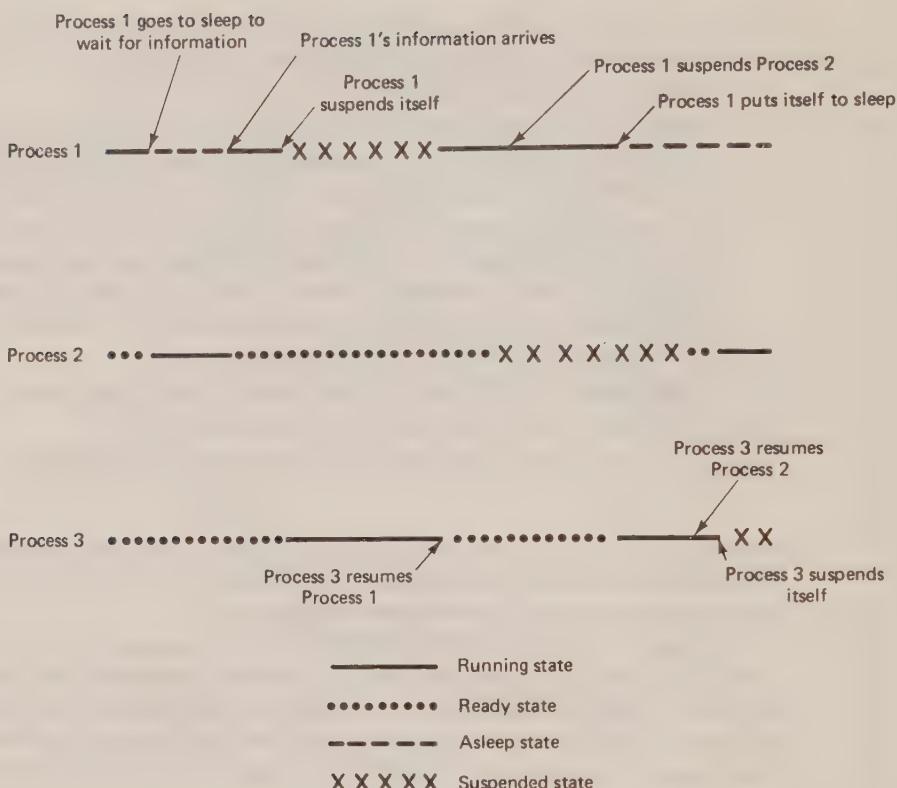


Figure 7-6 Typical activity under the iRMX 86 operating system.

languages. To facilitate the use of iRMX 86, Intel has designed a chip, the 80130, which incorporates much of the supporting hardware that is needed and a 16K-byte block of ROM that contains the code for the system calls. The 80130 is considered further in Chap. 12.

7-2 SEMAPHORE OPERATIONS

In multiprogramming systems, processes are allowed to share common software and data resources as well as hardware resources. In many situations, a common resource may be accessed and updated by only one process at a time and other processes must wait until the one currently using the shared resource is finished with it. A resource of this type, which is commonly referred to as a *serially reusable resource*, must be protected from being simultaneously accessed and modified by two or more processes. A serially reusable resource may be a hardware resource (such as a printer, card reader, or tape drive), a file of data, or a shared memory area.

For example, let us consider a personnel file that is shared by processes 1 and 2. Suppose that process 1 performs insertions, deletions, and changes, and process 2 puts the file in alphabetical order according to last names. If accessed sequentially, this file would either be updated by process 1 and then sorted by process 2, or vice versa. However, if both processes were allowed to access the file at the same time, the results would be unpredictable and almost certainly incorrect. The solution to this problem is to allow only one process at a time to enter its *critical section of code*, i.e., that section of code that accesses the serially reusable resource.

Preventing two or more processes from simultaneously entering their critical sections for accessing a shared resource is called *mutual exclusion*. One way to attain mutual exclusion is to use flags to indicate when the shared resource is already in use. To examine how this is done and some of the problems associated with this approach let us consider the possibility of using only one flag. If there is only one flag and FLAG=1 indicates that the resource is free and FLAG=0 indicates it is busy, then two processes that access the resource might be implemented as follows:

Process 1

P1:
.
.
TRYAGAIN1: TEST FLAG,1
JZ TRYAGAIN1
MOV FLAG,0
.
.
MOV FLAG,1
.
.

} Critical section in process 1

Process 2

P2:
.
.
TRYAGAIN2: TEST FLAG,1
JZ TRYAGAIN2
MOV FLAG,0
.
.
MOV FLAG,1
.
.

} Critical section in process 2

where the first MOV instructions set the flag to 0 in an attempt to prevent the other process from entering the resource. The last MOV instructions are to reset the FLAG to 1 after the access is complete so that the other process can once again enter the resource.

The problem with the above approach is that it is possible for both processes to enter their critical sections. To see how this can happen, suppose that FLAG = 1 and that the following sequence occurs:

1. The TEST instruction in process 1 is executed.
 2. A switch is made to process 2 before the JZ instruction in process 1 is executed.
 3. The TEST instruction in process 2 is executed.

At this point both processes are free to enter their critical sections of code.

To avoid the above problem two flags can be employed, one which indicates that process 1 is in its critical section and one for indicating that process 2 is in its critical section. Before either process can enter its critical section it would first clear its flag to indicate its intention and then repeatedly test the flag associated with the other process until it becomes 1. This could be implemented by coding both processes as follows:

While this solution does guarantee mutual exclusion, it introduces yet another problem, called *mutual blocking*. If a switch from, say, process 1 to process 2 occurs between the MOV and TEST instructions, then process 2 might clear FLAG2, causing both flags to be 0. Since neither TEST has been performed at that time, neither process will be able to enter its critical section. Both processes will be hung up in their waiting loops.

For a solution to the critical section problem that both provides mutual exclusion and avoids mutual blocking, a third flag is needed that indicates which process has the higher priority when the mutual blocking occurs. This concept was first proposed by Dekker and was discussed by Dijkstra [6]. Dijkstra has also extended this solution to the n -process case.

When using an 8086 or 8088, a simple solution to the problem is provided by

the presence of the XCHG instruction. The required access control can be obtained by using a single flag and structuring the processes as indicated below:

```
P1:
.
.
.
TRYAGAIN1:    MOV      AL,0
                XCHG    AL,FLAG
                TEST   AL,AL
                JZ     TRYAGAIN1
.
.
.
MOV      FLAG,1
.
.
.
```

{ Critical section

If FLAG = 1 when the XCHG instruction is executed, FLAG is cleared and AL is set to 1. Even if the system switches processes after the XCHG instruction, the new process cannot enter its critical section because FLAG has already been cleared. This solution is attributed to the capability of loading and setting an operand with a single instruction (i.e., the XCHG instruction), and can be applied to any number of processes.

A flag used to reserve a shared resource is called a *semaphore* and the operations of requesting and releasing the resource are commonly known as the *P* and *V* *semaphore operators*. As shown above, the P operator can be easily implemented with the MOV, XCHG, TEST, and JZ instructions, and the V operator with a MOV instruction. However, for this implementation, when one process is in its critical section, other processes that are presently competing for the same resource will be idling in the wait loops associated with their P operators. This may cause a significant amount of CPU time to be wasted. To better utilize the CPU time, the P operator could be modified as follows:

```
AGAIN:    MOV      AL,0
          XCHG    AL,SEMAPHORE
          TEST   AL,AL
          JNZ    NEXT
.
.
.
          { Call the monitor in such a way
          .           that it will put the current
          .           process into the blocked state
NEXT:     JMP      SHORT AGAIN
.
.
.
          { Critical section
```

Therefore, if the semaphore is cleared, instead of repeating the wait loop, the

monitor changes the state of the current process to blocked and switches a ready process to running. Similarly, the V operator should be modified to be:

| | | |
|-----|---|---|
| MOV | SEMAPHORE,1 | } |
| . | Call the monitor in such a way that it will | |
| . | unblock those processes that were blocked due to the semaphore | |

This sets the semaphore to 1 and allows the monitor to move those processes which were blocked because the shared resource was unavailable from the blocked list to the ready list. The first of these processes that resumes its execution will be able to enter its critical section.

This method not only takes advantage of a relatively large amount of wait time by using it to execute other processes, but also provides a method for synchronizing two processes or for passing a message from one process to the other. In the example shown in Fig. 7-7, the processes A and B need to interact with each other at certain points as they are executed. The two processes may be executed on a time-shared basis until point PB1 is reached by process B, at which time process B must wait for a message or access results produced by process A. After point PA1, both processes once again enter into a time-sharing mode which continues until PA2 is reached. At this point, process A must wait for a message or be synchronized with process B.

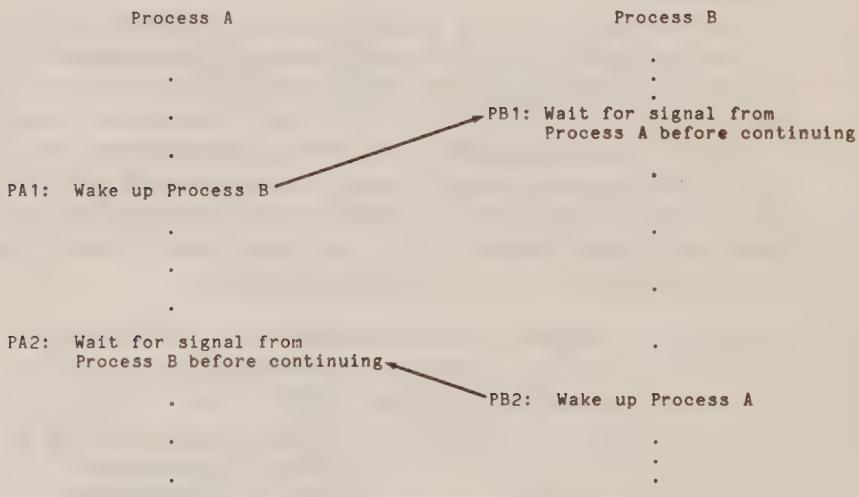


Figure 7-7 Synchronization of two processes.

The P operator could be used to block a process while the V operator could be used to wake up a process. For the above example, two synchronization semaphores, WAKEA and WAKEB, need to be given an initial value of 0. Then, synchronization between the processes could be accomplished as follows:

Process A

PA1: V(WAKEB) ;WAKE PROCESS B

PA2: P(WAKEA) ;WAIT FOR PROCESS B

and

Process B

PB1: P(WAKEB) ;WAIT FOR PROCESS A

PB2: V(WAKEA) ;WAKE PROCESS A

7-3 COMMON PROCEDURE SHARING

In a multiprogramming system, it is desirable to allow two or more users to share procedures. These procedures are known as *common procedures*, and typically are library routines or system programs such as I/O driver, code conversion routines, and so on. For example, several users may simultaneously want to use a text editor to edit their programs. Without code sharing, several copies of the text editor would need to be loaded into memory. With code sharing, only one copy needs to be kept in memory, thus providing a significant savings in storage.

A common procedure may be shared in serial fashion just like any other shared resource, but as with other resources, before a common procedure is called the calling procedure must perform a P operator and after it is exited a V operator must be performed to allow another user to use the shared procedure. With serial sharing the primary additional requirement is that all local variables in the procedure be reinitialized so that the values left over from the previous call are not used. This implies that a serially shared procedure must reinitialize itself at the beginning of each call.

The application of serially shared procedures is rather limited, and a broader form of sharing is necessary, a form in which a procedure is shared in a time-

multiplexed fashion or is concurrently reusable. This means that a procedure must be such that it can be called by another process before the execution of the procedure due to a prior call is completed. Such a procedure is said to be *reentrant*. A reentrant procedure must consist of code, called *pure code*, which does not modify itself. To meet this requirement a reentrant procedure can store the data to be modified by the routine only in memory locations that are associated with the calling process; it cannot, even temporarily, store such data in locations that are local to itself. To see this, suppose that TEMP is a local variable in the reentrant procedure and that it is used to store an intermediate result, and consider the following sequence of events:

1. Process 1 is running and calls the reentrant procedure.
2. The reentrant procedure puts an intermediate result in TEMP.
3. Process 2 gains control of the CPU.
4. Process 2 calls the reentrant procedure.
5. The reentrant procedure again puts an intermediate result in TEMP.

At this point the original intermediate result is destroyed and, when a return is eventually made to process 1 and the reentrant procedure is resumed, an invalid result may be produced for process 1.

To avoid this problem, *all* results including the contents of the registers must be stored in locations that are associated with the calling process. In the above example this would mean that the two intermediate results would be put in separate locations, one associated with process 1 and the other with process 2. The problem encountered here is similar to the one discussed when recursive procedures were considered (see Sec. 4-3-4), except that for the recursive situation new locations that are used for storing intermediate results are allocated for each successive call. The same stack is used for storing multiple copies of intermediate results on a last-in-first-out basis because recursive calls will exit in the reversed order. Here the information is stored back into the calling process.

For a machine that has stack facilities, such as the 8086 and 8088, the usual way of storing intermediate results is to associate a stack with each process. Figure 7-8 illustrates how two processes might access a reentrant procedure. When process 1 calls the reentrant procedure, the SS and SP registers remain unchanged and the reentrant procedure stores its results, intermediate or otherwise, on the stack for process 1. When the processes are switched, the contents of SS and SP are changed to point to the top of the stack associated with process 2. Then when the reentrant procedure is entered it will store its results on the stack associated with process 2, and the previous results will be undisturbed.

Because switching between the procedures also switches the contents of the DS and ES registers, DS and ES could also be used to store information back into the calling process. In fact, any acceptable subroutine linkage could be implemented, the only requirement being that *no* data could be stored locally to the reentrant procedure.

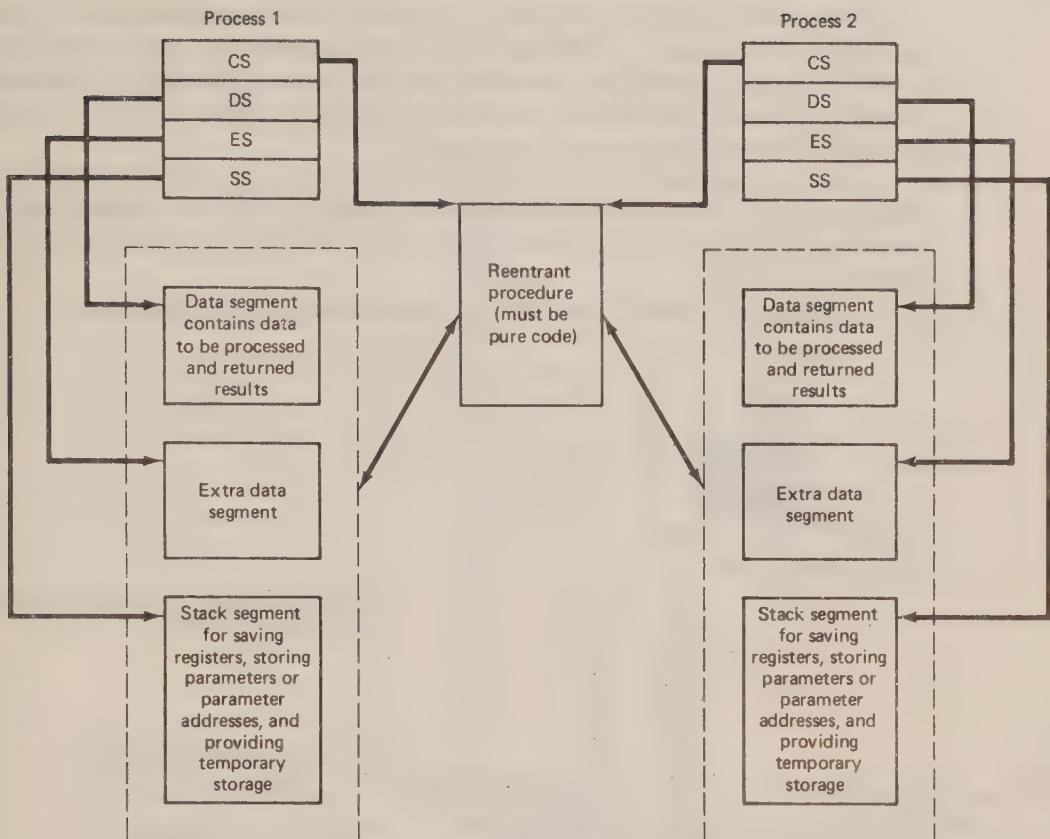


Figure 7-8 Reentrant code shared by more than one process.

To illustrate how to write a reentrant procedure, let us consider one that multiplies two 32-bit signed integers and returns a 64-bit result. Signed multiplication is normally accomplished by first multiplying the absolute values of the two operands and then putting the product into 2's complement form. In order to leave the multiplier and multiplicand unchanged, it is necessary for the procedure to temporarily store their absolute values. It is also necessary to temporarily store part of the unsigned product. A reentrant routine for performing a double-precision multiplication is given in Fig. 7-9. As shown in the figure, six words in the stack are allocated as temporary storage, two words for part of the product and two words for each operand. To allocate and access these words, 12 is subtracted from SP and the result is loaded into BP; then references are made to them relative to BP.

Because, as shown in Fig. 7-8, the stack is not part of the procedure, the procedure does not write anything into itself. Therefore, the principal requirement of reentrant code is satisfied and the procedure can be shared in time-multiplexed fashion. If this procedure were to be called by another process before the current

call is completed, the SS register would be saved by the system and a new segment address would be loaded into SS during the switching operation. As a result, if the newly entered process calls the reentrant procedure, then a different area for storing TEMPX, TEMPY, and TEMPZ is used by the new process and the temporary data for the suspended process is preserved.

After a multiplication has been carried out, a proper return is made to the calling process by restoring all of the register contents except BP, adding 12 to SP, restoring BP, and executing a RET 6 instruction. The 6 in the return instruction

Figure 7-9 Reentrant procedure for signed multiplication of two 32-bit operands.

```

FRAME      STRUC
    TEMPZ     DW      ?,?
    TEMPY     DW      ?,?
    TEMPX     DW      ?,?
    SAVE_BP   DW      ?
    SAVE_CS_IP DW      ?,?
    Z_ADDRESS DW      ?
    Y_ADDRESS DW      ?
    X_ADDRESS DW      ?
FRAME      ENDS

MULTIPLY   PROC    FAR
    PUSH    BP
    SUB    SP,12
    MOV    BP,SP
    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    PUSH    SI
    PUSH    DI
    MOV    SI,0
    MOV    BX,[BP].X_ADDRESS
    MOV    AX,[BX]
    MOV    [BP].TEMPX,AX
    MOV    AX,[BX+2]
    MOV    [BP].TEMPX+2,AX
    MOV    BX,[BP].Y_ADDRESS
    MOV    AX,[BX]
    MOV    [BP].TEMPY,AX
    MOV    AX,[BX+2]
    MOV    [BP].TEMPY+2,AX
    CMP    [BP].TEMPX+2,0
    JGE    CHECK_Y
    NOT    [BP].TEMPX
    NOT    [BP].TEMPX+2
    ADD    [BP].TEMPX,1
    ADC    [BP].TEMPX+2,0
    MOV    SI,1
    CMP    [BP].TEMPY+2,0
    JGE    START
    NOT    [BP].TEMPY
    NOT    [BP].TEMPY+2
    ADD    [BP].TEMPY,1
    ADC    [BP].TEMPY+2,0
    XOR    SI,1
    MOV    AX,[BP].TEMPX
    MUL    [BP].TEMPY
    MOV    [BP].TEMPZ,AX
    MOV    CX,DX
    MOV    AX,[BP].TEMPX+2
    MUL    [BP].TEMPY
    MOV    BX,DX
    ADD    CX,AX
    ADC    BX,0
    MOV    AX,[BP].TEMPX
    MUL    [BP].TEMPY+2
    ;SAVE BP
    ;RESERVE 6 WORDS OF WORKING SPACE
    ;BP POINTS TO FRAME
    ;USE SI AS SIGN INDICATOR
    ;MOVE OPERAND X TO TEMPX
    ;MOVE OPERAND Y TO TEMPY
    ;IS X POSITIVE
    ;YES, CHECK Y
    ;ELSE TAKE 2'S COMPLEMENT
    ;OF X
    ;INDICATE X IS NEGATIVE
    ;IS Y POSITIVE
    ;YES, START MULTIPLY
    ;ELSE TAKE 2'S COMPLEMENT
    ;OF Y
    ;SI HAS SIGN OF PRODUCT
    ;THIS CODE PERFORMS THE
    ;UNSIGNED DOUBLE PRECISION
    ;MULTIPLY AND STORES THE LOWER
    ;TWO WORDS OF THE PRODUCT AT
    ;TEMPZ AND THE UPPER TWO WORDS
    ;OF THE PRODUCT IN DX:AX

```

Figure 7-9 Continued.

```

ADD    CX,AX
ADC    BX,DX
MOV    [BP].TEMPZ+2,CX
MOV    CX,0
ADC    CX,0
MOV    AX,[BP].TEMPX+2
MUL    [BP].TEMPY+2
ADD    AX,BX
ADC    DX,CX
MOV    CX,[BP].TEMPZ+2
CMP    SI,0
JE     STORE_Z           ;CHECK SIGN
NOT   [BP].TEMPZ         ;AND NEGATE PRODUCT IF
NOT   CX                 ;SIGN IS NEGATIVE
NOT   AX
NOT   DX
ADD   [BP].TEMPZ,1
ADC   CX,0
ADC   AX,0
ADC   DX,0
STORE_Z: MOV   BX,[BP].ADDRESS_Z ;STORE PRODUCT
          [BX+6],DX ;IN LOCATION
          [BX+4],AX ;POINTED TO
          [BX+2],CX ;BY ADDRESS_Z
          AX,[BP].TEMPZ
          [BX],AX
          DI             ;RESTORE REGISTERS AND RETURN
          POP  SI
          POP  DX
          POP  CX
          POP  BX
          POP  AX
          ADD  SP,12
          POP  BP
          RET  6
MULTIPLY ENDP

```

is to compensate for the fact that the offsets of the two factors X and Y and the product Z are assumed to be pushed onto the stack by the calling process.

In addition to being pure, the code in a reentrant procedure may need to be *position independent*. This means that the code can run correctly no matter where it is placed in physical memory or where it is placed relative to the processes that call it. All local references must be made using some form of relative addressing, or the code must be able to adjust its local addresses according to its current location. Relative addressing is the much more desirable method for producing position-independent code and is accomplished on the 8086/8088 primarily by using the segment registers. The main reason why reentrant procedures should be position independent is that they are shared by more than one process and, therefore, are normally dynamically linked to various procedures at execution time. This linkage is minimized if the addresses in the reentrant code do not need to be modified.

7-4 MEMORY MANAGEMENT

A consistent set of program units that are coupled together but do not reference units outside the set, except through mass storage, is called a *job* (or *program*). In order to efficiently utilize the system resources and achieve the maximum degree of concurrent processing between the CPU and I/O, it is desirable to store as many

jobs as possible in memory. Therefore, how to manage the memory so that its wasted space is kept to a minimum is an important consideration in the design of a multiprogramming system.

The simplest method for storing more than one job in the memory is the *partition allocation scheme*. As depicted in Fig. 7-10, if this scheme is used, each job is allocated a contiguous area of memory. When a job needs to be executed, the loader requests the required amount of memory from the *memory management routine*, which is the part of the operating system that is responsible for storage allocation. If the memory management routine finds a contiguous free area whose size is larger than the job size, the beginning address of that free area is returned to the loader. After adjusting the necessary relative addresses so that they become physical addresses, the loader can load the job into the allocated area. Of course, the initial state of the job is "ready" and the job may not start executing immediately. If there is no single free area large enough, the job must wait before it can be loaded from mass storage.

The partition allocation scheme requires the memory management routine to maintain a memory map table. A possible implementation of this table is to assign an entry in the table to each partition. Each entry would contain the status, size, and beginning address of the partition, with the status being:

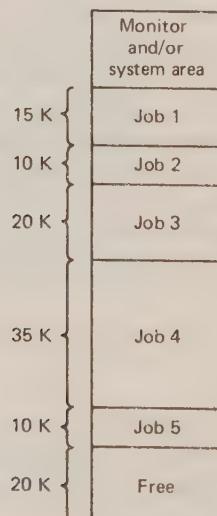
Allocated—If the partition is currently allocated to a job.

Free—If the partition is available.

Unused—If the entry is not associated with a partition.

The third possibility may result from two adjacent free partitions being combined into one.

Figure 7-10 Partition memory allocation.



Upon a request for storage allocation, the memory management routine checks the size of each free partition, starting at the top of the table. If a free partition is found whose size is equal to or larger than the required amount, its status is changed to allocated, its beginning address is returned, and, if the requested amount of memory is smaller than the amount in the partition, a new free partition is added to the table to account for the excess space. This algorithm for selecting a free partition is called the *first fit algorithm*. Another algorithm, the *best fit algorithm*, searches the entire memory map table to find the smallest free partition that meets the request.

When a job is terminated, the area it occupied is returned to the system. The memory management routine will update the memory map table to reflect the possible merge of adjacent free partitions that result from the released memory. Two possible situations that cause a merge are shown in Fig. 7-11. In the first case the released partition is adjacent to an existing free area. After the merge, the two free partitions are combined into a single larger partition and assigned a beginning address which is the lower of the two initial beginning addresses. In the second case, two existing free areas are separated by the released partition. As a result

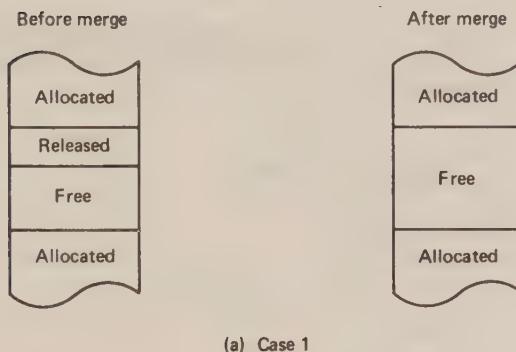
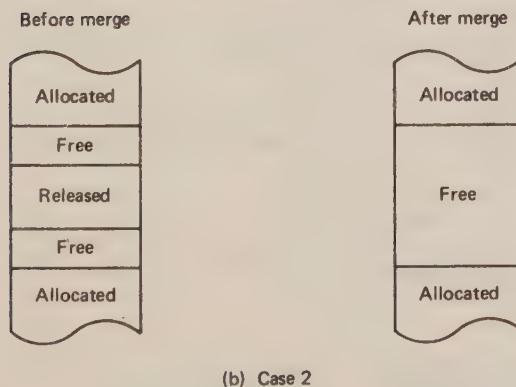


Figure 7-11 Merging adjacent free partitions.

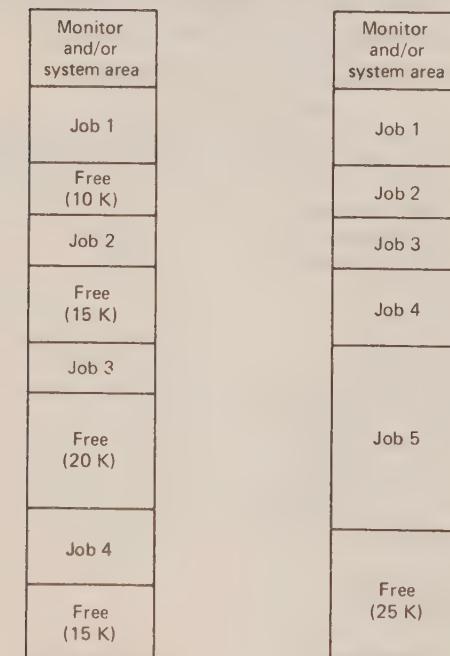


of the merge, three partitions are combined into one. Merging is necessary to keep the sizes of the free partitions as large as possible.

Simplicity and the fact that no special hardware is required are the major advantages of using the partition allocation scheme. However, as current jobs are completed and new jobs are loaded, more and more small free areas begin to appear. This causes the *fragmentation problem* illustrated in Fig. 7-12(a). As the small free areas become scattered in memory, the system may not be able to load a job even though the total amount of available space is large enough; thus, a significant portion of memory may be wasted.

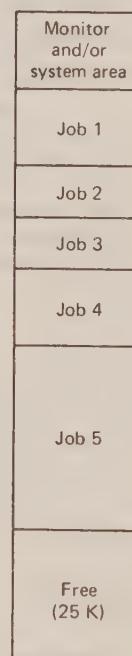
One way to eliminate memory fragmentation is to combine all free spaces into a single space by compressing the jobs in the memory together, as shown in Fig. 7-12(b). This scheme is known as the *relocation partition allocation scheme*. However, to implement this scheme a machine must be able to easily adjust each job so that, after the job is relocated from one area of memory to another, it can properly resume its execution.

For the 8086 and 8088, all physical addresses are formed by adding effective addresses to segment addresses multiplied by 16_{10} , with the segment registers indicating the segment addresses. This facility allows a program to be easily translated into position-independent code so that the program can be moved from one area



Job 5 needs 35 K and
cannot be loaded

(a) Fragmentation



(b) After compression is completed
and Job 5 is loaded

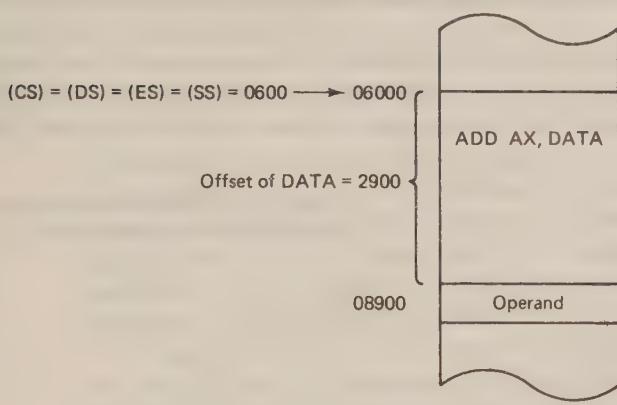
Figure 7-12 Memory fragmentation problem and the compression process.

to another without modifying its machine code. Each time a relocation is performed the segment addresses (i.e., the contents of the segment registers) must be updated by the monitor.

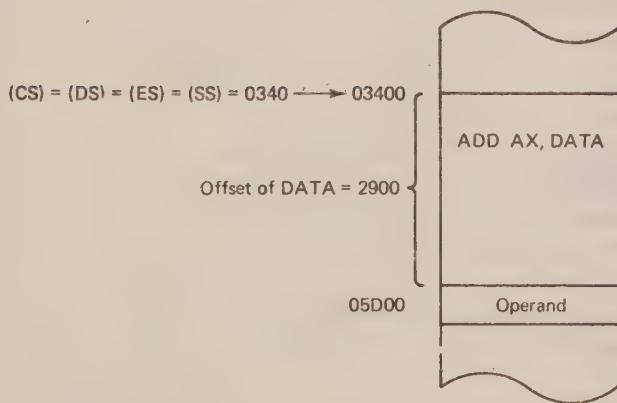
Figure 7-13 shows how relocation can be accomplished by using the segment registers. In this example, the job consists of a single segment and, therefore, all addresses are relative to the beginning of this segment. As the program is relocated to address 03400 from 06000, only the segment registers CS, DS, ES, and SS need to be changed to 0340 by the monitor. Note that before relocation, the ADD instruction accesses DATA at $06000 + 2900 = 08900$, and after relocation it will correctly access DATA at $03400 + 2900 = 05D00$.

In order to ensure position independence, the memory management routine must have complete and sole control over the segment registers, and these registers must not be modified from within the program (as they were in the examples in the earlier chapters). Also, all procedure calls and jumps must be near. Otherwise, if the CS is saved by a call which is made before the relocation and the return is

Figure 7-13 Relocation as performed by segment registers.



(a) Before relocation



(b) After relocation

made after the relocation, the CS would point to the old return segment address and the program would return to the wrong place. Because segment registers may not be modified by a relocatable program, a job may have at most four segments—the code, data, extra data, and stack segments—and the maximum job size is limited to $4 \times 64K = 256K$.

Because compression requires the operating system to move jobs from one area to another, it is a time-consuming process and compression should not be performed unless it is necessary. Under a relocation partition allocation scheme, the memory management routine normally allocates the needed space and maintains the memory map table as if a partition allocation scheme were used. However, if a single free area larger than the requested size cannot be found the operating system must then determine whether the total amount of free memory is large enough. If it is, a compression process is initiated and the base addresses for each job (which are indicated by the saved contents of the segment registers) are adjusted and the memory map table is updated accordingly.

Another approach to reducing the fragmentation problem is to divide a program into several pieces. As shown in Fig. 7-14, the pieces of each job are stored in separate areas which are not necessarily physically contiguous, but are logically adjacent to each other. When using an 8086/8088, a job can be divided into several segments, each with a length that varies from 16 bytes to 64K bytes. Just where

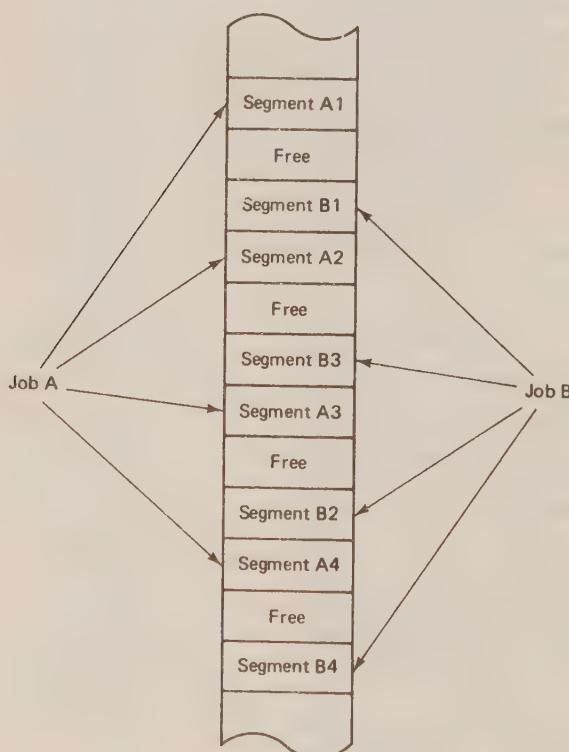


Figure 7-14 Solving fragmentation by using several segments for each job.

each segment is to be loaded depends on the run-time situation. To simplify the address adjustments that are required during the process of loading a job, it is best to load the segment registers from a table which contains the locations of all the segments in a job. This implies that all intersegment control transfers should be made using indirect branch or call instructions that obtain their branch addresses from this table.

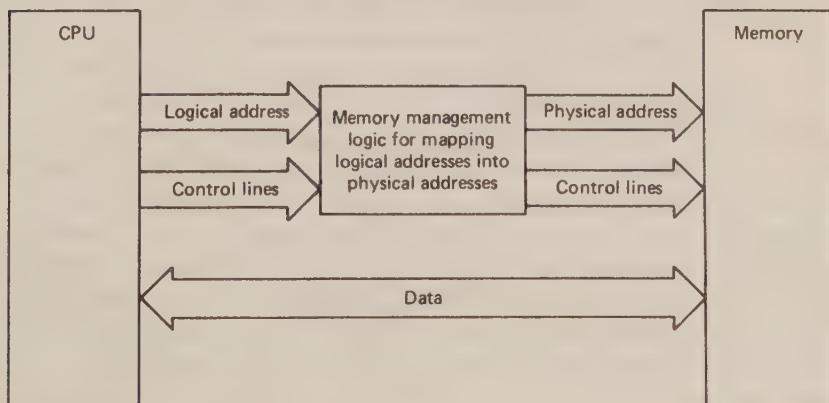
One possible implementation is to construct a single pointer table which would include a set of pointers for each job. The pointer table would contain the locations of the code, data, and stack segments for the job corresponding to it. The ES register would be reserved for accessing the pointer table. An intersegment control transfer would then be accomplished by an indirect branch or call through the pointer table using the ES register to obtain the segment address. Similarly, for accessing a new data segment, the LDS instruction would be used to load the DS register from the ES-based table. This would allow the operating system to load each segment into a separate area and only the pointer table would need to be updated by the memory management routine.

7-5 VIRTUAL MEMORY AND THE 80286

A more sophisticated memory management scheme can be achieved by the hardware dynamic address translation design illustrated in Fig. 7-15. For each memory reference, the *logical address* output by the CPU is translated into the *physical address*, which is the address sent to the memory by the memory management hardware. Logical addresses are the addresses that are generated by the instructions according to the addressing modes.

Because the logical addresses may be different from the physical addresses, the user can design a program in a logical space, also called a *virtual space*, without consideration for the limitations imposed by the real memory. This provides two major advantages. First, by dividing a program into several pieces with each mapped

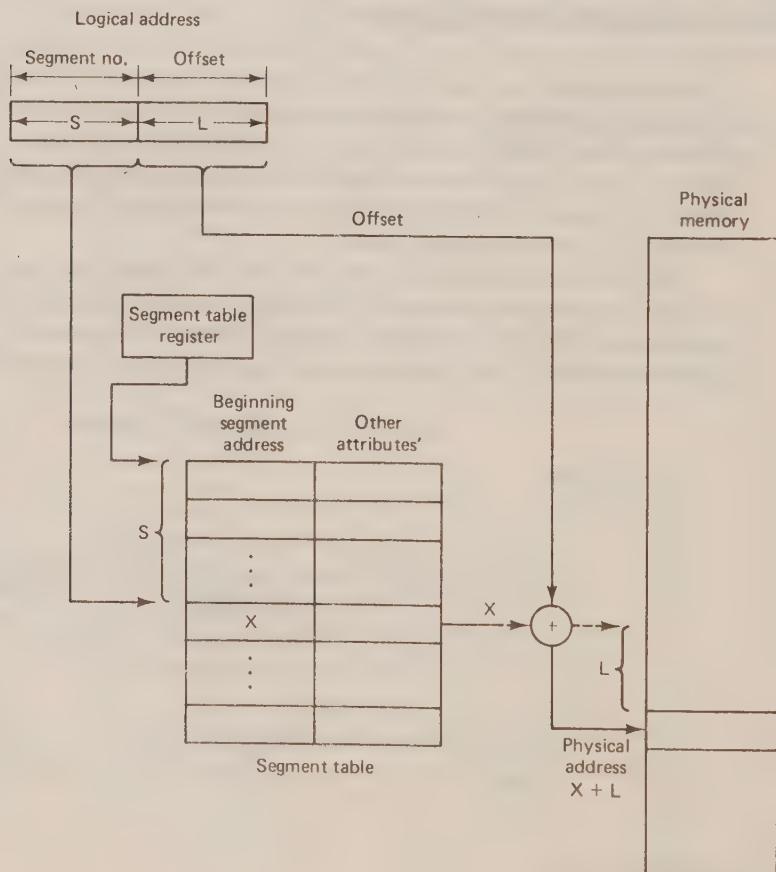
Figure 7-15 Dynamic address translation.



into an area in real memory, a program need not occupy a contiguous memory section. Therefore, memory fragmentation is reduced and less memory space is wasted. Second, it is not necessary to store the entire program in memory while it is executing. Whenever a portion of the program that is not currently in memory is referenced, the operating system can suspend the program, load in the required section of code, and then resume the program's execution. This allows a program's size to be larger than the actual memory available. In other words, a user "virtually" has more memory to work with than actually exists.

With address translation hardware, the program is divided into segments according to the logical structure of the program and the resulting memory management scheme is called *segmentation*. When using segmentation, each logical address consists of two fields, the segment number and the offset within the segment. The number of bits representing the segment number governs the maximum number of segments allowed in a program, and the number of bits allocated to the offset specifies the maximum segment size. For example, if the segment number

Figure 7-16 Segmentation scheme.



and offset have m and n bits, respectively, a program may have up to 2^m segments, with each segment having a maximum size of 2^n bytes, thus providing each user with a virtual storage of 2^{m+n} bytes.

Figure 7-16 illustrates the mapping of logical addresses into physical addresses. The segment number S in a logical address is used as an index into the segment table, which returns the beginning physical address X of the referenced segment. This address is added to the offset L to form the physical address of the memory location. Because each job may be assigned a separate area in the segment table, the base address of that section of the segment table that is associated with the currently executing job is stored in a register called the *segment table register*. The index S is made relative to the segment table register. When the system switches from one job to another, the segment table register is updated to point to a new section of the segment table. Because the address translation must be performed for every memory reference, either the entire segment table or that portion of the table containing the beginning addresses of the segments that are currently in use must be stored in registers which are part of the memory management hardware.

Each entry in the segment table is referred to as a segment descriptor. Depending on the particular implementation, a segment descriptor may include attributes in addition to the beginning segment address. The most common of these additional attributes are the:

Status Field—Indicates whether or not the referenced segment is in the memory. A single bit is sufficient to distinguish the two possible situations. If the referenced segment is not present in the memory, a segment fault occurs which causes a trap to the monitor. The monitor then loads the required segment into memory, updates the segment table, and continues with the program. When sufficient space is not currently available, one or more existing segments are swapped out to mass storage.

Segment Length Field—Indicates the size of the segment. This information prevents other segments from being accessed due to invalid addresses. When the offset is greater than the segment length, a trap occurs.

Protection Field—Provides protection against unauthorized reading, writing, or execution. Normally, this field contains 3 bits, which indicate:

Readable: The contents may be fetched as operands.

Writeable: The contents may be modified.

Executable: The contents may be fetched as instructions.

Each segment, as a logical unit in a program, can be given combinations of the access restrictions by setting combinations of these bits.

Reference Field—Provides useful information in determining which segment is to be replaced. When a new segment is called for, it may be necessary to remove a segment currently in memory to make more space available. Replacement algorithms such as the FIFO, least frequently used (LFU), and least recently used (LRU) schemes have been proposed and studied. However, except for the FIFO scheme, their implementations require complex hardware. An alternative is to have a reference bit associated with each

segment. When a segment is used, its reference bit is automatically set to 1. After all of the reference bits are set to 1, the monitor can reset them to 0. By examining the reference bits, the monitor can pick one which has not been referenced recently.

Change Field--Indicates whether or not the segment has been modified since being brought into memory. When the segment is replaced, if it has been changed, it is necessary to store it back into mass storage before loading the incoming segment. If the segment has not been modified, the monitor need only load the incoming segment from mass storage, thus saving a write operation.

During a memory reference, a typical sequence of operations performed by the memory management hardware is:

1. Determine the segment number and access the associated entry in the segment table.
2. Check the status. If the needed segment is not in memory, generate a trap to the monitor which initiates a segment fault service routine.
3. Compare the offset and the segment length. If the offset is too large, generate a trap to the monitor.
4. Check the access type. If an unauthorized reference is being attempted, generate a trap to the monitor.
5. Add the beginning segment address to the offset to form the physical address and put the result on the address bus.
6. Update the reference and change fields and wait for the next memory reference.

A segment fault service routine would typically include the following steps:

1. Check the memory map table. If the required segment fits into a free memory area, go to step 4; otherwise, proceed to step 2.
2. Select the segment to be removed by examining the reference bits and update the status of the selected segment.
3. Check the change bit of the segment to be replaced and, if this segment has been modified, write it back into mass storage.
4. Load the required segment into memory and update the memory map and segment tables.
5. Return to the instruction that caused the segment fault.

Traps that occur during address translation and are not caused by a segment not being in memory would normally cause the current job to be terminated.

Intel has combined an improved version of its 8086 CPU with segmentation memory management logic similar to that described above and put the resulting logic onto a single chip known as the 80286. In addition to including memory

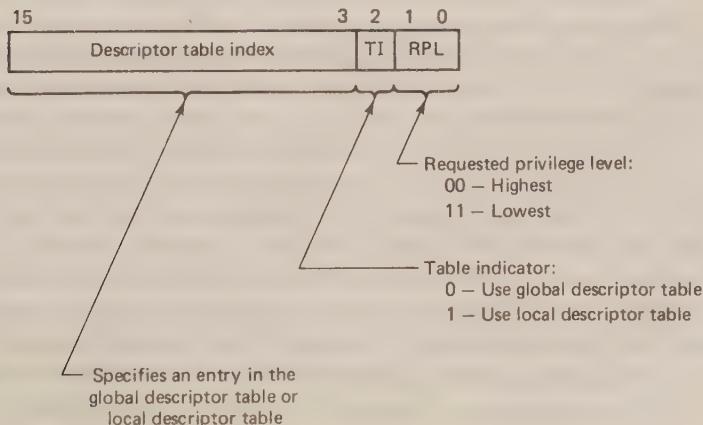
management facilities, the 80286 executes a superset of the 8086 instruction set and is six times faster than the 8086. A program written for an 8086 or 8088 can, after being reassembled with no or minimal modifications, be executed by the 80286.

The 80286 has the same four segment registers, CS, DS, ES, and SS, and the same general register set as the 8086, but also includes several other registers that are used for multiprogramming purposes. How the segment registers are used to generate physical addresses depends on the operating mode of the processor. There are two such modes, the real address mode and the protected virtual address mode, or simply the protected mode. Under the real address mode, the function of the segment registers is the same as for the 8086, in that the contents of a segment register multiplied by 16 is the beginning address of the segment.

For the protected mode, the contents of a segment register are used as a segment selector which points to where the segment descriptor is stored in a descriptor table. The hardware supports one interrupt descriptor table, one global descriptor table, and an unlimited number of local descriptor tables. Each task has its own local descriptor table so that it will be protected from other tasks, and the global descriptor table is for shared segments. These tables can be located anywhere in memory and are pointed to by the interrupt descriptor table register, global descriptor table register, and local descriptor table register, which are in the 80286 and must be loaded by the operating system using special instructions. In addition, a 48-bit extension register is associated with each of the segment registers. These extensions are for holding the descriptors for the current code, data, extra data, and stack segments and are dynamically updated.

Figure 7-17 shows the bit definitions of the segment registers and Fig. 7-18 shows the format of their extensions. Bit 2 of a segment register specifies whether the segment descriptor is in the global descriptor table or a local descriptor table. Bits 1-0 specify the segment's privilege level, which can be used to protect the segment from being accessed by tasks with lower privilege levels. The remaining

Figure 7-17 Segment register format of an 80286 in protected mode.



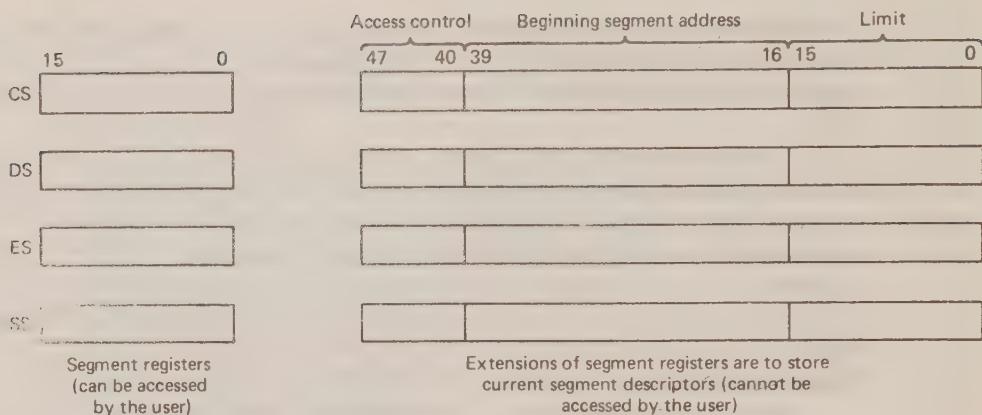


Figure 7-18 Segment registers and their extensions.

bits indicate the entry in the descriptor table that is to be used to perform the address translation. Each extension consists of an 8-bit access control field, a 24-bit beginning-of-segment address field, and a 16-bit limit field.

When a memory reference is made, if the segment register that is used is not changed, then a 24-bit physical address is formed by adding the 24-bit segment address in the register's extension to the effective address indicated by the instruction (provided that the limit and access rights are not violated). If the segment register is changed either explicitly (as with a data transfer instruction) or implicitly (as with a far branch instruction), then the 80286 will examine the bits in the segment register and automatically load a new extension from the appropriate descriptor table using the offset contained in the segment register. The 80286 will then determine if the needed segment is in memory. If the segment is in memory, the memory reference is completed by adding the segment address (in the new extension) to the effective address. If the segment is not in memory, the 80286 will execute a trap and the operating system will bring the segment in from mass storage. Once in memory, the interrupted instruction can be restarted.

Note that when in protected mode the 80286 outputs a 24-bit address and, therefore, the 80286, which must have 24 address pins, can accommodate up to 16 megabytes of memory. Also note that a logical address consists of a 14-bit segment number (a 13-bit table index plus a 1-bit table indicator) and a 16-bit effective address. This allows each job to utilize up to 1 gigabyte (2^{30}) of virtual memory.

As shown in Fig. 7-18, a segment register extension, and any corresponding descriptors, also contain a control access field and a limit field. The limit field stores the offset of the last byte in the segment. For code and data segments, this field should be the largest offset in the segment, i.e., the size of the segment. However, a stack segment grows from its highest address toward its lowest address. Therefore, for a stack segment, the limit field should store the lowest offset in the segment and access should be denied if the effective address is less than this limit.

The access control field is defined in Fig. 7-19. The P bit indicates whether or not the segment is in memory and the DPL bits specify the privilege level of

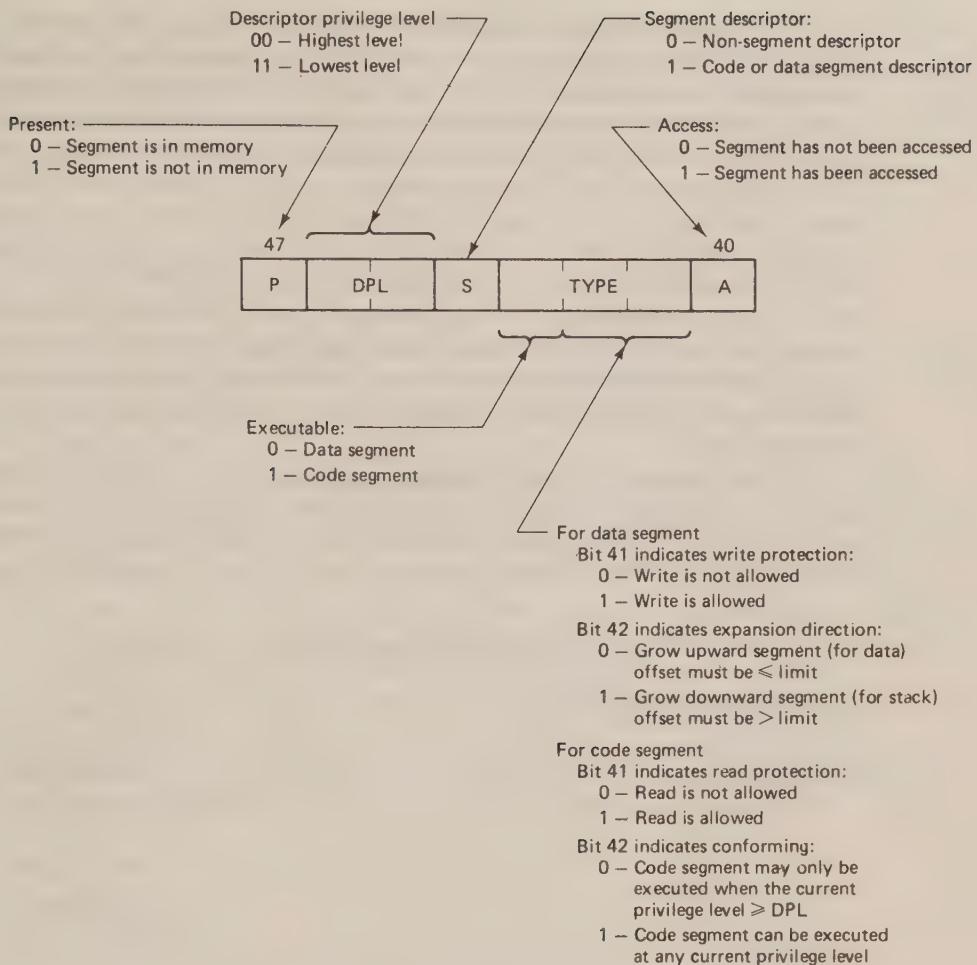


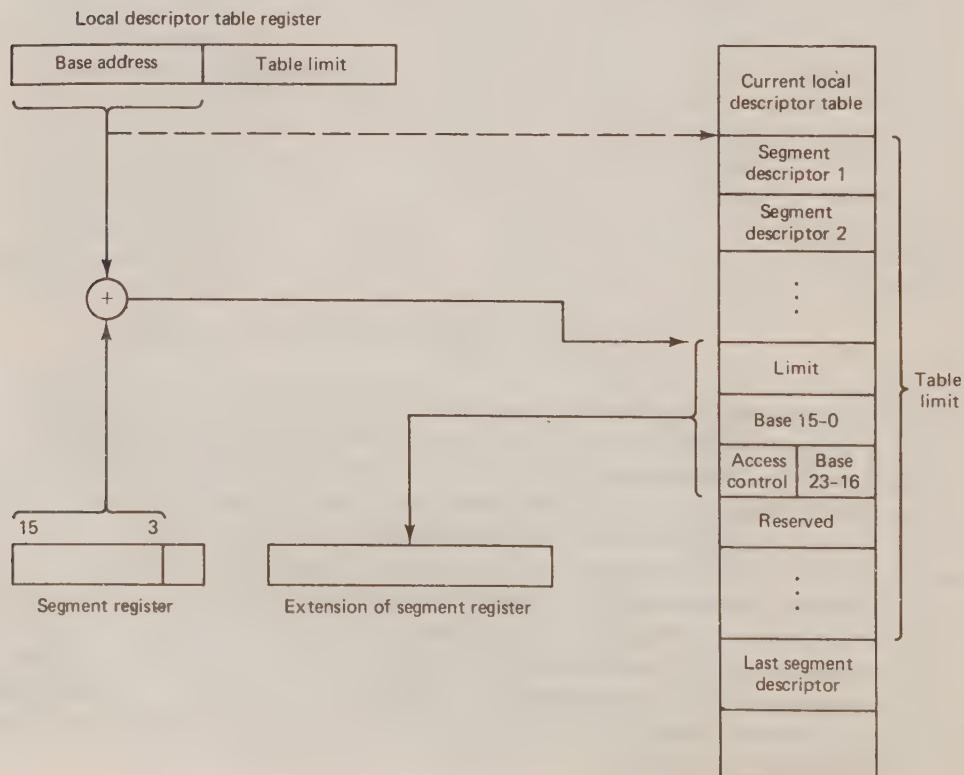
Figure 7-19 Definition of the access control field.

the segment. In conjunction with the TYPE field, these bits may protect a code segment against being executed by a task whose current task privilege level is lower than the descriptor privilege level. The S-bit indicates whether the descriptor is a segment descriptor or nonsegment descriptor. In addition to segment descriptors for the code, data, and stack segments, the 80286 has system control descriptors for special system data and control transfer operations. The most significant TYPE bit (bit 43) distinguishes a data segment from a code segment and can be used to determine whether or not a segment can be executed. For a data segment, bit 42 indicates the expansion direction and bit 41 provides protection against write operations. For a code segment, the same 2 bits are used for protection against being executed if the current task privilege level is lower than the DPL and against read operations, respectively.

It is important to note that the extensions of the segment registers are not programmable. As mentioned above, when the contents of a segment register are changed by an instruction, the selected segment descriptor will automatically be loaded into the extension from the appropriate descriptor table in memory. Figure 7-20 illustrates this process when the new segment descriptor is in the local descriptor table for the current task. The base address in the local descriptor table register points to the base of the current local descriptor table and the upper 13 bits in the segment register are used as an index. The first three words of the selected descriptor are loaded into the extension of the segment register. (There is a fourth word in each descriptor, but it is not presently used.) Switching from one task to another will change the contents of the local descriptor table register so that it points to a different local descriptor table. The 80286 has 16 instructions for loading and storing the descriptor table registers and for modifying and verifying various descriptor fields.

To aid process (or task) switching, each task has a task state segment which serves as a save area for its entire execution state. The current task state segment is specified by a system segment descriptor that is pointed to by the contents of a register called the task register. The task state segment is primarily for saving the

Figure 7-20 Loading the extension of a segment register.



registers. During a task switch, the 80286 saves the entire execution state in the current task state segment, loads a new execution state from the selected task state segment, and resumes the new task. For the details on task switching and other multiprogramming features of the 80286, the reader should refer to the appropriate Intel manuals (see the Bibliography).

BIBLIOGRAPHY

1. Alexy, George, *Multiprogramming with the iAPX 88 and iAPX 86 Microsystems*, Application Note Ap-106 (Santa Clara, Calif.: Intel Corporation, 1980).
2. *1982 Data Component Catalog* (Santa Clara, Calif.: Intel Corporation, 1982).
3. *iRMX 86 Nucleus Reference Manual* (Santa Clara, Calif.: Intel Corporation, 1981).
4. *Introduction to the iAPX 286* (Santa Clara, Calif.: Intel Corporation, 1982).
5. Wharton, John, *Using Operating System Firmware Components to Simplify Hardware and Software Design*, Application Note Ap-130 (Santa Clara, Calif.: Intel Corporation, 1982).
6. Dijkstra, E. W., "Cooperating sequential processes," Technological University, Eindhoven, The Netherlands, 1965. (Reprinted in *Programming Languages*, F. Genuys, ed., Academic Press, Inc., New York, 1968.)
7. Shaw, Alan C., *The Logical Design of Operating Systems* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1974).
8. Madnick, Stuart E., and John J. Donovan, *Operating Systems* (New York: McGraw-Hill Book Company, 1974).

EXERCISES

1. Assume that the four jobs A, B, C, and D, none of which involves I/O activity, are to be executed and their execution times are 60, 20, 40, and 10 minutes, respectively. Given that all of the jobs are submitted to the system at the same time and in the order indicated in the preceding sentence:
 - (a) Determine the average turnaround time if these jobs are executed serially in the order specified.
 - (b) Determine the average turnaround time under a nonprioritized time-sharing system. Turnaround time for a job is defined as the time between the job being submitted and the job being completed.
2. Consider a multiprogramming system that handles ticket reservations through two terminals. Assume that each terminal initiates a process to perform the following operations:
 1. Input the number of requested tickets from a terminal.
 2. Compare the number of available tickets stored in a common variable TICKET_AVA with the requested number. If the requested number is larger, go to step 4. Otherwise, proceed with step 3.

3. Subtract the requested number from TICKET_AVA, print tickets, and return to step 1.
 4. Print a message "NOT ENOUGH TICKETS" and return to step 1.
- Suppose that TICKET_AVA has five tickets left, and the inputs to the terminals are 4 and 3.
- (a) If the critical section problem does not occur, determine the possible final values in TICKET_AVA and the numbers of tickets issued.
 - (b) Determine all possible results if the critical section problem occurs.
3. Some machines do not have special test and set instructions. Assume that the 8086 does not have an XCHG instruction and show a possible implementation of the semaphore operators. (*Hint:* Use a rotate through carry instruction.)
 4. Suppose that processes A and B are structured as follows:

Process A:

- Step 1. Decrement COUNT by 1. (COUNT is shared and can also be modified by other processes; therefore, it must be protected from simultaneous usage.)
- Step 2. Wake up process B.
- Step 3. Wait to be awakened by process B.
- Step 4. Execute a noncritical section of code.
- Step 5. Go to step 1.

Process B:

- Step 1. Wait to be awakened by process A.
- Step 2. Execute a critical section of code.
- Step 3. Wake up process A.
- Step 4. Go to step 1.

- (a) Define the required semaphores and, if applicable, determine their initial values.
- (b) Insert the semaphore operators in processes A and B that are needed to provide process synchronization.

See the example at the end of Sec. 7-2.

5. What is pure code? List some applications of pure code.
6. What is memory fragmentation? How can it be reduced?
7. Assume that ARRAY1, ARRAY2, and PROC1 are in segments SEGA, SEGB, and SEGC, respectively. For the following instruction sequence, indicate which instructions are position dependent if SEGA, SEGB, and SEGC are stored as a unit together with the code segment containing the instruction sequence:

| | |
|------|------------------|
| MOV | AX,SEGA |
| MOV | DS,AX |
| MOV | BX,OFFSET ARRAY1 |
| ADD | DX,[BX][SI] |
| MOV | AX,SEGB |
| MOV | DS,AX |
| MOV | BX,OFFSET ARRAY2 |
| MOV | [BX][SI],DX |
| CALL | FAR PTR PROC1 |

8. Rewrite the instruction sequence in Exercise 7 as position-independent code by making the following assumptions:

1. A pointer table has been set up as follows:

```
SEG PTR      SEGMENT
    ARRAY1PTR   DD  ARRAY1
    ARRAY2PTR   DD  ARRAY2
    PROC1PTR    DD  PROC1
.
.
.
SEG PTR      ENDS
```

2. After the program has been brought in, the operating system will fill in the actual segment addresses in the pointer table.
3. The segment register ES is reserved to access the pointer table and will always contain the segment address of SEG PTR.
9. Suppose that the ready queue is as shown in Fig. 7-3 and that the process at the top of the queue is to be switched to the running state. Write a program sequence that will update the first-element pointer and process table and put the beginning address of the appropriate process control block in BX.
10. Repeat Exercise 9, but this time assume the prioritized queue in Fig. 7-4.
11. Suppose that process 11 is to be added to the bottom of the queue shown in Fig. 7-3 and that BX contains the address of its process control block. Write a program sequence that will update the last-element pointer and the process table.
12. Repeat Exercise 11 for the prioritized queue in Fig. 7-4. Assume that process 11 has a priority level of 2.
13. Assume that the maximum segment size is 1000 bytes and a job may have only three segments stored in memory. For the following reference sequence:

500,1200,1400,2300,900,5300,3500,2800,3800,
4300,2500,3600,3700,4200,5400,5500,3650,2700

determine the number of segment faults that would occur if the system were based on a:

- (a) FIFO replacement algorithm.
- (b) LFU replacement algorithm.
- (c) LRU replacement algorithm.
14. Is it necessary to include additional hardware in the memory management logic for implementing a FIFO replacement scheme? Why is it necessary to include additional hardware in the memory management logic in order to implement a LFU or LRU replacement scheme?
15. In order for the memory management hardware to enforce read, write, and execute protection, what information is needed from the CPU along with the logical address?

8

System Bus Structure

A set of conductors used for communicating information between the components in a computer system is called a *bus*. If a bus connects two minor components within a major component (e.g., the control unit to the set of working registers within the CPU), it is called an *internal bus*. When a bus connects two major components, such as a CPU and an interface, it is called an *external bus*. Because an internal bus is ordinarily internal to an IC device and its construction is dependent on the device, the exact construction of these buses is of little interest to us. External buses, on the other hand, have common characteristics that must be understood when designing the overall architecture of a computer system. Some systems include more than one external bus and they will be considered in Chap. 11. Others contain only one bus, which is referred to as the *system bus*, and it is the basic structure of system buses, particularly those in 8086- and 8088-based systems, that is discussed in this chapter.

Figure 8-1 illustrates the fundamental structure of a system bus and its relationships to the various components of the computer system. In this chapter we will be concerned primarily with the interface between the CPU and system bus, the bus control logic. Precisely how the other interfaces connected to the bus respond to and interpret the signals on the bus is studied in Chaps. 9 and 10. The complexity of the bus control logic depends on the amount of translation needed between the system bus and the pins on the CPU, the timing requirements, whether or not interrupt management is included, and the size of the overall system. If a system component other than the CPU can be master of the bus, then all of the address and data lines and most of the control lines must be capable of being logically disconnected from the CPU or bus control logic, i.e., most pins connected to the bus must be such that they can enter a high-impedance state.

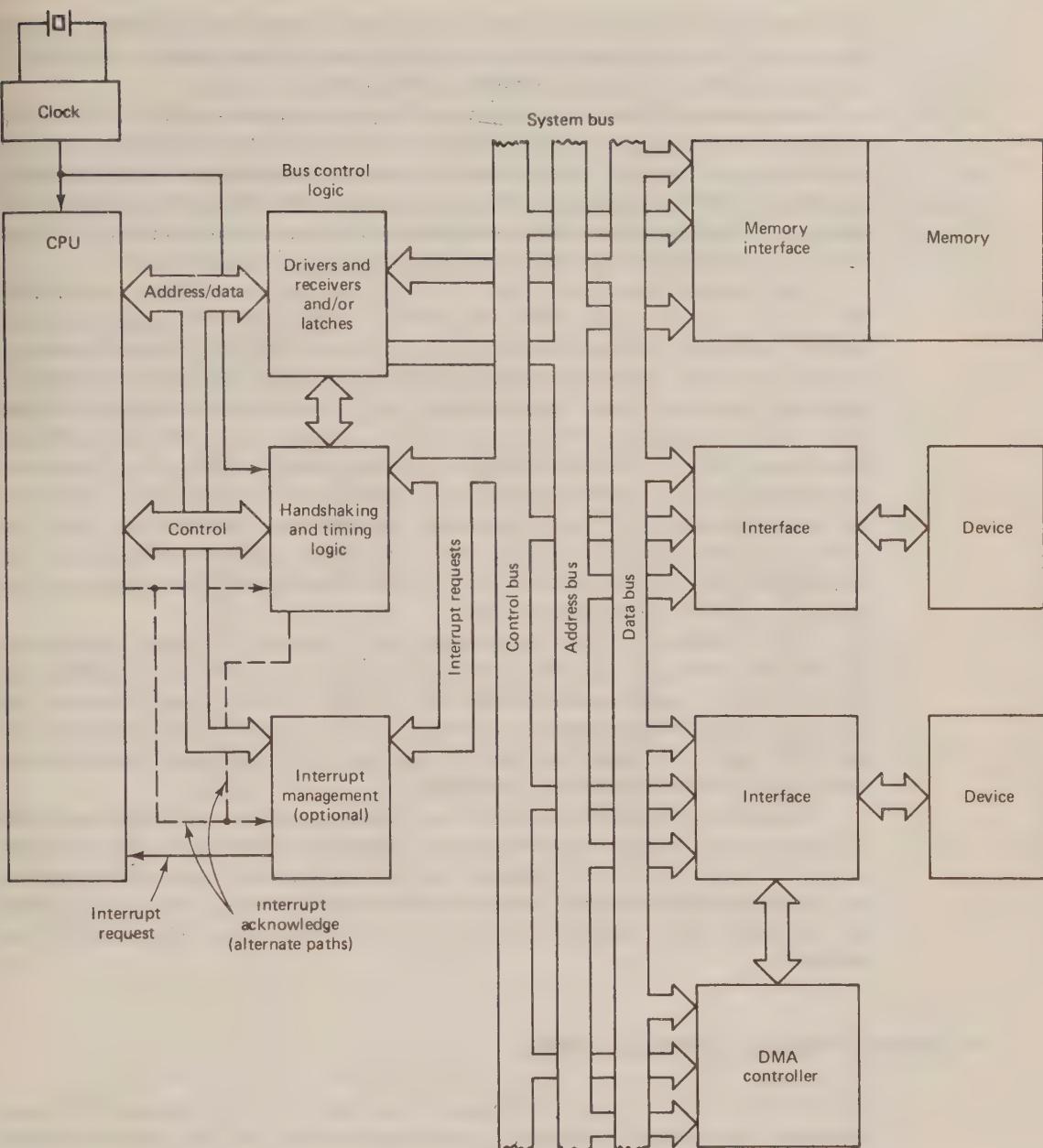


Figure 8-1 Typical system bus architecture.

Generally, the circuits that drive the pins on a single-chip CPU have a quite limited driving capability and can be connected to only a few interfaces. For a small system, many or all of the CPU control pins could be used directly and the handshaking logic could be reduced or, perhaps, eliminated. Similarly, drivers and

receivers would not be needed for the data and address lines. However, systems with several interfaces would need bus driver and receiver circuits connected to the bus in order to maintain adequate signal quality.

In addition to drivers and receivers, timing considerations may be such that latches are needed to hold the address that is output by the CPU during one part of the bus cycle but must be maintained throughout the cycle. Some processors, including the 8086 and 8088, use the same pins for both data and addresses. This means that during a data transfer operation the address, which is output first, must be latched before the bus is used to transfer the data.

The timing of the signals within the CPU and bus control logic is controlled by a clock. The bus cycles and CPU activity are controlled by groups of clock pulses. The exact number of clock pulses, or cycles, within a bus cycle varies. A typical CPU input transaction would proceed by outputting the address of the data during the first clock cycle, signaling that a read is to take place during the second clock cycle, waiting an indeterminate number of clock cycles for the addressed device to put the data on the data lines, inputting the data, and signaling the device that the transfer is complete during the last clock cycle. For an output transaction the address would again be output during the first cycle and the data would be output during the second cycle along with a write signal. After the addressed device has accepted the data it would return a transfer complete acknowledgment which causes the CPU to enter the last cycle.

As in previous chapters the discussion in this chapter will center on the 8086, and, for the most part, the 8088 is considered only by pointing out its differences from the 8086. Keep in mind that the 8088 is fully software compatible with the 8086 and, as an 8-bit processor, is hardware compatible with the peripherals in the 8080/8085 family. By using the 8088, an existing 8080/8085-based system can be updated with minimal hardware modification.

Section 8-1 considers the modes and pin assignments of the 8086 and 8088 microprocessors and the single-bus configurations of systems based on these processors. The second section describes the timing of bus transfers on 8086/8088 systems and the third section examines the Intel 8259A priority interrupt controller. Section 8-4 discusses bus standards by focusing on the Intel MULTIBUS bus standard.

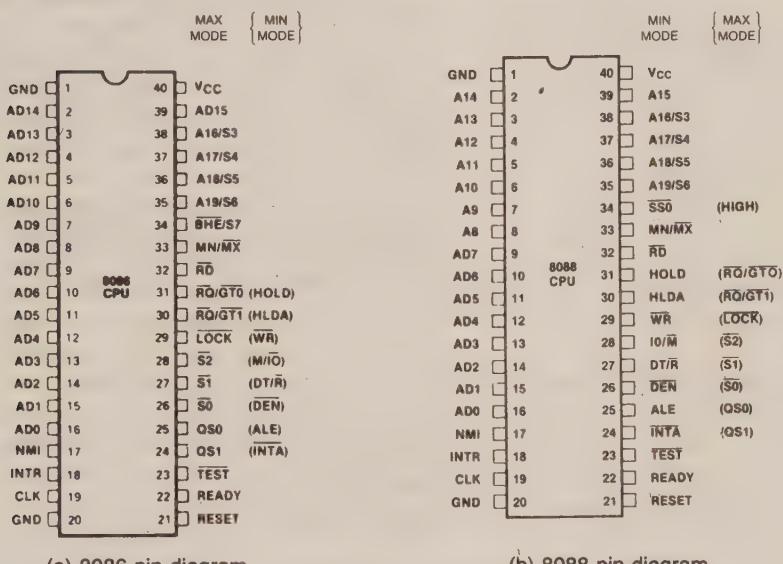
8-1 BASIC 8086/8088 CONFIGURATIONS

In order to adapt to as many situations as possible both the 8086 and 8088 have been given two modes of operation, the minimum mode and the maximum mode. The minimum mode is used for a small system with a single processor, a system in which the 8086/8088 generates all the necessary bus control signals directly (thereby minimizing the required bus control logic). The maximum mode is for medium-size to large systems, which often include two or more processors. In the maximum mode, the 8086/8088 encodes the basic bus control signals into 3 status

bits, and uses the remaining control pins to provide the additional information that is needed to support a multiprocessor configuration.

Pin diagrams for the 8086 and 8088 and the pin definitions that are common to both modes are given in Fig. 8-2. Pin 33 (MN/MX) determines the configuration option. When it is strapped to ground the processor is to be used in a maximum mode configuration and when it is strapped to +5 V it is to be operated in its minimum mode. Both processors multiplex the address and data signals and both have 20 address pins with address and status signals being multiplexed on the 4 most significant address pins. However, because the 8088 can only transfer 8 bits

Figure 8-2 Pin assignment summary. [Parts (a) and (b) reprinted by permission of Intel Corporation. Copyright 1981.]



(a) 8086 pin diagram

(b) 8088 pin diagram

| Pin(s) | Symbol | In/Out 3-State | Description |
|-------------------|----------|-------------------|---|
| 1 | GND | | Ground |
| 2-16 ¹ | AD14-ADO | I/O-3 | Outputs address during the first part of the bus cycle and inputs or outputs data during the remaining part of the bus cycle. |
| 17 | NMI | I | Nonmaskable interrupt request - positive-going edge triggered. |
| 18 | INTR | I | Maskable interrupt request - level-triggered |
| 19 ² | CLK | I | Clock - 33% duty cycle, maximum rate depends on CPU model 5 MHz for 8086 8 MHz for 8086-2 10 MHz for 8086-1 |

| | | | | | | | | | | | | | | | | | | |
|-----------------|-------------------|------------|---|----|----|----------|---|---|----|---|---|----|---|---|------------|---|---|----|
| 20 | GND | | Ground | | | | | | | | | | | | | | | |
| 21 | RESET | I | Terminates activity, clears PSW, IP, DS, SS, ES, and the instruction queue, and sets CS to FFFF. Processing begins at FFFF0 when signal is dropped. Signal must be 1 for at least 4 clock cycles. | | | | | | | | | | | | | | | |
| 22 | READY | I | Acknowledgment from memory or I/O interface that CPU can complete the current bus cycle. | | | | | | | | | | | | | | | |
| 23 | TEST | I | Used in conjunction with the WAIT instruction in multiprocessing environments. A WAIT instruction will cause the CPU to idle, except for processing interrupts, until a 0 is applied to this pin - see Chap. 11. | | | | | | | | | | | | | | | |
| 24-31 | - | - | Definition depends on mode - see Figs. 8-3 and 8-8. | | | | | | | | | | | | | | | |
| 32 | <u>RD</u> | 0-3 | Indicates a memory or I/O read is to be performed. | | | | | | | | | | | | | | | |
| 33 | MN/ <u>MX</u> | I | CPU is in minimum mode when strapped to +5 V and in maximum mode when grounded. | | | | | | | | | | | | | | | |
| 34 ³ | <u>BHE/S7</u> | 0-3 | If 0 during first part of bus cycle this pin indicates that at least one byte of the current transfer is to be made on pins AD15-AD8; if 1 the transfer is made on AD7-AD0. Status S7 is output during the latter part of bus cycle, but, presently, S7 has not been assigned a meaning. | | | | | | | | | | | | | | | |
| 35-38 | A19/S6- A16/S3 | 0-3 | During the first part of the bus cycle the upper 4 bits of the address are output and during the remainder of the bus cycle status is output. S3 and S4 indicate the segment register being used as follows: | | | | | | | | | | | | | | | |
| | | | <table border="0"> <tbody> <tr> <td>S4</td> <td>S3</td> <td>Register</td> </tr> <tr> <td>0</td> <td>0</td> <td>ES</td> </tr> <tr> <td>0</td> <td>1</td> <td>SS</td> </tr> <tr> <td>1</td> <td>0</td> <td>CS or none</td> </tr> <tr> <td>1</td> <td>1</td> <td>DS</td> </tr> </tbody> </table> | S4 | S3 | Register | 0 | 0 | ES | 0 | 1 | SS | 1 | 0 | CS or none | 1 | 1 | DS |
| S4 | S3 | Register | | | | | | | | | | | | | | | | |
| 0 | 0 | ES | | | | | | | | | | | | | | | | |
| 0 | 1 | SS | | | | | | | | | | | | | | | | |
| 1 | 0 | CS or none | | | | | | | | | | | | | | | | |
| 1 | 1 | DS | | | | | | | | | | | | | | | | |
| | | | S5 gives the current setting of IF. S6 is always 0. | | | | | | | | | | | | | | | |
| 39 ¹ | AD15 | I/0-3 | Same as AD14-AD0 | | | | | | | | | | | | | | | |
| 40 | VCC | - | Supply voltage - +5 V ± 10% | | | | | | | | | | | | | | | |

¹On 8088, AD15-AD8 are A15-A8 and are only for outputting address bits.

²5 MHz for the 8088, and 8 MHz for the 8088-2.

³On 8088, this pin is denoted S80 and is used in minimum mode to denote status. Logically equivalent to S0 - see Fig. 8-8. It is always 1 in maximum mode.

(c) Pin definitions

Figure 8-2 Continued.

of data at a time, only eight of its pins are used for data, as opposed to 16 for the 8086. Except for pins 28 and 34 the two processors have the same control pin definitions. Pin 28 differs only in the minimum mode. For the 8088 this minimum mode signal is inverted from that of the 8086, so that the 8088 is compatible with the Intel 8085 microcomputer chip.

On the 8086, pin 34 (BHE) designates whether or not at least 1 byte of a

transfer is to be made on AD15 through AD8. A 0 on this pin indicates that the more significant data lines are to be used; otherwise, only AD7 through AD0 are used. Together the BHE and A0 signals indicate to the interfaces connected to the bus how the data are to appear on the bus. The four possible combinations are defined as follows:

| Operation | BHE | A0 | Data pins used |
|--------------------------------------|-----|----|--|
| Write/read a word at an even address | 0 | 0 | AD15-AD0 |
| Write/read a byte at an even address | 1 | 0 | AD7-AD0 |
| Write/read a byte at an odd address | 0 | 1 | AD15-AD8 |
| Write/read a word at an odd address | 0 | 1 | AD15-AD8 (First bus cycle: puts the least significant data byte on AD15-AD8) |
| | 1 | 0 | AD7-AD0 (Next bus cycle: puts the most significant data byte on AD7-AD0) |

where 0 is low and 1 is high.

Because, on the 8088, only AD7-AD0 can transfer data, this pin is not needed to indicate the upper or lower half of the data bus and is free to provide status information.

Pins 1 and 20 are grounded. Pins 2 through 16 and 39 (AD15-AD0) hold the address needed for the transfer during the first part of the bus cycle, and are free to transfer the data during the remaining part of the cycle.

Pins 17 and 18 (NMI and INTR) are for interrupt requests, which were discussed in Chap. 6. Pin 19 (CLK) is for supplying the clock signal that synchronizes the activity within the CPU.

Pin 21 (RESET) is for inputting a system reset signal. Most systems include a line that goes to all system components and a pulse is automatically sent over this line when the system is turned on, or the reset pulse can be manually generated by a switch that allows the operator to reinitialize the system. A 1 on the reset line causes the components to go to their "turn on" state. For the processor this state is having the PSW, IP, DS, SS, ES, and instruction queue cleared and CS set to FFFF. With (IP) = 0000 and (CS) = FFFF the processor will begin executing at FFFF0. Normally, this location would be in a read-only section of memory and would contain a JMP instruction to a program for initializing the system and loading the application software or operating system. Such a program is referred to as a *bootstrap loader*.

Pin 22 (READY) is for inputting an acknowledge from a memory or I/O interface that input data will be put on the data bus or output data will be accepted from the data bus within the next clock cycle. In either case, the CPU and its bus control logic can complete the current bus cycle after the next clock cycle. Pin 23

($\overline{\text{TEST}}$) is used in conjunction with the WAIT instruction and is employed primarily in multiprocessing situations (see Chap. 11). Pins 24 through 31 are mode dependent and are considered later. Pin 32 ($\overline{\text{RD}}$) indicates that an input operation is to be performed and, in minimum mode, is used along with pin 28, which distinguishes a memory transfer from an I/O transfer, and pin 29, which indicates an output operation, to determine the type of transfer.

During the first part of a bus cycle pins 35–38 (AD19/S6–AD16/S3) output the 4 high-order bits of the address, and during the remaining part of the cycle they output status information. Status bits S3 and S4 indicate the segment register that is being used to generate the address and bit S5 reflects the contents of the IF flag. S6 is always held at 0 and indicates that an 8086/8088 is controlling the system bus (see page 460).

Pin 40 (VCC) receives the supply voltage, which must be $+5\text{ V} \pm 10\%$. Systems based on an 8086 or 8088 are ordinarily designed so that only a TTL-compatible +5-V supply voltage and ground are needed, thus simplifying the design of the power supply.

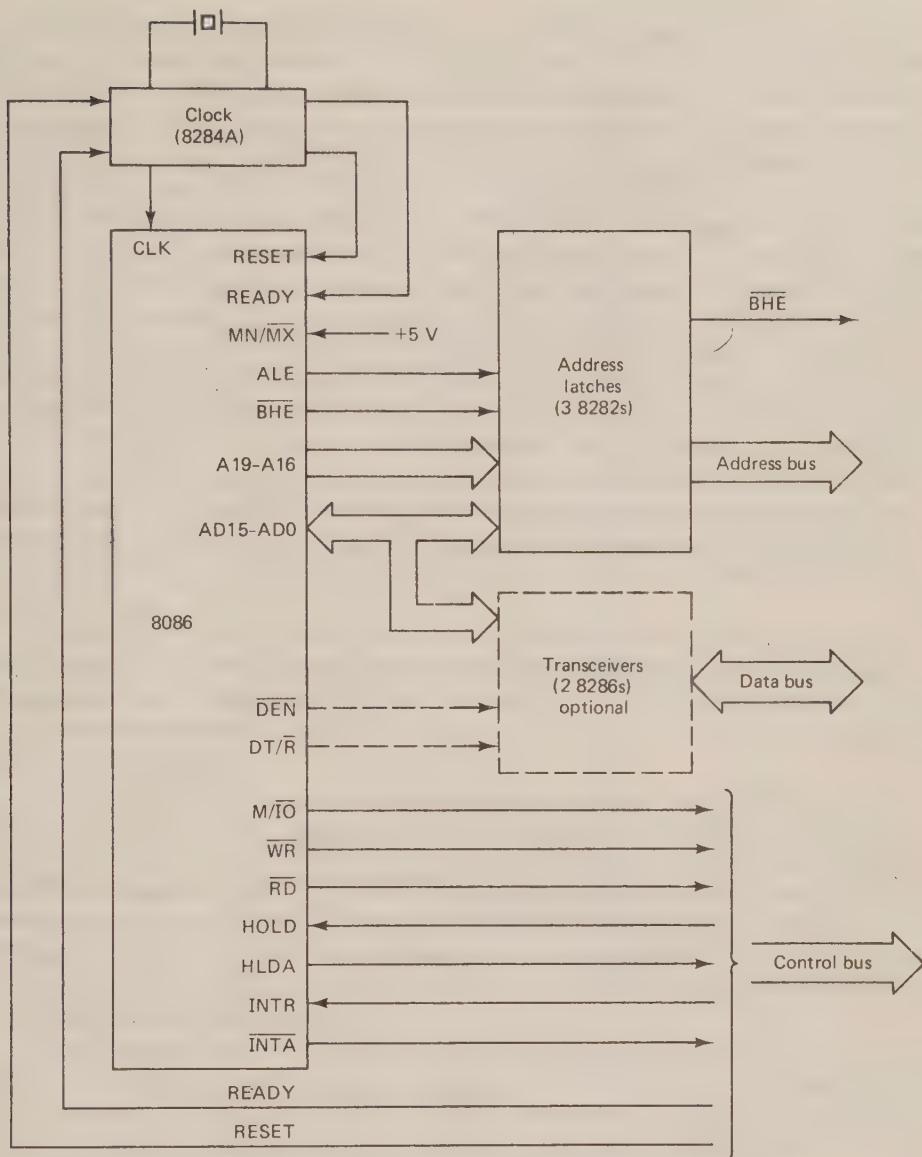
8-1-1 Minimum Mode

A processor is in minimum mode when its MN/MX pin is strapped to $+5\text{ V}$. The definitions for pins 24 through 31 for the minimum mode are given in Fig. 8-3 and a typical minimum mode configuration is shown in Fig. 8-4. The address must be

Figure 8-3 Pin definitions for the minimum mode.

| Pin | Symbol | In/Out 3-State | Description |
|-----------------|---------------------------------|-------------------|--|
| 24 | $\overline{\text{INTA}}$ | 0-3 | Indicates recognition of an interrupt request. Consists of two negative going pulses in two consecutive bus cycles. |
| 25 | ALE | 0 | Outputs a pulse at the beginning of the bus cycle and is to indicate an address is available on the address pins. |
| 26 | $\overline{\text{DEN}}$ | 0-3 | Output during the latter portion of the bus cycle and is to inform the transceivers that the CPU is ready to send or receive data. |
| 27 | DT/R | 0-3 | Indicates to the set of transceivers whether they are to transmit (1) or receive (0) data. |
| 28 ¹ | $\text{M}/\overline{\text{IO}}$ | 0-3 | Distinguishes a memory transfer from an I/O transfer. For a memory transfer it is 1. |
| 29 | $\overline{\text{WR}}$ | 0-3 | When 0, it indicates a write operation is being performed. It is used in conjunction with pins 28 ($\text{M}/\overline{\text{IO}}$) and 32 ($\overline{\text{RD}}$) to specify the type of transfer. |
| 30 | HLDA | 0 | Outputs a bus grant to a requesting master. Pins with tristate gates are put in high impedance state while $\text{HLDA}=1$. |
| 31 | HOLD | I | Receives bus requests from bus masters. The 8086/8088 will not gain control of the bus until this signal is dropped. |

¹For the 8088, the symbol is $\text{IO}/\overline{\text{M}}$ and a 1 indicates an I/O transfer.



Note: In an 8088 system \overline{BHE} is \overline{SSO} , $M/I\overline{O}$ is IO/\overline{M} , and only one 8286 is needed.

Figure 8-4 Minimum mode system.

latched since it is available only during the first part of the bus cycle. To signal that the address is ready to be latched a 1 is put on pin 25, the address latch enable (ALE) pin. Typically, the latching is accomplished using Intel 8282s, as shown in Fig. 8-5. Because 8282 is an 8-bit latch, two of them are needed for a 16-bit address and three are needed if a full 20-bit address is used. In an 8086 system, \overline{BHE} would

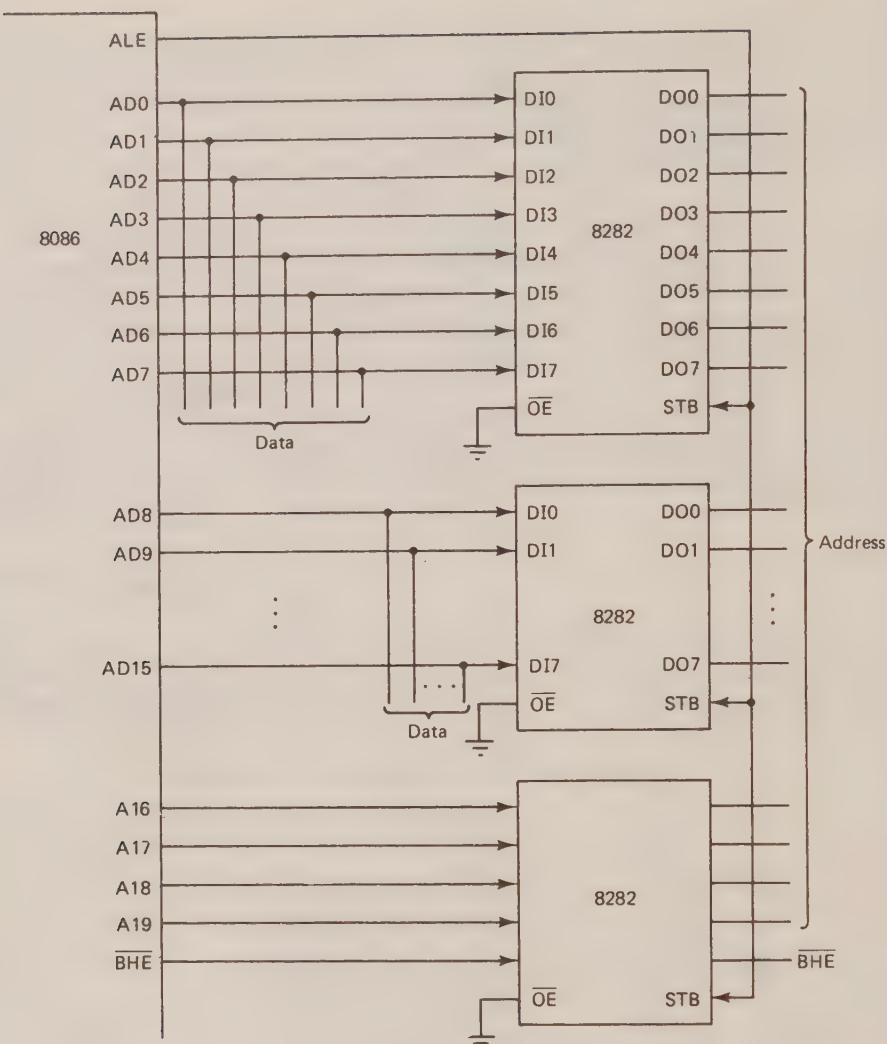


Figure 8-5 Application of 8282 latches.

also have to be latched. For a small 8088 system that has only 64k bytes of memory, only two 8282s would be required. A signal on the STB pin latches the bits applied to the input data lines DI7-DI0. Therefore, STB is connected to the 8086's ALE pin and DI7-DI0 are attached to eight of the address lines. An active low signal on the \overline{OE} enables the latch's outputs DO7-DO0, and a 1 at this pin forces the outputs into their high-impedance state. In an 8086/8088 single-processor system that does not include a DMA controller (DMA is discussed in Chap. 9) this pin is grounded.

If a system includes several interfaces, then drivers and receivers, which may

not be needed on small, single-board systems, will be required for the data lines. The Intel IC device for implementing the transceiver (driver/receiver) block shown in Fig. 8-4 is the 8286 transceiver device. The 8286 contains 16 tristate elements, eight receivers, and eight drivers. Therefore, only one 8286 is needed to service all of the data lines for an 8088, but two are required in an 8086 system. Figure 8-6 shows how 8286s are connected into a system and a logic diagram of one of its cells. The 8286 is symmetric with respect to its two sets of data pins, either the pins A7–A0 can be the inputs and B7–B0 the outputs, or vice versa. The output enable (\overline{OE}) pin determines whether or not data are allowed to pass through the 8286 and the transmit (T) pin controls the direction of the data flow. When $\overline{OE} = 1$, data are not transmitted through the 8286 in either direction. If it is 0, then T = 1 causes A7–A0 to be the inputs and T = 0 results in B7–B0 being the inputs. In an 8086/8088-based system the \overline{OE} pin would be connected to the DEN pin, which is active low whenever the processor is performing an I/O operation. The A7–A0 pins are connected to the appropriate address/data lines and the T pin is tied to the processor's DT/R pin. Thus, when the processor is outputting the data flow is from A7–A0 to B7–B0, and when it is inputting the flow is in the other direction. The processor floats the \overline{DEN} and DT/R pins in response to a bus request on the HOLD pin.

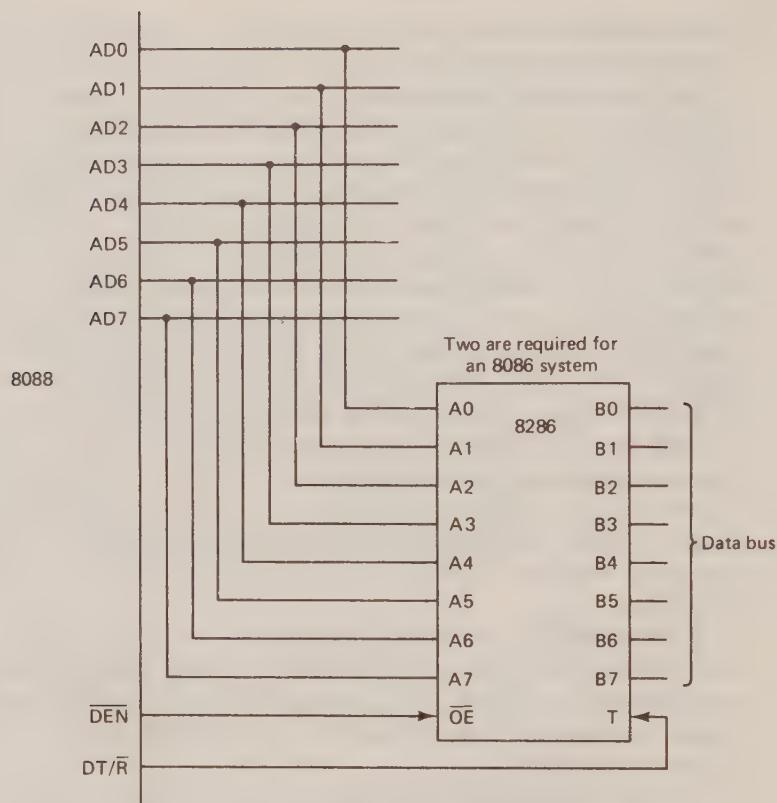
Sometimes a system bus is designed so that the address and/or data signals are inverted. Therefore, the 8282 and 8286 both have companion chips that are the same as the 8282 and 8286 except that they cause an inversion between their inputs and outputs. The companion for the 8282 is the 8283 and the companion for the 8286 is the 8287.

The third component, other than the processor, that appears in Fig. 8-4 is an 8284A clock generator. This device, which is actually more than just a clock, is detailed in Fig. 8-7. In addition to supplying a train of pulses at a constant frequency it synchronizes ready (RDY) signals, which indicate an interface is ready to complete a transfer, and reset (RES) signals, which initialize the system, with the clock pulses. Although these two signals may be sent at any time, the 8284A will not reflect them in its READY and RESET outputs until the trailing edge of the clock pulse in which they are received.

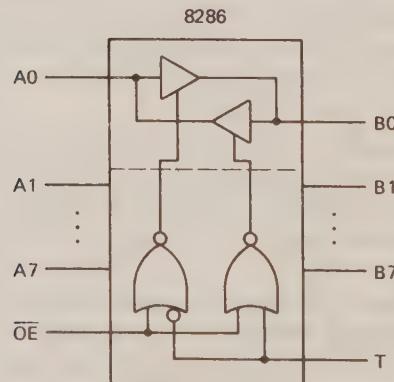
The frequency source applied to the 8284A may be from a pulse generator that is connected to the EFI pin or an oscillator that is connected across X1 and X2. If the input to F/C is 1, then the EFI input determines the frequency; otherwise, it is the oscillator input. In either case the clock output CLK is one-third of the input frequency.

All three of the devices considered above, the 8282, 8286, and 8284A, require only +5-V supply voltages. Their inputs and outputs are TTL compatible and, therefore, the devices are compatible with each other and with the 8086 and 8088. Many of the details of these devices have not been included here and, for more information, one should refer to the Intel manuals—see Reference 1. This comment applies particularly to the 8284A clock.

In a minimum system the control lines do not need to be passed through transceivers, but can be used directly. The M/IO, RD, and WR lines specify the



(a) 8088 connections



(b) Internal logic

Figure 8-6 Application and internal logic of an 8286.

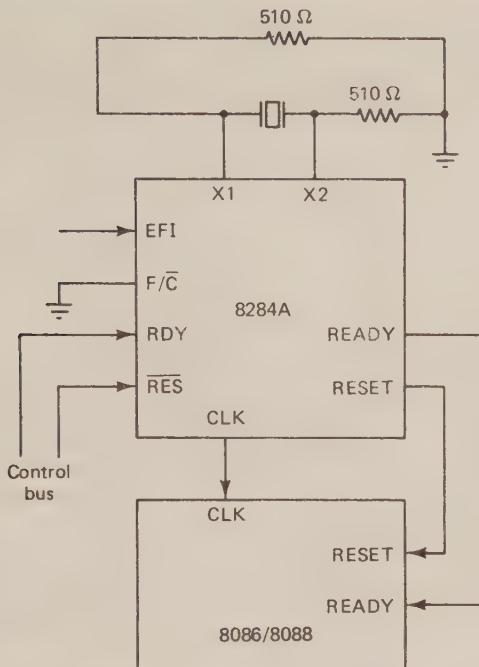


Figure 8-7 Typical 8284A clock connection.

type of transfer according to the following table:

| M/I \bar{O} | \bar{RD} | \bar{WR} | |
|---------------|------------|------------|--------------|
| 0 | 0 | 1 | I/O read |
| 0 | 1 | 0 | I/O write |
| 1 | 0 | 1 | Memory read |
| 1 | 1 | 0 | Memory write |

where 0 is low and 1 is high.

The purposes of the bus request (HOLD), bus grant (HLDA), interrupt request (INTR), and interrupt acknowledge (INTA) lines were discussed in Chap. 6. The INTA signal consists of two negative pulses output during two consecutive bus cycles. The first pulse informs the interface that its request has been recognized, and upon receipt of the second pulse, the interface is to send the interrupt type to the processor over the data bus.

8-1-2 Maximum Mode

A processor is in maximum mode when its MN/MX pin is grounded. The maximum mode definitions of pins 24 through 31 are given in Fig. 8-8 and a typical maximum mode configuration is shown in Fig. 8-9. It is clear from Fig. 8-9 that the main difference between minimum and maximum mode configurations is the need for additional circuitry to translate the control signals. This circuitry is for converting the status bits S_0 , S_1 , and S_2 into the I/O and memory transfer signals needed to

| Pin | Symbol | In/Out 3-State | Description |
|----------|--|-------------------|---|
| 24,25 | QS1, QSO | 0 | Reflects the status of the instruction queue. This status indicates the activity in the queue during the previous clock cycle - see Chap. 11. |
| 26,27,28 | $\overline{S_0}, \overline{S_1}, \overline{S_2}$ | 0-3 | Indicates the type of transfer to take place during the current bus cycle: |
| | | | $\begin{array}{ccc} \overline{S_2} & \overline{S_1} & \overline{S_0} \\ \hline 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array}$ Interrupt acknowledge Read I/O port Write I/O port Halt Instruction fetch Read memory Write memory Inactive - passive |
| | | | (1 represents high and 0 represents low.) The status becomes active prior to the beginning of a bus cycle and returns to inactive during the later part of the cycle. |
| 29 | <u>LOCK</u> | 0-3 | Indicates the bus is not to be relinquished to other potential bus masters. It is initiated by a <u>LOCK</u> instruction prefix and is maintained until the end of the next instruction - see Chap. 11. It is also active during and between the two <u>INTA</u> pulses. |
| 30 | <u>RQ/GT1</u> | I/O | For inputting bus requests and outputting bus grants. |
| 31 | <u>RQ/GT0</u> | I/O | Same as <u>RQ/GT1</u> except that a request on <u>RQ/GT0</u> has higher priority. |

Note: In maximum mode the 8086 and 8088 pins have the same definitions except for pin 34, which on the 8088 is always 1.

Figure 8-8 Maximum mode pin definitions.

direct data transfers, and for controlling the 8282 latches and 8286 transceivers. It is normally implemented with an Intel 8288 bus controller. Also included in the system is an interrupt priority management device; however, its presence is optional.

The $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ status bits specify the type of transfer that is to be carried out and when used with an 8288 bus controller they obviate the need for the M/IO (or IO/M), WR, INTA, ALE, DT/R, and DEN signals that are output over pins 24 through 29 when the processor is operating in minimum mode. Except for the case $S_1 = \overline{S_0} = 1$, $S_2 = 0$ indicates a transfer between an I/O interface and the CPU and $S_2 = 1$ implies a memory transfer. The $\overline{S_1}$ bit specifies whether an input or output is to be performed. From the status the 8288 is able to originate the address latch enable signal to the 8282s, the enable and direction signals to the 8286 transceivers, and the interrupt acknowledge signal to the interrupt controller.

The QS0 and QS1 pins are to allow the system external to the processor to interrogate the status of the processor instruction queue so that it can determine which instruction it is currently executing, and the LOCK pin indicates that an instruction with a LOCK prefix is being executed and the bus is not to be used by

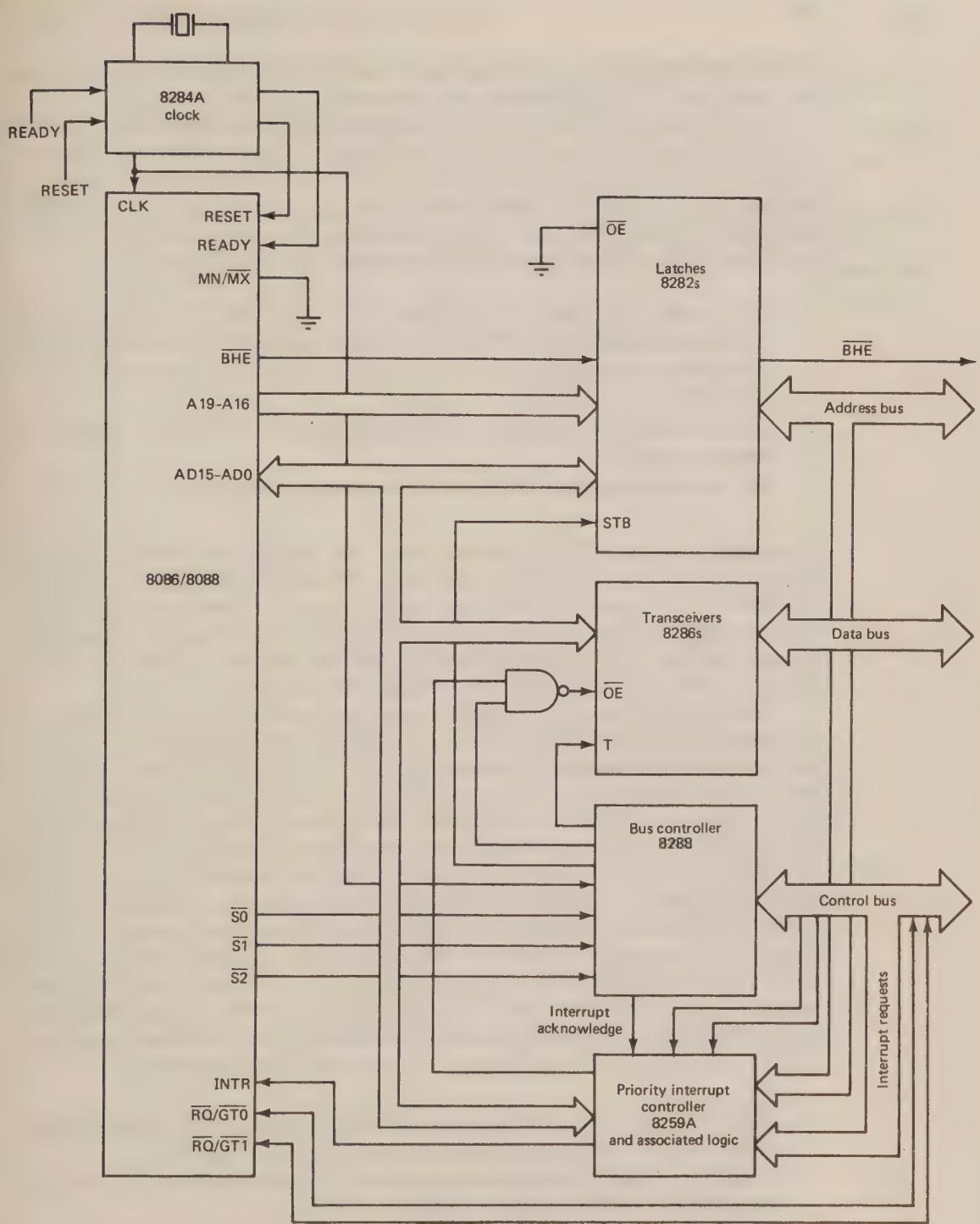


Figure 8-9 Typical maximum mode configuration.

another potential master. These pins are needed only in multiprocessor systems and, along with the LOCK prefix, are discussed in detail in Chap. 11.

The HOLD and HLDA pins become the $\overline{RQ}/GT0$ and $\overline{RQ}/GT1$ pins. Both bus requests and bus grants can be given through each of these pins. They are exactly the same except that if requests are seen on both pins at the same time, then the one on $\overline{RQ}/GT0$ is given higher priority. A request consists of a negative pulse arriving before the start of the current bus cycle. The grant is a negative pulse that is issued at the beginning of the current bus cycle provided that:

1. The previous bus transfer was not the low byte of a word to or from an odd address if the CPU is an 8086. For an 8088, regardless of the address alignment, the grant signal will not be sent until the second byte of a word reference is accessed.
2. The first pulse of an interrupt acknowledgment did not occur during the previous bus cycle.
3. An instruction with a LOCK prefix is not being executed.

If condition 1 or 2 is not met, then the grant will not be given until the next bus cycle, and if condition 3 is not met, the grant will wait until the locked instruction is completed. In response to the grant the three-state pins are put in their high-impedance state and the next bus cycle will be given to the requesting master. The processor will be effectively disconnected from the system bus until the master sends a second pulse to the processor through the \overline{RQ}/GT pin.

An expanded view of a maximum mode system which shows only the connections to an 8288 is given in Fig. 8-10. The $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ pins are for receiving the corresponding status bits from the processor. The ALE, DT/R, and DEN pins provide the same outputs that are sent by the processor when it is in minimum mode (except that DEN is inverted from \overline{DEN}). The CLK input permits the bus controller activity to be synchronized with that of the processor. The \overline{AEN} , IOB, and CEN pins are for multiprocessor systems and are considered in Chap. 11. In a single-processor system AEN and IOB are normally grounded and a 1 is applied to CEN. The meaning of the MCE/PDEN output depends on the mode, which is determined by the signal applied to IOB. When IOB is grounded it assumes its master cascade enable (MCE) meaning and can be used to control cascaded 8259As (see Sec. 8-3). In the event that +5 V is connected to IOB, the peripheral data enable (\overline{PDEN}) meaning, which is used in multiple-bus configurations, is assumed.

The remaining pins given in Fig. 8-10 have the following definitions:

INTA—Issues the two interrupt acknowledgment pulses to a priority interrupt controller or an interrupting device when $S_0 = S_1 = S_2 = 0$

IORC (I/O Read Command)—Instructs an I/O interface to put the data contained in the addressed port on the data bus.

IOWC (I/O Write Command)—Instructs an I/O interface to accept the data on the data bus and put the data into the addressed port.

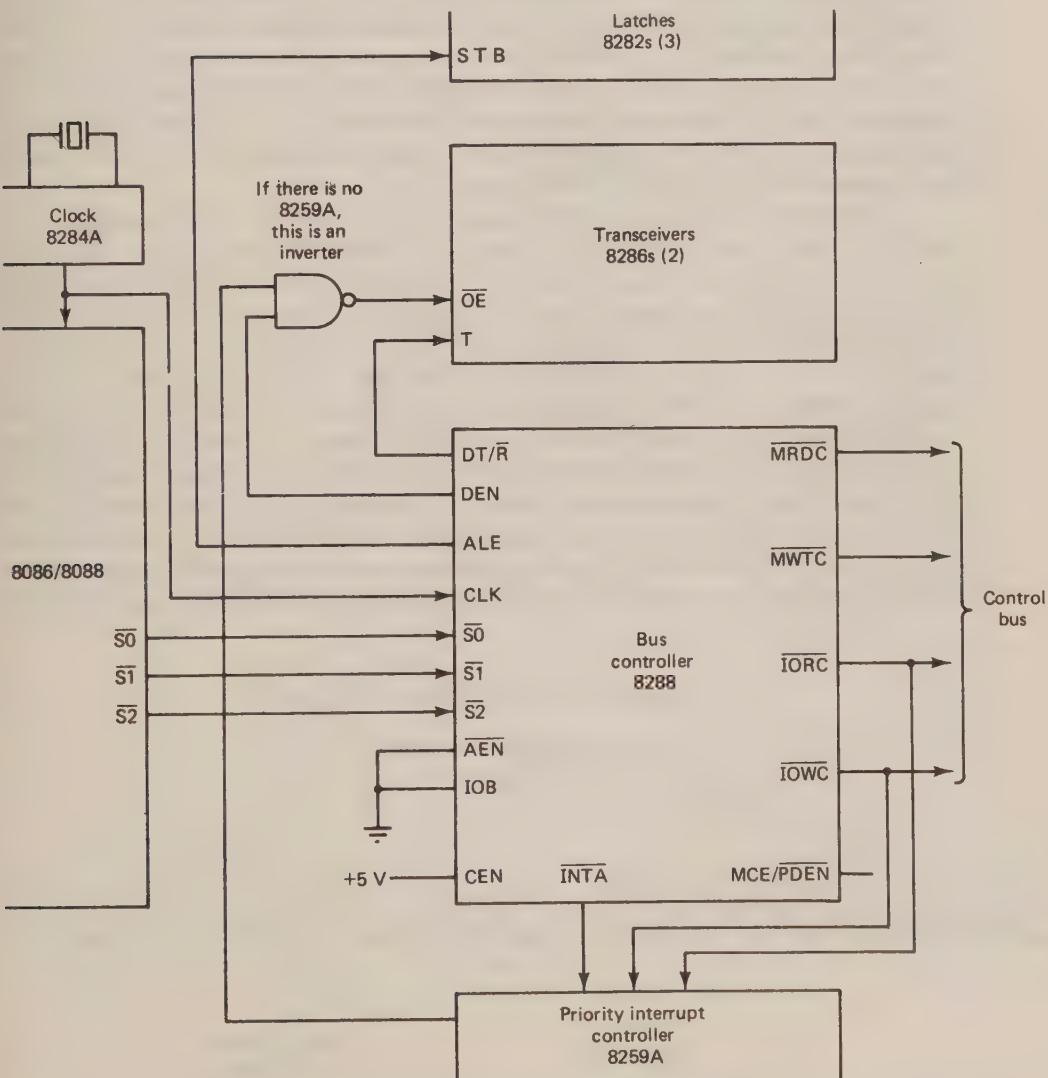
MRDC (Memory Read Command)—Instructs the memory to put the contents of the addressed location on the data bus.

MWTC (Memory Write Command)—Instructs the memory to accept the data on the data bus and put the data into the addressed memory location.

These signals are active low and are output during the middle portion of a bus cycle. Clearly, only one of them will be issued during any given bus cycle.

Not shown in Fig. 8-10 are the **AIOWC** (advanced I/O write command) and **AMWC** (advanced memory write command) pins. They serve the same purposes

Figure 8-10 Connections to an 8288 bus controller.



as the IOWC and MWTC pins except that they are activated one clock pulse sooner. This gives slow interfaces an extra clock cycle to prepare to input the data.

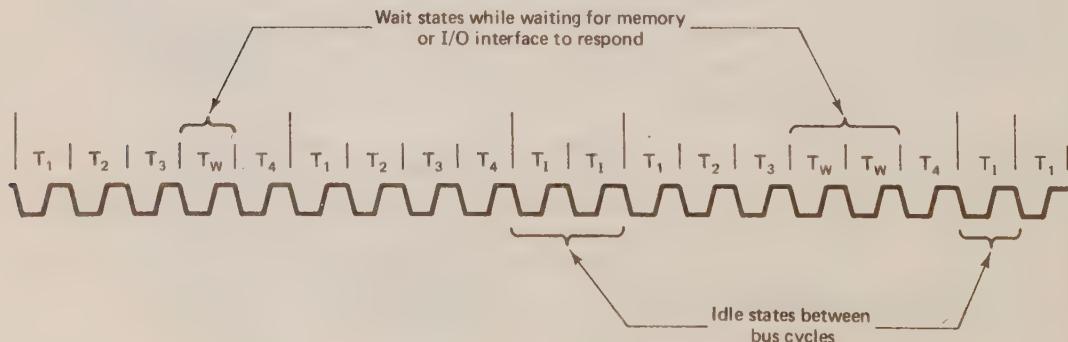
As with the other 8086 supporting devices, the 8288 requires a +5-V supply voltage and has TTL-compatible inputs and outputs. For more detailed information one should refer to the Intel manuals—see Reference 1.

8-2 SYSTEM BUS TIMING

Until now we have vaguely referred to the first part, middle part, and last part of the bus cycle. It is the objective of this section to put the discussion of timing on a more precise footing. The length of a bus cycle in an 8086/8088 system is four clock cycles, denoted T_1 through T_4 , plus an indeterminate number of wait state clock cycles, denoted T_w . If the bus is to be inactive after the completion of a bus cycle, then the gap between successive cycles is filled with idle state clock cycles represented by T_1 . Wait states are inserted between T_3 and T_4 when a memory or I/O interface is not able to respond quickly enough during a transfer. A typical succession of bus cycles is given in Fig. 8-11.

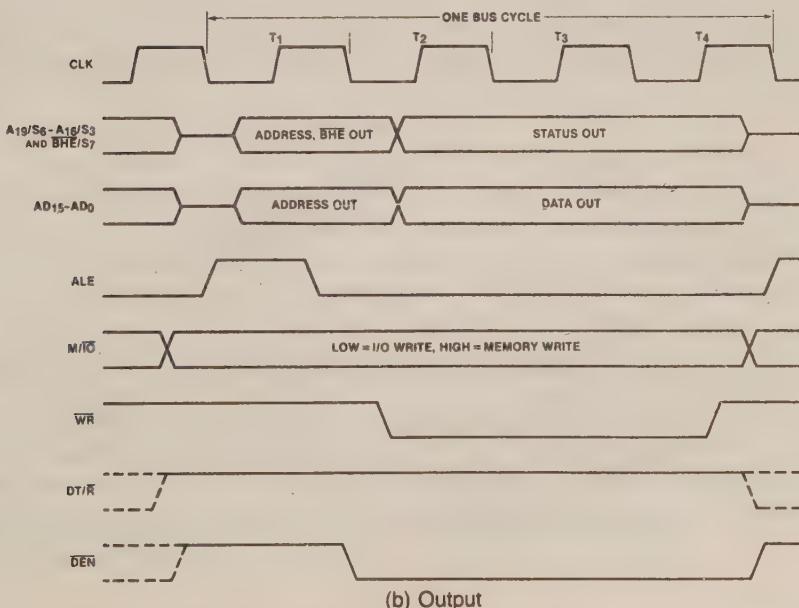
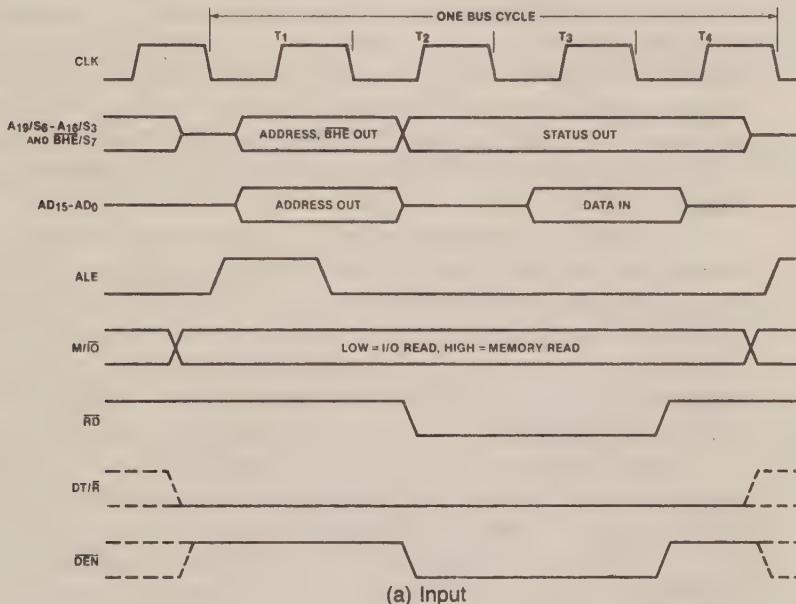
The timing diagrams for 8086 minimum mode input and output transfers that require no wait states are shown in Fig. 8-12. When the state of the processor is such that it is ready to initiate a bus cycle it applies a pulse to the ALE pin during T_1 . Before the trailing edge of the ALE signal the address, BHE, M/IO, DEN, and DT/R signals should be stable, with DEN = 1 and DT/R = 0 for an input and DT/R = 1 for an output. At the trailing edge of the ALE signal the 8288 latch the address. During T_2 the address is dropped and S3 through S7 are output on AD16/S3–AD19/S6 and BHE/S7, and DEN is lowered to enable the 8286 transceivers. If an input is being conducted, RD is activated low during T_2 and AD15–AD0 enter a high-impedance state in preparation for input. If the memory or I/O interface can perform the transfer immediately, there are no wait states and the data are put on the bus during T_3 . After the input data are accepted by the processor, RD is raised to 1 at the beginning of T_4 and, upon detecting this transition, the

Figure 8-11 Typical sequence of bus cycles.



memory or I/O interface will drop its data signals. For an output, the processor applies the $\overline{WR} = 0$ signal and then the data during T_2 , and in T_4 \overline{WR} is raised and the data signals are dropped. For either an input or output, DEN is raised

Figure 8-12 8086 minimum mode bus timing diagrams (Reprinted by permission of Intel Corporation. Copyright 1979.)



Note: For an 8088, M/I/O is IO/M and BHE/S7 becomes SSO which is present throughout the bus cycle (i.e., it changes at the same time as IO/M). Also, only AD7-ADO carry data.

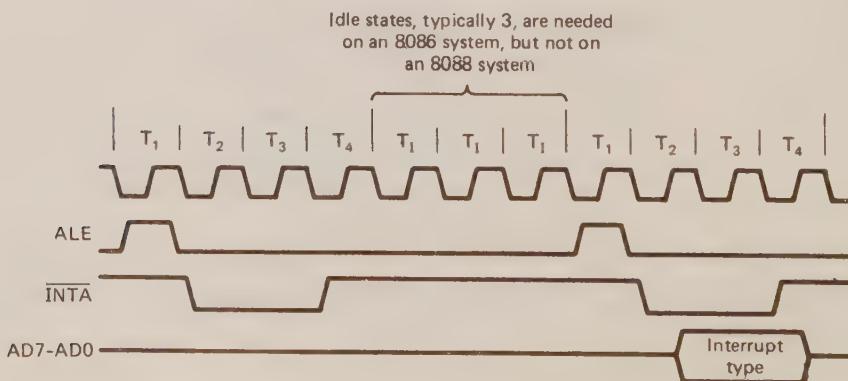
during T_4 to disable the transceivers and the M/\overline{IO} signal is set according to the next transfer at this time or during a subsequent T_1 state.

The bus timing has been designed so that the memory or I/O interface involved in a transfer can control when data are to be placed on or taken from the bus by the interface. This is done by having the interface send a READY signal to the processor (perhaps via an 8284A) when it has made data available or accepted data. If a READY signal has not been received by the processor by the beginning of T_3 , then one or more T_w states will be inserted between T_3 and T_4 until a READY has been received. The bus activity during T_w is the same as during T_3 . A signal applied to an RDY input of an 8284A will cause a READY output to the processor at the trailing edge of the current clock cycle; therefore, if a wait state is to be avoided, an RDY input must be received before the beginning of the T_3 clock cycle.

The timing diagram for an interrupt acknowledge is shown in Fig. 8-13. If an interrupt request has been recognized during the previous bus cycle and an instruction has just been completed, then a negative pulse will be applied to INTA during the current bus cycle and the next bus cycle. Each of these pulses will extend from T_2 to T_4 . Upon receiving the second pulse, the interface accepting the acknowledgement will put the interrupt type on AD7-AD0, which are floated the rest of the time during the two bus cycles. The type will be available from T_2 to T_4 .

Figure 8-14 shows the timing of a bus request and bus grant in a minimum mode system. The HOLD pin is tested at the leading edge of each clock pulse. If a HOLD signal is received by the processor before T_4 or during a T_1 state, then the CPU activates HLDA and the succeeding bus cycles will be given to the requesting master until that master drops its request. The lowered request is detected at the rising edge of the next clock cycle and the HLDA signal is dropped at the trailing edge of that clock cycle. While HLDA is 1, all of the processor's three-state outputs are put in their high-impedance state. Instructions already in

Figure 8-13 Interrupt acknowledgment.



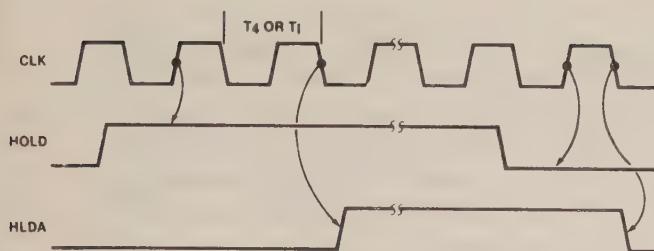


Figure 8-14 Bus request and bus grant timing on a minimum mode system
(Reprinted by permission of Intel Corporation. Copyright 1979.)

the instruction queue will continue to be executed until one of them requires the use of the bus. The instruction

MOV AX,BX

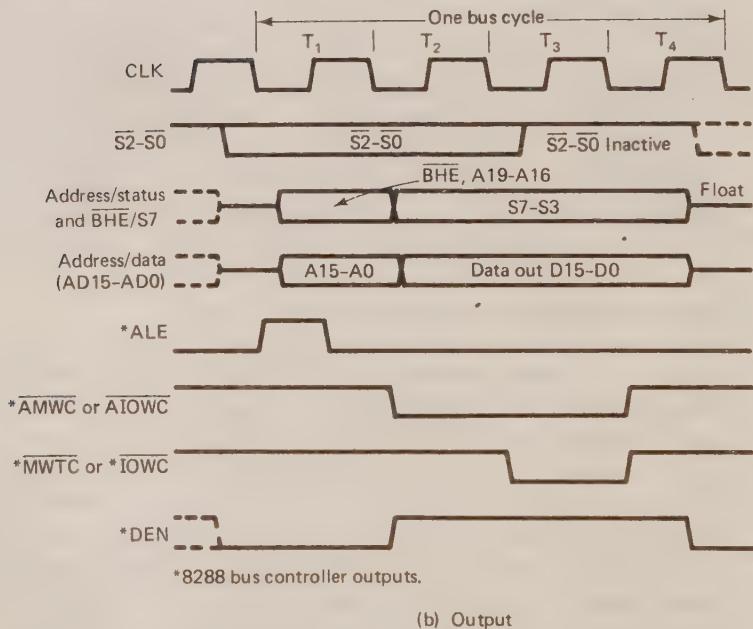
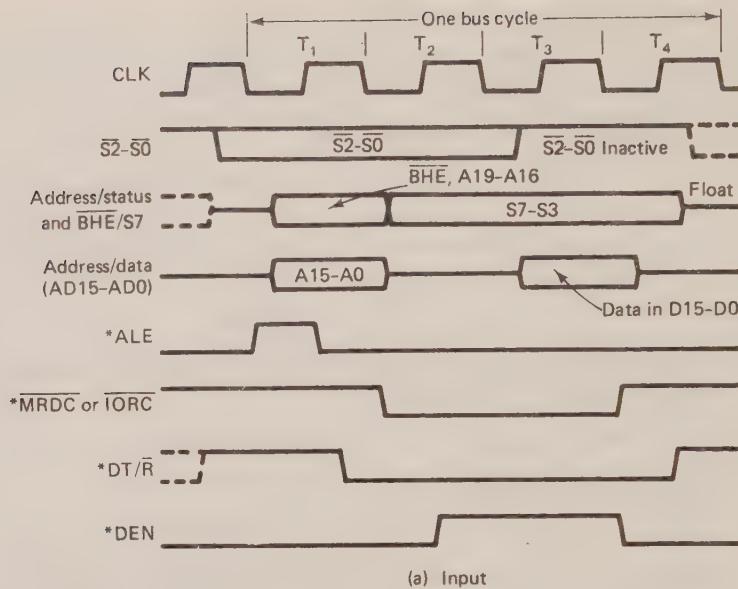
could execute completely, but

MOV AX,NUMBER

would only execute until it is necessary to bring in data from the location NUMBER.

The timing diagrams for input and output transfers on a maximum mode system are given in Fig. 8-15. The $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ bits are set just prior to the beginning of the bus cycle. Upon detecting a change from the passive $\overline{S_0} = \overline{S_1} = \overline{S_2} = 1$ state, the 8288 bus controller will output a pulse on its ALE pin and apply the appropriate signal to its DT/R pin during T_1 . In T_2 , the 8288 will set DEN = 1, thus enabling the transceivers, and, for an input, will activate either MRDC or IORC. These signals will be maintained until T_4 . For an output, the AMWC or AIOWC is activated from T_2 to T_4 and the MWTC or IOWC is activated from T_3 to T_4 . The status bits $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ will remain active until T_3 and will become passive (all 1s) during T_3 and T_4 . As with the minimum mode, if the READY input is not activated before the beginning of T_3 , wait states will be inserted between T_3 and T_4 .

Interrupt acknowledgment signals are the same as in the minimum mode case except that a 0 is applied to the LOCK pin from T_2 of the first bus cycle to T_2 of the second bus cycle. Bus requests and grants are handled differently, however, and the timing on an $\overline{RQ}/\overline{GT}$ pin is shown in Fig. 8-16. A request/grant/release is accomplished by a sequence of three pulses. The $\overline{RQ}/\overline{GT}$ pins are examined at the rising edge of each clock pulse and if a request is detected (and the necessary conditions discussed previously are met), the processor will apply a grant pulse to the $\overline{RQ}/\overline{GT}$ immediately following the next T_4 or T_1 state. When the requesting master receives this pulse it seizes control of the bus. This master may control the bus for only one bus cycle or for several bus cycles. When it is ready to relinquish the bus it will send the processor the release pulse over the same line that it made its request. As noted before, $\overline{RQ}/\overline{GT}_0$ and $\overline{RQ}/\overline{GT}_1$ are the same except that $\overline{RQ}/\overline{GT}_0$ has higher priority.



Note: For an 8088, $\overline{\text{BHE}}$ is not present and data are transferred only over AD7-AD0.

Figure 8-15 Timing diagrams for a maximum-mode system.

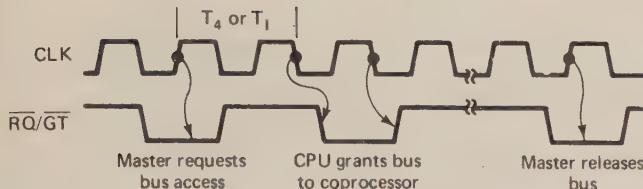


Figure 8-16 Timing for maximum mode bus requests and grants. (Reprinted by permission of Intel Corporation. Copyright 1979.)

8-3 INTERRUPT PRIORITY MANAGEMENT

The interrupt priority management logic indicated in Fig. 8-1 can be implemented in several ways. It does not need to be present in systems which use software priority management or simple daisy chaining, but more complex systems may require the efficiency gained by including hardware for managing the I/O interrupts. Many manufacturers have made priority management devices available and Intel is no exception. Although such a device made by one manufacturer could be used with processors made by other manufacturers, generally there are fewer compatibility problems if the CPU and interrupt priority device are produced by the same company. Therefore, we will be concerned with the Intel 8259A programmable interrupt controller (PIC), which has been specifically designed to work with the 8086/8088 as well as other members of the Intel microprocessor family.

The 8259A has been designed so that it can operate alone or in concert with other 8259As. In order to limit the initial discussion, an interrupt system involving a single 8259A device is considered first; then the discussion is extended to systems that can include as many as nine 8259As.

8-3-1 Interrupt System Based on a Single 8259A

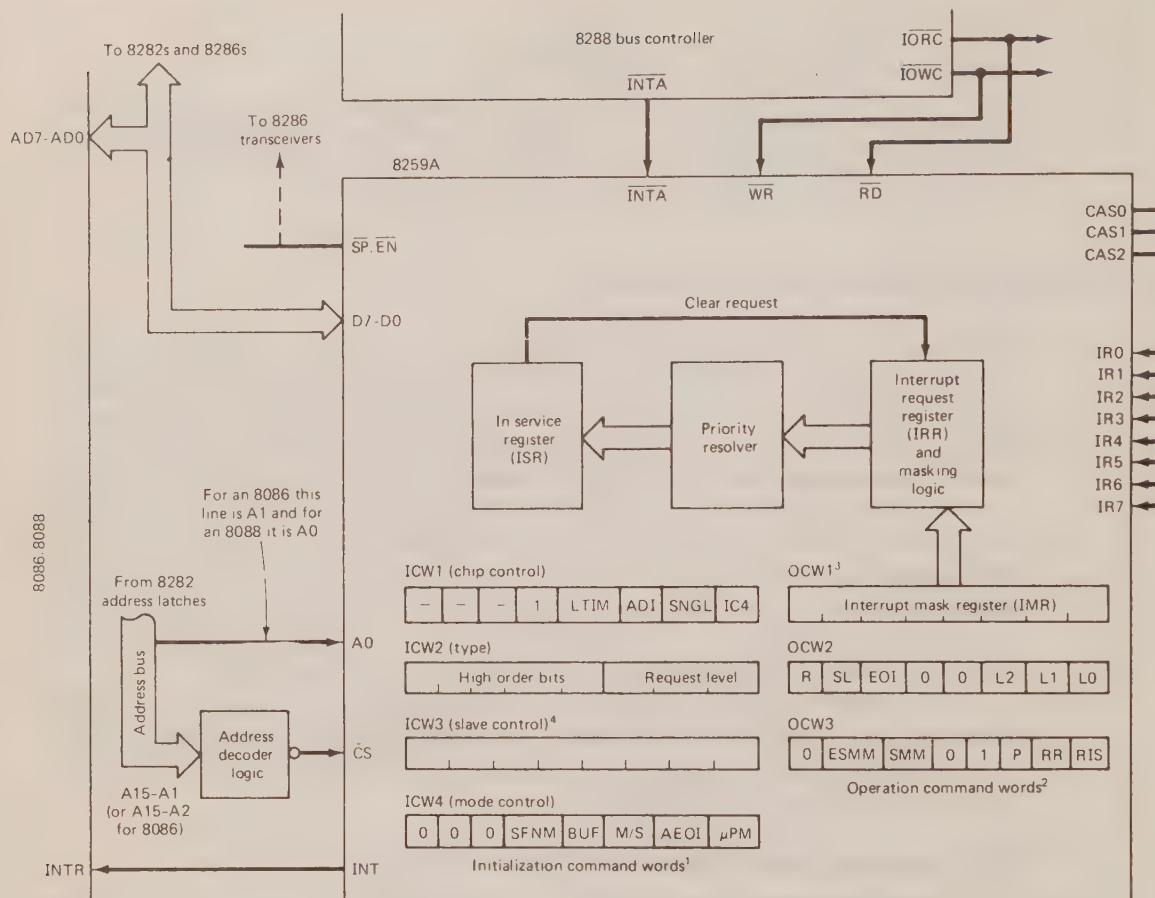
The 8259A is contained in a 28-pin dual-in-line package that requires only a +5-V supply voltage. Its organization is shown in Fig. 8-17 along with its connections to a maximum mode system. Its pins (other than the supply voltage and ground pins) are defined as follows:

D7-D0—For communicating with the CPU over the data bus. On a few systems bus drivers may be needed, but on other systems direct connections can be used.

INT—To send interrupt request signals to the CPU.

INTA—To receive interrupt acknowledge signals from the CPU. The 8259A assumes that an acknowledgment consists of two negative pulses, thus making it compatible with 8086/8088 systems.

RD—To signal the 8259A that it is to place the contents of the IMR, ISR, or IRR register or a priority level on the data bus. Which of these possibilities is placed on the bus depends on the state of the 8259A and is discussed below.



¹ A0 = 0 for addressing the first word (ICW1) and 1 for addressing the succeeding words.

³ Bits correspond to IR inputs: Bit = 1 means IR is masked and Bit = 0 means it is not masked.

² A0 = 1 for addressing the first word and 0 for addressing the succeeding words.

⁴ If 8259A is a master, Bit = 1 indicates that the corresponding IR input is connected to a slave. For a slave, bits 3-7 are 0 and bits 0-2 identify the slave.

Figure 8-17 Organization of the 8259A programmable interrupt controller.

WR—To signal the 8259A that it is to accept data from the data bus and use the data to set the bits in the command words. How the received data are distributed is discussed later.

CS—For indicating that the 8259A is being accessed. This pin is connected to the address bus through the decoder logic that compares the high-order bits of the address of the 8259A with the address currently on the address bus. Input to this pin can be combined with $\overline{S2}$ to give the ready signal.

A0—For indicating which port of the 8259A is being accessed. Two addresses must be reserved in the I/O address space for each 8259A in the system.

IR7-IR0—For receiving interrupt requests from I/O interfaces or other 8259As referred to as slaves.

CAS2-CAS0—To identify a particular slave device. Their function is considered in Sec. 8-3-2.

SP/EN—For one of two purposes; either as an input to determine whether the 8259A is to be a master ($\overline{SP/EN} = 1$) or as a slave ($\overline{SP/EN} = 0$), or as an output to disable the data bus transceivers when data are being transferred from the 8259A to the CPU. Whether the $\overline{SP/EN}$ pin is used as an input or output depends on the buffer mode discussed below.

For an 8088 the two addresses associated with an 8259A are normally consecutive, and the A0 line is connected to the A0 pin, but because there are only eight data pins on the 8259A and the 8086 always inputs the interrupt pointer from the lower 8 bits of its 16-bit data bus, all data transfers to and from the 8259A must be made over the lower byte of the bus. The easiest way to guarantee that all transfers will use the lower half of the bus is to connect the A1 line to A0 and use two consecutive even addresses, with the first being divisible by 4. However, to simplify the following discussion, the second address will be referred to as the odd address for both cases.

The control portion of the 8259A contains several programmable bits that can be viewed as being contained in seven 8-bit registers. These registers are divided into two groups, with one group containing the initialization command words (ICWs) and the other group containing the operation command words (OCWs). The *initialization command words* are normally set by an initialization routine when the computer system is first brought up and remain constant throughout its operation. By contrast, the *operation command words* are used to dynamically control the processing of interrupts.

The IRR (and its associated masking logic), priority resolver, and ISR are for receiving and controlling the interrupts that arrive at the IR7-IR0 pins. The IRR latches the incoming requests and, in conjunction with the priority resolver, allows unmasked requests with sufficient priority to put a 1 on the INT pin. The priority resolver logic determines the priorities of the requests in the IRR and the ISR is for holding the requests currently being processed.

After a bit in the IRR is set to 1 it is compared with the corresponding mask bit in the IMR. If the mask bit is 0, the request is passed on to the priority resolver, but if it is 1, the request is blocked. When an interrupt request is input to the priority resolver its priority is examined and, if according to the current state of the priority resolver the interrupt is to be sent to the CPU, the INT line is activated.

Assuming that the IF flag in the CPU is 1, the CPU will enter its interrupt sequence at the completion of the current instruction and return two negative pulses over the INTA line. Upon the arrival of the first pulse, the IRR latches are disabled so that the IRR will ignore further signals on the IR7-IR0 lines. This state is maintained until the end of the second INTA pulse. Also, the first INTA pulse will cause the appropriate ISR bit to be set and the corresponding IRR bit to be cleared. The second INTA pulse causes the current contents of ICW2 to be placed

on D7–D0, and the CPU uses this byte as the interrupt type. If the automatic end of interrupt (AEOI) bit in ICW4 is 1, at the end of the second INTA pulse the ISR bit that was set by the first INTA pulse is cleared; otherwise, the ISR bit is not cleared until the proper end of interrupt (EOI) command is sent to OCW2.

As indicated above, the initialization command words are normally filled by an initializing routine when the system is turned on and contain the control bits that are held constant throughout the system's operation. The 8259A has an even address ($A_0 = 0$) and an odd address ($A_0 = 1$) associated with it and the initialization command words must be filled consecutively by using the even address for ICW1 and the odd address for the remaining ICWs.

The definitions of the bits in ICW1 are:

Bits 7–5—Not used in an 8086/8088 system, only in an 8080 or 8085 system.

Bit 4—Always set to 1. It directs the received byte to ICW1 as opposed to OCW2 or OCW3, which also use the even address ($A_0 = 0$).

Bit 3 (LTIM)—Determines whether the edge-triggered mode ($LTIM = 0$) or the level-triggered mode ($LTIM = 1$) is to be used. The edge-triggered mode causes the IRR bit to be cleared when the corresponding ISR bit is set.

Bit 2 (ADI)—Not used in an 8086/8088 system, only in an 8080 or 8085 system.

Bit 1 (SNGL)—Indicates whether or not the 8259A is cascaded with other 8259As. SNGL = 1 when only one 8259A is in the interrupt system.

Bit 0 (IC4)—Is set to 1 if an ICW4 is to be output to during the initialization sequence. For an 8086/8088 system this bit must always be set to 1 because bit 0 in ICW4 must be set to 1.

Bits 7–3 of ICW2 are filled from bits 7–3 of the second byte output by the CPU during the initialization of the 8259A, and bits 2–0 are set according to the level of the interrupt request, e.g., a request on IR6 would cause them to be set to 110. ICW3 is significant only in systems including more than one 8259A and is output to only if SNGL = 0. This case is discussed in Sec. 8-3-2. ICW4 is output to only if IC4 (ICW1) is set to 1; otherwise, the contents of ICW4 is cleared. The bits in ICW4 are defined as follows:

Bits 7–5—Always set to 0.

Bit 4 (SFNM)—If set to 1, the special fully nested mode is used. This mode is utilized in systems having more than one 8259A and is discussed below.

Bit 3 (BUF)—BUF = 1 indicates that the $\overline{SP}/\overline{EN}$ is to be used as an output to disable the system's 8286 transceivers while the CPU inputs data from the 8259A. If no transceivers are present, BUF should be set to 0 and, in systems involving only one 8259A, a 1 should be applied to the $\overline{SP}/\overline{EN}$ pin.

Bit 2 (M/S)—This bit is ignored if BUF = 0. For a system having only one 8259A, this bit should be 1; otherwise, it should be 1 for the master and 0 for the slaves.

Bit 1 (AEOI)—If AEOI = 1, then the ISR bit that caused the interrupt is cleared at the end of the second INTA pulse.

Bit 0 (μ PM)— μ PM = 1 indicates the 8259A is in an 8086/8088 system. This bit being 0 implies an 8080 or 8085 system.

A typical program sequence for setting the contents of the ICWs, which assumes that the even address of the 8259A is 0080, is:

```

MOV AL,13H
OUT 80H,AL
MOV AL,18H
OUT 81H,AL
MOV AL,0DH
OUT 81H,AL

```

The first two instructions cause the requests to be edge triggered, denote that only one 8259A is used, and inform the 8259A that an ICW4 will be output. The next two instructions cause the 5 most significant bits of the interrupt type to be set to 00011. ICW3 is not output to because SNGL = 1; therefore, the last two instructions set ICW4 to 0D, which informs the 8259A that the special fully nested mode is not to be used, the $\overline{SP/EN}$ is used to disable transceivers, the 8259A is a master, EOI commands must be used to clear the ISR bit, and the 8259A is part of an 8086/8088 system.

There are three OCWs. The command word OCW1 is used for masking interrupt requests; when the mask bit corresponding to an interrupt request is 1, then the request is blocked. OCW2 and OCW3 are for controlling the mode of the 8259A and receiving EOI commands. A byte is output to OCW1 by using the odd address associated with the 8259A and bytes are output to OCW2 and OCW3 by using the even addresses. OCW2 is distinguished from OCW3 by the contents of bit 3 of the data byte. If bit 3 is 0, the byte is put in OCW2, and if it is 1, it is put in OCW3. Both OCW2 and OCW3 are distinguished from ICW1, which also uses the even address, by the contents of bit 4 of the data. If bit 4 is 0, then the byte is put on OCW2 or OCW3 according to bit 3. There is no ambiguity in ICW2, ICW3, ICW4, and OCW1 all using the odd address because the initialization words must always follow ICW1 as dictated by the initialization sequence, and an output to OCW1 cannot occur in the middle of this sequence.

Referring back to Fig. 8-17, bits L2-L0 of OCW2 are for designating an IR level, bit 5 is for giving EOI commands, and bits 6 and 7 are for controlling the IR levels. Recall that when the AEOI bit in ICW4 is 1, the ISR bit, which is set by the interrupt request, is reset automatically at the end of the second INTA pulse, but if AEOI = 0, then the ISR bit must be explicitly cleared by an EOI command, which consists of sending an OCW2 with bit 5 equal to 1. When an EOI command is given the meanings of the four possible combinations of bit 7, the R (rotate) bit, and bit 6, the SL (set level) bit, are:

| R | SL | |
|---|----|---|
| 0 | 0 | Nonspecific, normal priority mode |
| 0 | 1 | Specifically clears the ISR bit indicated by L2-L0 |
| 1 | 0 | Rotate priority so that a device after being serviced has the lowest priority |
| 1 | 1 | Rotate priority until position specified by L2-L0 is lowest |

The bits in OCW2 are only temporarily retained by the 8259A until the actions specified by them are carried out. This statement is particularly important with regard to the EOI bit. Let us now examine these possibilities in greater detail beginning with the normal priority mode (00).

Ordinarily, a request on IR0 has the highest priority, one on IR1 has the next highest priority, and so on. When the first INTA pulse arrives the priority resolver allows only the unmasked request having the highest priority to set its ISR bit. Because the 3 least significant bits of ICW2, where ICW2 indicates the interrupt type and determines the interrupt pointer address, are determined by which ISR bit is set, the address of the interrupt routine depends on which ISR bit is set. Therefore, the interrupt routine associated with the device connected to the highest-priority IR pin is begun first and the other requests must wait until further interrupts are allowed.

Under the normal priority mode, if ISR_n is set, the priority resolver will not recognize any requests on IR7 through IR(n+1), but will recognize unmasked requests on IR(n-1) through IR0. Consequently, if the IF flag in the CPU has been reset to 1, requests having higher priority than the one being processed may cause the current interrupt routine to be interrupted while those having lower priorities are kept waiting. The lower-priority requests are processed according to their priorities as the higher-priority ISR bits are cleared. If AEOI = 1, the ISR bit corresponding to an interrupt is automatically cleared at the end of the second INTA pulse. When AEOI = 0 the ISR must be cleared by the interrupt routine by setting bit 5 of OCW2.

As an example of the normal priority mode, suppose that initially AEOI = 0 and all ISR and IMR bits are clear. Also suppose that, as shown in Fig. 8-18, requests occur simultaneously on IR2 and IR4, then a request arrives at IR1, and last a request arrives at IR3, and that these are the only requests. First, ISR2 will be set and the interrupt routine associated with IR2 will start executing. After this routine resets the IF flag to 1 and when the IR1 request is made, ISR1 will be set and the IR1 routine will be executed in its entirety. While it is executing it should reset the IF bit to 1 and send the necessary command to clear ISR1. Upon the return to the IR2 routine, ISR2 is cleared. Then ISR4 is set and its routine is begun. While this routine is executing IR3 is made. It is acknowledged as soon as IF is reset to 1, and ISR3 is set. Then the IR3 routine is initiated. Before the IR3 routine is completed it should clear ISR3 and set IF. The return is made to the IR4 routine, which should clear ISR4 before returning to the IR2 routine. The IR2 routine, having already cleared the ISR2 bit, would simply return to the interrupted program. (Note that if IF is not reset to 1 within the interrupt routine, further interrupts will not be processed until the routine is completed, i.e., the IRET instruction is encountered.)

Although a 1 sent to bit 5 of OCW2 normally causes the highest-priority ISR bit (i.e., the last ISR bit to be set) to be cleared, any ISR bit can be explicitly cleared by sending an OCW2 with the R, SL, and EOI bits set to 011 and putting

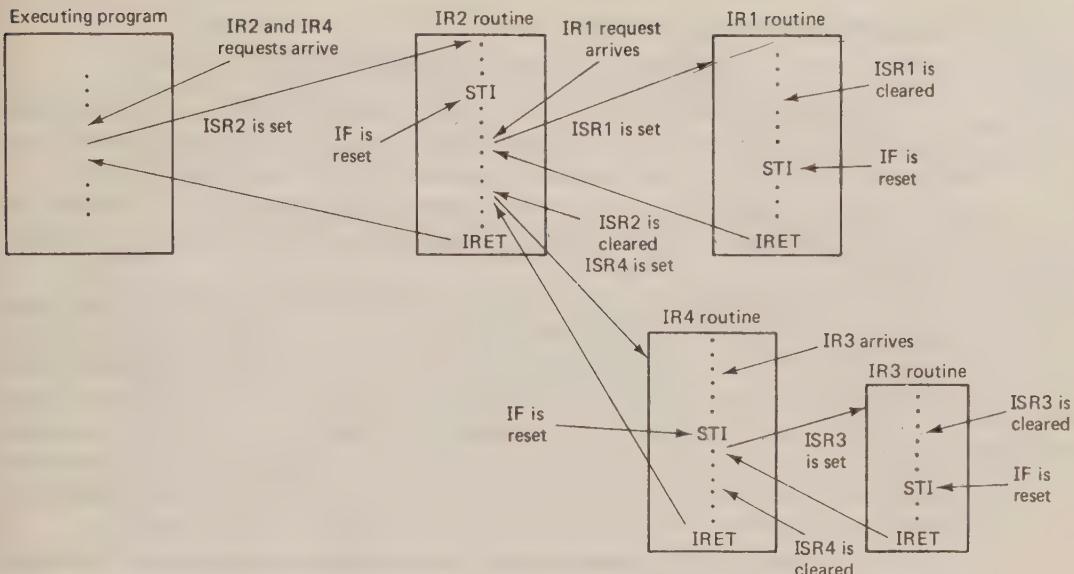


Figure 8-18 Actions taken in the normal operating mode when a typical sequence of interrupts occurs.

the number of the bit to be cleared in L2-L0. If

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

is sent to OCW2, then ISR3 will be cleared.

In addition to the normal priority mode discussed above, OCW2 can rotate the priority by assigning bottom priority to any one of the IR levels. In this case the other priorities will follow as if the normal ordering had been rotated. For instance, if the lowest priority is given to IR4, then the order of priorities will be:

IR5, IR6, IR7, IR0, IR1, IR2, IR3, IR4

(i.e., IR5 is rotated into the top-priority position). A rotation by one can be obtained by letting the combination for the R and SL bits be 10. If the R and SL bit combination is 11, then the IR level with the lowest priority is the one specified by L2-L0. If IR5 currently has top priority and

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|

is sent to OCW2, then the new priority ordering would be

IR6, IR7, IR0, IR1, IR2, IR3, IR4, IR5

but if

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

is sent, the new ordering would be

IR3, IR4, IR5, IR6, IR7, IR0, IR1, IR2

The R and SL bits may also have significance when EOI = 0. In this case R = 1 and SL = 0 cause automatic rotations when AEOI = 1, and R = SL = 0 turns off this action so that automatic rotations do not take place. R = SL = 1 and EOI = 0 result in the lowest priority being designated by L2-L0 without an EOI command being sent. The remaining combination, R = 0 and SL = 1, causes no action.

In OCW3 the ESMM (enable special mask mode) and SMM (special mask mode) bits can be used to negate the priority modes discussed above. If a byte is sent to OCW3 in which both ESMM and SMM are set to 1, then unmasked interrupt requests are processed as they arrive (provided that the processor's IF bit is 1) and the priority order is ignored. By subsequently sending a byte to OCW3 in which ESMM = 1 and SMM = 0, a switch back to the priority ordering of interrupts can be made. If a byte with the ESMM bit equal to 0 is sent to OCW3, then the SMM bit will have no effect and the special mask mode will not change.

The P (polling) bit is used to place the 8259A in polling mode. This mode assumes that the CPU is not accepting interrupts (IF = 0) and it is necessary for the interrupt requests in the IRR to be polled. When the P bit is 1 the next RD signal would cause the appropriate bit in the ISR to be set just as if INTA signal had been received, and would return to the AL register in the CPU a byte of the form

| | | | | | | | |
|---|---|---|---|---|----|----|----|
| I | - | - | - | - | W2 | W1 | W0 |
|---|---|---|---|---|----|----|----|

where I = 1 indicates that an interrupt is present and W2, W1, and W0 give the IR level of the highest-priority interrupt. For example, if P = 1, the priority ordering is

IR3, IR4, IR5, IR6, IR7, IR0, IR1, IR2

there are unmasked interrupts on IR4 and IR1, and the instruction

IN AL,80H

(where 0080 is the even address of the 8259A) is executed, then

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | - | - | - | - | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

is input to the AL register.

When P = 0, the contents of IRR or ISR can be read into the AL register by setting RR = 1 and executing the instruction

IN AL,80H

If at the time the IN instruction is executed RIS = 0, then IRR is input; otherwise, ISR is read. The contents of IMR can always be read by using the odd address of 8259A, e.g., for the address assignment indicated above,

IN AL,81H

would input the contents of IMR to AL.

Because the OCW3 bits (except for ESMM) are used to specify whether or not the 8259A is in special mask mode and which information is to be put on the data bus during a read, these bits are retained until they are reset by the next output to OCW3. For example, if P = 0, RR = 1, and RIS = 0, any read from the even address of the 8259A before a new byte is sent to OCW3 will cause IRR to be read.

As a final note regarding the internal workings of the 8259A, let us examine what happens when noise interferes with a request. An IR input must remain high until after the trailing edge of the first INTA pulse. If it does not, then the 8259A will simulate a 1 on IR7. Therefore, provided that no device is connected to IR7, requests on this line would indicate improper drops in signals on the other request lines and the interrupt routine associated with IR7 would serve as a noise "cleanup" routine. If a device is connected to IR7, this noise detection scheme could still be used because an IR7 request from a device will set ISR7, while a noise-related request on IR7 will not affect ISR7. Thus, the IR7 routine could distinguish between the two events by reading the ISR and testing bit 7.

8-3-2 Interrupt System Based on Multiple 8259As

A multiple 8259A interrupt system is diagrammed in Fig. 8-19. In this figure data bus drivers are not shown, but they could be inserted as shown in Fig. 8-9. Although the SP/EN pin on the master 8259A is connected to the data bus transceivers, a 0 is applied to the SP/EN pins of the slaves. Only one slave is shown, but up to seven more slaves could be similarly connected into the system, permitting up to 64 distinct interrupt request lines. When designing the address decoder logic, each 8259A must be given its own address pair in the I/O address space. The drivers inserted in the CAS2-CAS0 lines may or may not be needed, depending on the proximity of the master to the slaves.

In a multiple 8259A system the slaves must be initialized as well as the master. The master would be initialized in the same way as indicated above except that S_{NGL} would be set to 0 and ICW3 would need to be filled. A 1 would be put in each ICW3 bit for which the corresponding IR bit is connected to a slave and 0s would be put in the remaining bits. The SFNM bit may be set to 1 to activate the special fully nested mode. The S_{NGL} bit should also be set to 0 when initializing the slaves. Thus, an ICW3 will be required for each slave, but for a slave ICW3 has a different meaning. For a slave, ICW3 has the form

| | | | | | | | |
|---|---|---|---|---|-----|-----|-----|
| 0 | 0 | 0 | 0 | 0 | ID2 | ID1 | ID0 |
|---|---|---|---|---|-----|-----|-----|

where the 3 least significant bits provide the slave with an identification number. The identification number given to the slave should be the same as the number of the master request line to which its INT pin is connected.

When a slave puts a 1 on its INT pin this signal is sent to the appropriate IR pin on the master. Assuming that the IMR and priority resolver do not block this signal, it is sent to the CPU through the INT pin on the master. When the CPU returns the INTA signal the master will not only set the appropriate ISR bit and

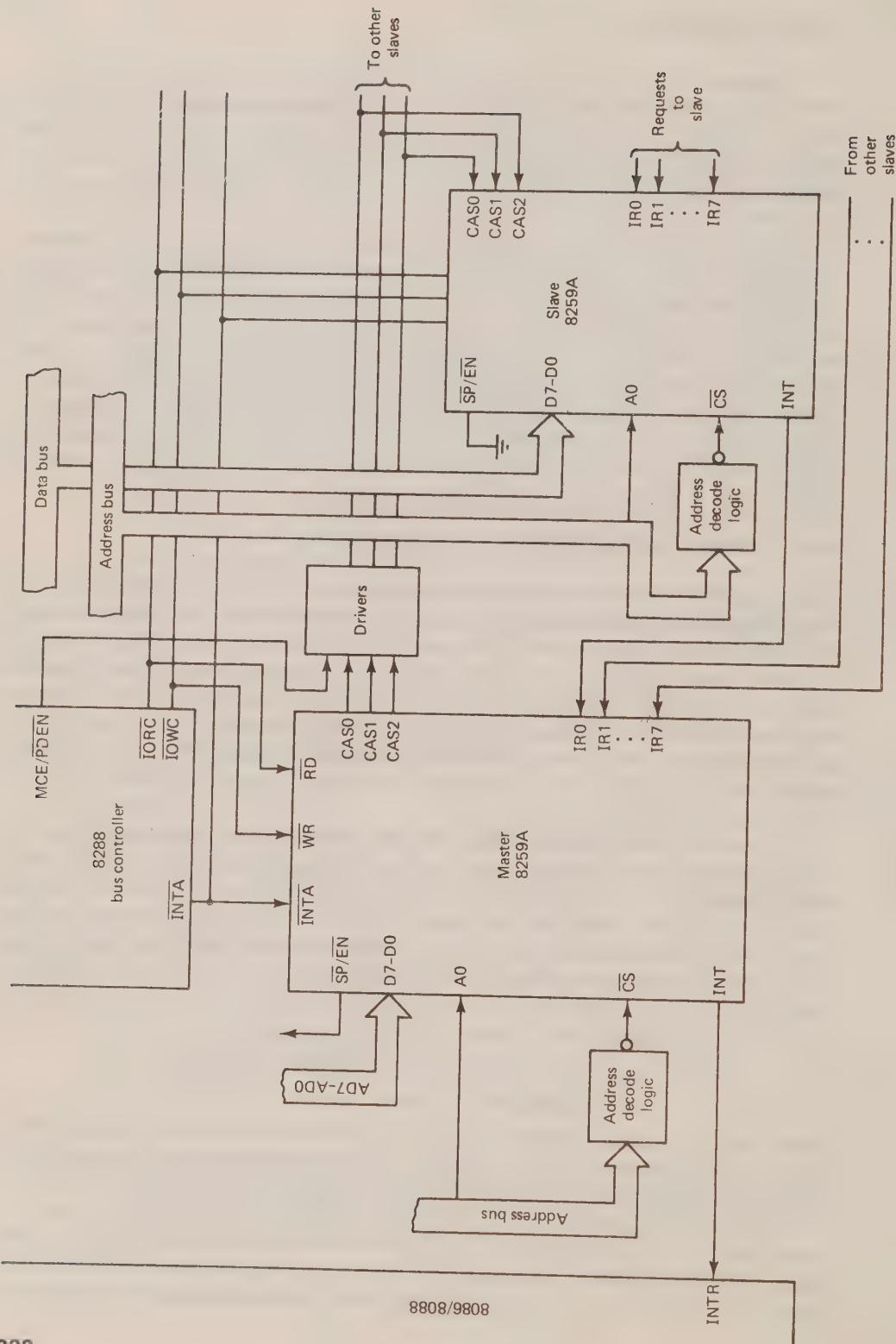
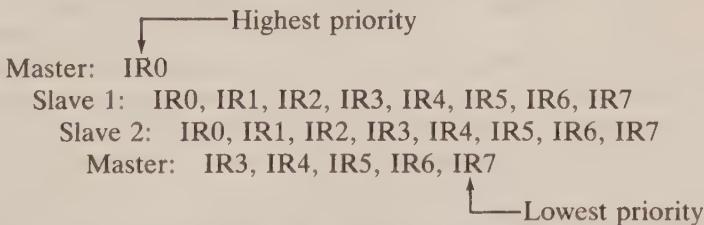


Figure 8-19 Multiple 8259A-based interrupt system.

clear the corresponding IRR bit, it will also check the corresponding bit in ICW3 to determine whether or not the interrupt came from a slave. If so, the master will place the number of the IR level on the CAS2-CAS0 lines; if not, it will put the contents of ICW2 on the data bus and no signals will be applied to the CAS2-CAS0 lines. The INTA signal is also received by all of the slaves, but only that slave whose ID matches the number sent to it by the master over the CAS2-CAS0 lines will accept the signal. In the selected slave the appropriate ISR bit will be set, the corresponding IRR bit will be cleared, and its ICW2 will be put on the data bus. Because ICW2 contains the interrupt type, it is important that unique combinations be put in the master and slave ICW2s during the initialization process. EOI commands are required for both the master and the slave if their AEOI bits are 0.

Except for the response to the INTA signal discussed above, the actions taken by all 8259A devices in the system are the same. Also, their modes are controlled and their registers are read in the same way. There is one exception, however. If the SFNM bit in the ICW4 of the master is initialized to 1, the master will enter the special fully nested mode. This mode is ordinarily used with the normal priority mode and AEOI = 0. In this case, the master will allow unmasked requests of sufficient priority to be passed on to the INT pin even if the corresponding ISR bit is already 1. This means that if a higher-priority request arrives at a slave while one or more of the slave's requests is being processed, the new request will be allowed to send its INT signal through the master. When using the special fully nested mode, two EOI commands may need to be sent by the interrupt routine. First, a nonspecific EOI command would be sent to the slave that caused the interrupt and then the slave's ISR would be tested. If and only if the ISR contains all 0s would a nonspecific EOI command be sent to the master. Therefore, assuming that slave 1 is connected to IR1 and slave 2 is connected to IR2 of a master, they are the only slaves, and the highest priority in all three 8259As is assigned to IR0, the fully nested order of priorities would be:



The masks in the master and slaves may, of course, be used to block out some of the requests.

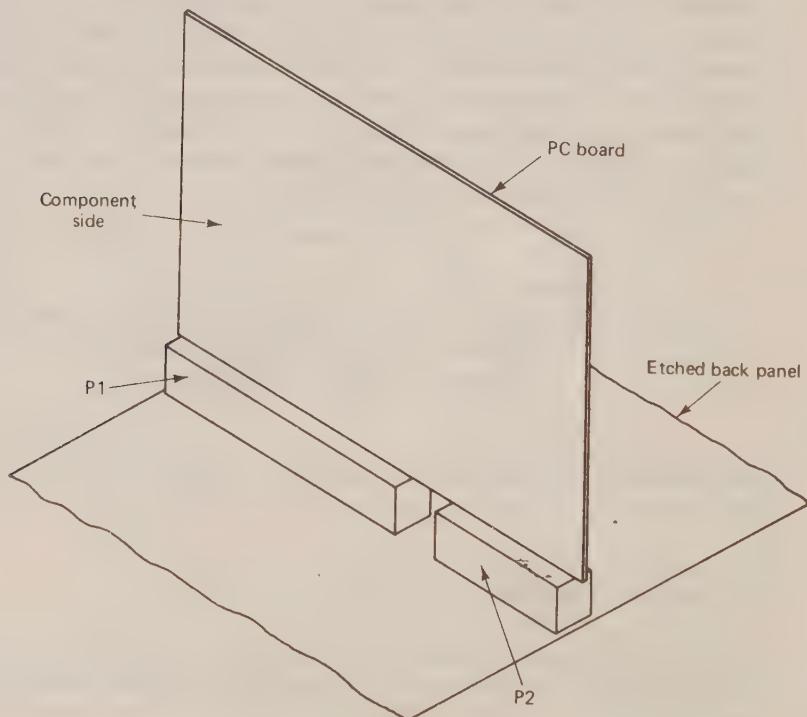
8-4 BUS STANDARDS

For reasons of simplicity, flexibility, and low cost, most microcomputers, including those involving multiprocessor configurations, are built around a primary system bus which connects all of the major components in the system. In order to obtain

a foundation while designing its products, a microcomputer manufacturer makes assumptions about the bus that is to be used to connect its devices together. Frequently, these assumptions become formalized and constitute what is referred to as a *bus standard*. If a line of products becomes popular, then its bus standard is widely accepted and other manufacturers may also design devices that satisfy the standard, particularly if the standard is adopted by an organization such as the Institute of Electrical and Electronics Engineers (IEEE). Because of the availability of a large number of compatible devices, a user can sometimes significantly reduce the system development time and cost by choosing a popular and well-designed bus. In this section we will briefly consider the Intel MULTIBUS.

The Intel MULTIBUS has gained wide industrial acceptance and several manufacturers offer MULTIBUS-compatible modules. This bus is designed to support both 8-bit and 16-bit devices and can be used in multiprocessor systems in which several processors can be masters. At any point in time, only two devices may communicate with each other over the bus, one being the master and the other the slave. The master/slave relationship is dynamic with bus allocation being accomplished through the bus allocation (i.e., request/grant) control signals. The MULTIBUS has been physically implemented on an etched backplane board which is connected to each module using two edge connectors, denoted P1 and P2, as shown in Fig. 8-20. The connector P1 consists of 86 pins which provide the major

Figure 8-20 Illustration of a module being plugged into a MULTIBUS.



bus signals, and P2 is an optional connector consisting of 60 auxiliary lines, primarily used for power failure detection and handling. (Although P1 and P2 are touched on below, they are not fully described and the reader should go to Reference 3 for the details.)

The P1 lines can be divided into the following groups according to their functions:

1. Address lines.
2. Data lines.
3. Command and handshaking lines.
4. Bus access control lines.
5. Utility lines.

The MULTIBUS has 20 address lines, labeled ADR0 through ADR13, where the numeric suffix represents the address bit in hexadecimal. The address lines are driven by the bus master to specify the memory location or I/O port being accessed. The MULTIBUS standard allows a 16-bit master or an 8088 to use all 20 lines to access memory and only the lower 12 address lines (ADRB-ADR0) to select an I/O port, despite the fact that both the 8086 and 8088 permit 16-bit I/O addresses. For 8-bit masters, memory addressing and I/O port selection use only 16 lines (ADRF-ADR0) and 8 lines (ADR7-ADR0), respectively.

In addition to the address lines, the bus provides three address control signals. They are the byte high enable (BHEN), inhibit RAM (INH1), and inhibit ROM (INH2) signals. BHEN is used by 16-bit devices to indicate that data are to be transferred on the high byte of the data bus; this is required for word transfers. The MULTIBUS standard calls for all single bytes to be communicated over only the lower 8 bits of the bus; therefore, any 16-bit interface must include a swap byte buffer so that only the lower data lines are used for all byte transfers. (It should be pointed out that because an 8086 expects a byte to be put on the high-order byte of the bus when BHE is active, one may want to permit nonstandard MULTIBUS transfers between memory and an 8086.)

The two inhibit signals are provided for overlaying RAM, ROM, and auxiliary ROM in a common address space. For example, a bootstrap loader may be stored in an auxiliary ROM and a monitor in a ROM. Because the loader is needed only after a reset, INH1 and INH2 could both be activated while the loader is executing. Then, when the monitor is in control, INH2 could be raised while INH1 remains low. If control is passed to the user, INH1 and INH2 could both be deactivated, thus allowing the RAM to fill the entire memory space during normal operation.

There are 16 bidirectional data lines (DATF-DAT0), only eight of which are used in an 8-bit system. Data transfers on the MULTIBUS bus are accomplished by handshaking signals in a manner similar to that described in the preceding sections. The memory read (MRDC), memory write (MWTC), I/O read (IORC), and I/O write (IOWC) lines are defined to be the same as they were in the discussion of the 8288 bus controller. There is an acknowledge (XACK) signal which serves the same purpose as the READY signal in the discussion of the bus control logic,

i.e., to verify the end of a transfer. In a general setting it may be received by any bus master. Because a master must wait to be notified of the completion of a transfer, the duration of a bus cycle varies depending on the speed of the bus master and the slave. This asynchronous nature enables the system to handle slow devices without penalizing fast devices.

When an 8086 or 8088 is configured in its maximum mode, the 8288 generates bus commands that are fully compatible with the electrical requirements of the MULTIBUS. An 8086 or 8088 in minimum mode can also communicate with a MULTIBUS, provided that external logic is included to generate the necessary command signals. An 8086 or 8088 system with more than one CPU must be configured in maximum mode. Also, in order to obtain the inverted signals specified by the MULTIBUS standard, 8283 latches and 8287 transceivers are used instead of 8282s and 8286s. The command and handshaking signal lines also include 8 interrupt request lines (INT7–INT0) and one interrupt acknowledge line (INTA) to support priority management.

In order to support multiprocessor configurations, there are 6 lines designed for use by bus access logic. These bus access lines are typically connected to bus arbiters which ensure that only one bus master can have control of the MULTIBUS at a time. The bus arbiter is discussed in Chap. 11. The utility group consists of lines for ground, +5 V, -5 V, +12 V, -12 V, and timing.

The auxiliary bus which is connected to PC boards through the 60-pin P2 edge connector is optional. It is needed primarily to provide the system with the capability to detect and to handle power failures (although it also includes some seldom used control lines). When the ac voltage level drops below a certain level, the power supply generates a power failure signal. This signal causes the user-supplied power-failure-handling logic to generate an interrupt to the bus master, which then saves the current program's vital information. After the ac power is restored, the power-failure-handling logic signals the bus master to initiate a power-up sequence. During this sequence, the processor restores the information and resumes the execution of the suspended program. With a standby power source, normally provided by batteries, memory and other modules can remain active during power failures for reasonably long periods of time.

BIBLIOGRAPHY

1. 1982 *Data Component Catalog* (Santa Clara, Calif.: Intel Corporation, 1982).
2. *The 8086 Family User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
3. *Intel MULTIBUS Interfacing*, Application Note AP-28A (Santa Clara, Calif.: Intel Corporation, 1979).

EXERCISES

1. Use the logic diagram of the 8286 in Fig. 8-6 and truth tables to verify the statements regarding the data flow direction and enabling and disabling of the data flow.

2. Assume that the instruction

```
XCHG BL,DATA
```

is already in the queue and is ready to execute, and give a timing diagram of the bus activity during its execution [similar to the ones in Figs. 8-12(a) and 8-15] for each of the following cases:

- (a) Minimum mode 8086 with no wait states.
- (b) Minimum mode 8088 with no wait states.
- (c) Maximum mode 8086 with no wait states.
- (d) Maximum mode 8086 if a ready signal two clock periods long is received in the middle of T₃. (Also include the timing diagram of the ready signal.)
- (e) Maximum mode 8086 with no wait states, but assume that a bus request is detected during the input transfer and is dropped after only one bus cycle.

3. Consider a maximum mode 8086 system that is executing the instructions

```
MOV AX,DATA (DATA has an even address)
CMP AX,1
```

Suppose that an interrupt request arrives while the first instruction is executing and draw a timing diagram of the bus activity from the beginning of the first instruction until the interrupt type is received by the processor.

4. Assuming that the instruction is already in the queue, give the number of bus cycles needed to execute each of the following instructions:

- (a) PUSH AX
- (b) CALL NEAR PTR PROC_A
- (c) CALL FAR PTR PROC_B
- (d) MOV DATA,AX (DATA has an even address)
- (e) MOV DATA,AX (DATA has an odd address)
- (f) OUT 60H,AX

- 5. What is the minimum number of bus cycles that can occur between the time an interrupt request is recognized and the first instruction in the interrupt routine is fetched?
- 6. Give an 8086 timing diagram that shows the activity on the clock line and all of the address/data lines while the instruction

```
ADD AX,TAX
```

is being executed. First assume that TAX begins at an even address and then assume that it begins at an odd address.

7. Given that the instruction

```
JMP NEAR PTR BEGIN
```

is already in the instruction queue and the branch is made to

```
BEGIN: CMP AL,0
```

describe the actions taken during each bus cycle from the beginning of the execution of the JMP instruction until the end of the execution of the CMP instruction. Answer the exercise for each of the following cases:

- (a) An 8088 system.
- (b) An 8086 system with BEGIN at an even address.
- (c) An 8086 system with BEGIN at an odd address.

8. Write an initialization sequence for an 8259A, which is the only 8259A in an 8086 system and has an even address of 2130, that will cause:
 1. Requests to be level triggered.
 2. The interrupt type for an IR0 request to be 28.
 3. $\overline{SP/EN}$ to output a disable signal to the data bus transceivers.
 4. The ISR bits to be automatically cleared at the end of the second \overline{INTA} pulse.
 5. The IMR to be cleared.
9. Write an instruction sequence that will cause the priority of an 8259A, whose even address is 08A0, to be

IR4, IR5, IR6, IR7, IR0, IR1, IR2, IR3

Solve this exercise twice, once assuming that the highest priority is currently IR0 and once assuming that it is IR3.

10. Write a sequence of instructions that will transfer the contents of the IRR, ISR, and IMR in an 8259A, whose even address is 0500, to an array in memory that begins at REG_ARR.
11. Give an instruction sequence that will cause the requests on IR3, IR4, and IR6 of an 8259A, whose even address is 1208, to be blocked.
12. Suppose that IF = 0. Give a sequence of instructions that will poll the 8259A, whose even address is 1000, and branch to the n th offset in an array of eight offsets, where n is the number of the highest-priority interrupt request line that has received a request. If there are no requests, execution is to continue in sequence.
13. Given the following sequence of events, normal priority, and that EOI commands must be output, draw a diagram similar to the one in Fig. 8-18:
 1. Request on IR3.
 2. Request on IR2.
 3. Request on IR6.
 4. IF reset to 1.
 5. IF reset to 1.
 6. EOI to clear ISR2.
 7. EOI to clear ISR3.
 8. IF reset to 1.
 9. EOI to clear ISR6.
14. Consider a maximum mode 8086 system containing a master 8259A and one slave 8259A that is connected to IR1 of the master. The even addresses of the master and slave are 0580 and 0582, respectively. The interrupt type for IR0 of the master is to be 30 and for IR0 of the slave is to be 38, and all requests are to be edge triggered. EOI commands are to be used to clear the ISR bits. For both the master and slave the IMRs are to be cleared and $\overline{SP/EN}$ is used as an input. Give a program sequence for initializing the interrupt system.
15. For the system described in Exercise 14, write a program sequence for clearing the appropriate ISR bits after an interrupt request is made by the slave. Remember that the slave must be checked for all 0s before ISR1 in the master is cleared.
16. Consider an interrupt system containing a master and three slaves, with the slaves being connected to IR2, IR3, and IR6 of the master. Assume that the IMR in the master is set to

01010000

and that the IMRs in the slaves are cleared. Normal priority is used in all 8259As except the slave connected to IR3, and its highest-priority request line is IR5. List the unmasked request lines in the order of their priorities, with the highest priority listed first. Repeat the solution for the case when the master's highest-priority line is IR3.

9

I/O Interfaces

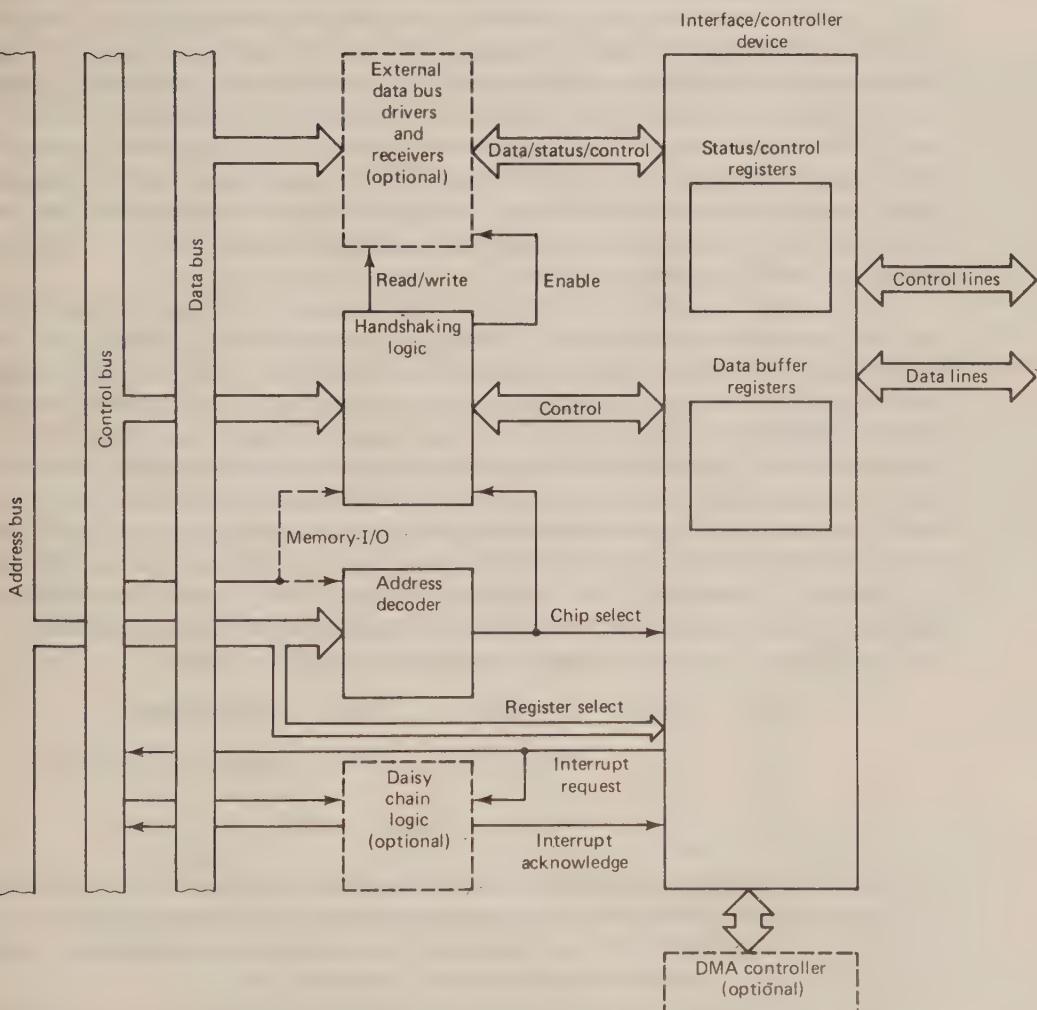
The I/O and memory interfaces are the counterparts to the bus control logic. What goes between the bus control logic and the interfaces is simply the conductors in the bus; therefore, the interfaces must be designed to accept and send signals that are compatible with the bus control logic and its timing. Although there are similarities in memory and I/O interfaces, there are also significant differences. This chapter will concentrate on I/O interfaces and Chap. 10 will study memories and their interfaces.

An I/O interface must be able to:

1. Interpret the address and memory-I/O select signals to determine whether or not it is being referenced and, if so, determine which of its registers is being accessed.
2. Determine whether an input or output is being conducted and accept output data or control information from the bus or place input data or status information on the bus.
3. Input data from or output data to the associated I/O device and convert the data from parallel to the format acceptable to the I/O device, or vice versa.
4. Send a ready signal when data have been accepted from or placed on the data bus, thus informing the processor that a transfer has been completed.
5. Send interrupt requests and, if there is no interrupt priority management in the bus control logic, receive interrupt acknowledgments and send an interrupt type.
6. Receive a reset signal and reinitialize itself and, perhaps, its associated device.

Figure 9-1 contains a block diagram of a typical I/O interface. The function of an I/O interface is essentially to translate the signals between the system bus and the I/O device and provide the buffers needed to satisfy the two sets of timing constraints. Most of the work done by the interface is accomplished by the large block on the right side of the figure. Most often this block is implemented by a single IC device, but its functions could be scattered across several devices. Clearly, its functions vary radically, depending on the I/O device with which it is designed to communicate.

Figure 9-1 Typical block diagram of an I/O interface.



An interface can be divided into two parts, a part that interfaces to the I/O device and a part that interfaces to the system bus. Although little can be said about the I/O device side of the interface without knowing a lot about the device, the bus sides of all interfaces in a given system are very similar because they connect to the same bus. To support the main interface logic there must be data bus drivers and receivers, logic for translating the interface control signals into the proper handshaking signals, and logic for decoding the addresses that appear on the bus. In an 8086/8088 system, 8286 transceivers could drive the data bus, just as they are used to drive the bus at its bus control logic end. However, the main interface devices may have built-in drivers and receivers that are sufficient for small, single-board systems.

The handshaking logic cannot be designed until the control signals needed by the main interface device are known, and these signals may vary from one interface to the next. Typically, this logic must accept read/write signals for determining the data direction and output the \overline{OE} and T signals required by the 8286s. In a maximum mode 8086/8088 system it would receive the \overline{IOWC} (or \overline{AIOWC}) and \overline{IORC} signals from the 8288 bus controller and in a minimum mode system it would receive the \overline{RD} , \overline{WR} , and $\overline{M/IO}$ (or $\overline{IO/M}$) signals. The interrupt request, ready, and reset lines would also pass through this logic. In some cases, the bus control lines may pass through the handshaking logic unaltered (i.e., be connected directly to the main interface device).

The address decoder must receive the address and, perhaps, a bit indicating whether the address is in the I/O address space or the memory address space. In a minimum mode system this bit could be taken from the M/IO (or IO/M) line, but in a maximum mode system the memory-I/O determination is obtained directly from the \overline{IOWC} and \overline{IORC} lines. If the decoder determines that its interface is being referenced, then the decoder must send signals to the main device indicating that it has been selected and which register is being accessed. The bits designating the register may be the low-order address bits, but are often generated inside the interface device from the read/write control signals as well as the address signals. For example, if there are two registers A and B that can be read from and two registers C and D that can be written into, then the read and write signals and bit 0 of the address bus could be used to specify the register as follows:

| Write | Read | Address Bit 0 | Register Being Accessed |
|-------|------|---------------|-------------------------|
| 0 | 1 | 0 | A |
| 0 | 1 | 1 | B |
| 1 | 0 | 0 | C |
| 1 | 0 | 1 | D |

If a daisy chain is included in the system instead of an interrupt priority management device, then each interface must contain daisy chain logic, such as that shown in Fig. 6-14b, and must include logic to generate the interrupt type. Also, the interface may be associated with a DMA controller.

Many interfaces are designed to detect at least two kinds of errors. Because the lines connecting an interface to its device are almost always subject to noise,

parity bits are normally appended to the information bytes as they are transmitted. If even parity is used the parity bit is set so that the total number of 1s, including the parity bit, is even. For odd parity the total number of 1s is odd. As these bytes are received the parity is checked and, if it is in error, a certain status bit is set in a status register. Some interfaces are also designed to check error detection redundancy bytes that are placed after blocks of data. The other type of error most interfaces can detect is known as an *overrun error*. As we have seen, when a computer inputs data it brings the data in from a data-in buffer register. If, for some reason, the contents of this register are replaced by new data before they are input by the computer, an overrun error occurs. Such an error also happens when data are put in a data-out buffer before the current contents of the register have been output. As with parity errors, overrun errors cause a certain status bit to be set.

Interfaces can be categorized according to the way in which they communicate with their I/O devices. The first section in this chapter concerns interfaces for those I/O devices which send and receive their information serially, and includes a discussion of long-distance communication. Section 9-2 considers general interfaces that transfer data to and from their devices in a parallel format. The third section considers interfaces that can provide programmable timing of either external or internal events or can count external events, and the fourth section discusses designing an interface into a unit with a keyboard and display. The next two sections examine high-speed communication by studying DMA and diskette controllers.

This chapter contains several examples that are centered on Intel interface devices. These devices have only eight data pins and, because the 8088 has only an 8-bit data bus, it is less confusing to assume that the microprocessor in these examples is an 8088. They can be connected to a 16-bit 8086 bus, but, because an 8086 expects odd-addressed bytes to be transferred over the high-order 8 bits of the bus, certain modifications and/or additions to the interface designs are needed. Therefore, all of the designs in all but the last section assume an 8-bit 8088 bus, and the last section in the chapter discusses the changes needed to connect these devices to a 16-bit 8086 bus. Also, the Intel examples in all but the last section assume a minimum mode system, and the last section considers the modifications required for maximum mode operation.

9-1 SERIAL COMMUNICATION INTERFACES

Many I/O devices transfer information to or from a computer serially, i.e., one bit at a time over a single conductor pair or communication channel, with each bit occupying an interval of time having a specified length. There are several serial interface devices and typically they are constructed as shown in Fig. 9-2. The status register would contain error and other information concerning the state of the current transmission, and the control register is for holding the information that determines the operating mode of the interface. The data-in buffer is paired with a serial input/parallel output shift register. During an input operation, the bits are

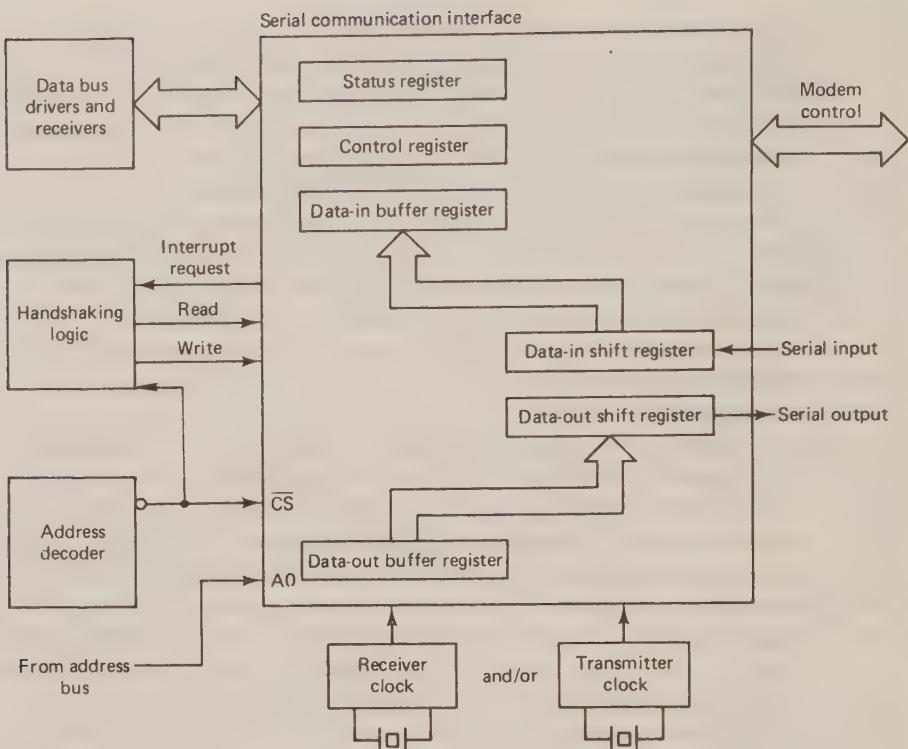


Figure 9-2 Serial interface.

brought into the shift register one at a time and, after a character has been received, the information is transferred to the data-in buffer register, where it waits to be taken by the CPU. (Although a single datum is not necessarily an alphanumerically coded character, it normally is and, therefore, it will be referred to as a character.) Similarly, the data-out buffer is associated with a parallel input/serial output shift register. An output is performed by sending data to the data-out buffer, transferring it to the shift register, and then shifting it onto the serial output line.

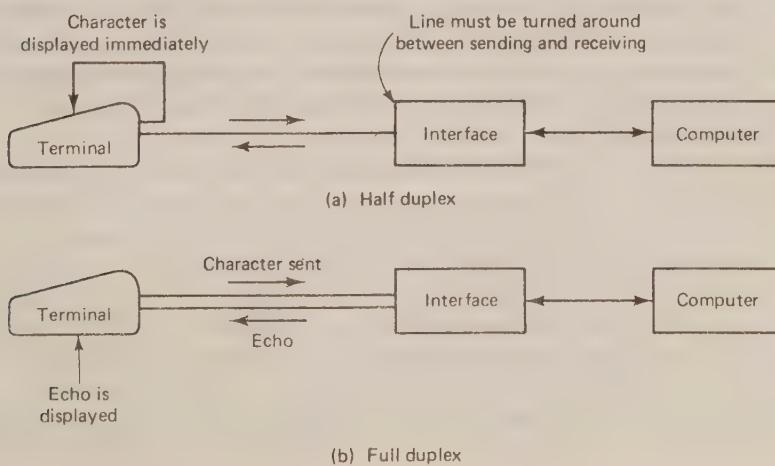
Although there are several ways in which the four port registers could be addressed, it has been assumed that the status register can only be read from and the control register can only be written into. Therefore, an active signal on the read line would indicate either the status or data-in buffer register and A0 could be used to distinguish between these two possibilities. Likewise, the write and A0 signals can be used to designate one of the other two registers.

The interface shown in Fig. 9-2 has separate lines for sending and receiving information. When different lines are used for the two signal directions the communication system is said to be *full duplex*. Such a system can transmit and receive at the same time. Terminals and other serial devices that are connected to a computer through a full duplex system normally send each character to the com-

puter without displaying it. The computer immediately sends the character back, or *echoes* the character, and the terminal then displays it. There is no loss of time because the echoed characters can be sent back at the same time new characters are being input by the computer. The advantage of this scheme is that the user gets visual verification not only that he has struck the correct key, but also that the correct character was received by the computer. The alternative, called *half duplex*, is to use the same line for both inputting and outputting. In this case a character typed into a terminal is displayed at the same time it is sent to the computer, and echoing is not used. If the computer has been receiving characters and then wishes to send characters, or vice versa, the communication link must be *turned around*, a process that requires time. Although full duplex avoids turnarounds and provides echoing, it does require an extra line. The two modes of communication are illustrated in Fig. 9-3. Many terminals and interfaces provide both turnaround capability for half duplex transmissions and separate pins for full duplex transmissions. For a device such as a printer that requires only one-way communication, clearly half duplex would be sufficient and turnarounds would not be needed.

There are two basic types of serial communication. There is *asynchronous serial communication* in which special bit patterns separate the characters, and *synchronous serial communication* which allows characters to be sent back to back but must include special "sync" characters at the beginning of each message and special "idle" characters in the data stream to fill up time when no information is being sent. An asynchronous transmission may include dead time of arbitrary lengths between the characters, while in a synchronous transmission the characters must be precisely spaced even though some of the characters may contain no information. Although both types may waste time sending useless bits, the maximum information rate of a synchronous line is higher than that of an asynchronous line with the same bit rate because the asynchronous transmission must include

Figure 9-3 Basic transmission modes.



extra bits with each character. On the other hand, the clocks at the opposing ends of an asynchronous transmission line do not need to have exactly the same frequency (as long as they are within permissible limits) because the special patterns allow for resynchronization at the beginning of each character. For a synchronous transmission the activity must be coordinated by a single clock since it is the clock that determines the position of each bit. This means that the clock timing must be transmitted as well as the data.

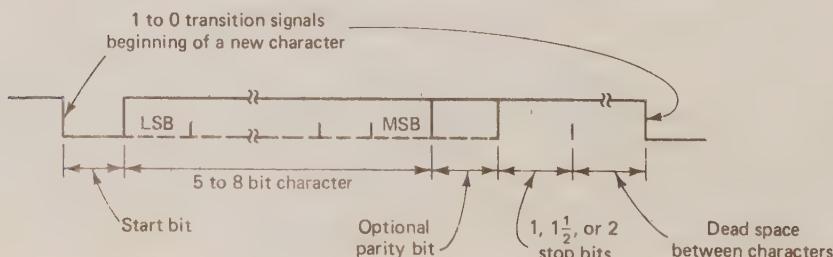
9-1-1 Asynchronous Communication

The format of an asynchronous character is shown in Fig. 9-4. A character contains several extra bits as well as the coded information. Before a character begins the line must be in its "1" state, often called its *mark* state. The transition from the mark state to the "0", or *space*, state marks the beginning of the character. The first bit is always 0 and is called the *start bit*. It is followed by from 5 to 8 information bits, the first of which is the LSB of the character. The information bits may or may not be followed by a parity bit and, if there is one, it may represent either even or odd parity. The last bits are 1s and are called *stop bits*. There may be 1, $1\frac{1}{2}$, or 2 stop bits.

Although the number of information bits, whether or not there is a parity bit, the type of parity, and the number of stop bits may be changed from one transmission (i.e., sequence of characters) to the next, these parameters are constant within a single transmission. On some interfaces these parameters can be programmed using the control registers; on others they are determined by switch settings within the interfaces. If these parameters are programmable, the interface would include a control register such as the one shown in Fig. 9-5. In this example bits 0 and 1 indicate the number of stop bits, bits 2 and 3 indicate the number of information bits, and bit 4 specifies whether or not parity bits are present. If bit 4 is 1, then bit 5 determines the type of parity.

At the transmitter end a clock is needed to determine the period of each bit by regulating the time between the shifts made by shift registers. After shifting out all of the bits the transmitter would continually output a mark state until another character is ready to be sent. At the receiver end a clock is also required for measuring the time between shifts so that the incoming signal is sampled at the correct times. Normally, the frequencies of these clocks are 16, 32, or 64 times the

Figure 9-4 Format of a standard asynchronous transmission.



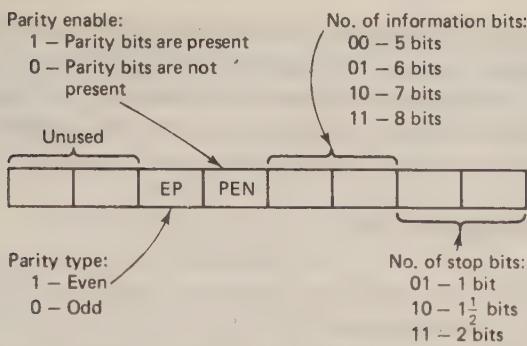


Figure 9-5 Typical control register for specifying the character format.

bit frequency. Assuming a factor of 16, after detecting the 1-to-0 transition at the beginning of a character the receiver would count eight clock pulses and sample the input. If it sees a 0, it assumes the transition did, in fact, initiate a start bit and was not due to noise. The receiver then samples the input at intervals of 16 clock pulses until all of the bits, including the stop bits, have been input, at which time it ceases its sampling until another 1-to-0 transition is detected. It is important to note that the start, stop, and parity bits are not sent to nor received from the CPU. During output, the transmitter inserts these bits in each character and, during input, the receiver deletes these bits from the received data.

This format allows the transmitter to insert intervals of indeterminate length between characters and the receiver to resynchronize itself at the beginning of each character. Without this resynchronization mechanism, the two clocks would eventually drift out of sync and the receiver would sample improperly. With resynchronization the receiver needs to assimilate the bit rate for only one character at a time. Incorrect sampling can occur only if the two clock frequencies are so different that a receiver shift is performed at the wrong time within the first few bits after a start bit. If this happens, the probability that at least one 0 will be detected at a time when a stop bit is supposed to be received becomes high. When a 0 is seen instead of a stop bit a *framing error* is said to occur. Therefore, most asynchronous serial interfaces are designed to detect three kinds of errors: parity errors, overrun errors, and framing errors.

The transmit and receive clocks that regulate the timing in the interface do not have to be the same and need not have the same frequency, but there are obvious advantages to using one clock to provide all of the necessary interface timing pulses. If the transmit and receive bit rates are different, then the electronics at the other end of the communication link must be able to handle two bit rates. The receiving rate for these electronics must match the interface transmit rate, and vice versa.

Devices that are capable of receiving and transmitting data in the format given in Fig. 9-4, converting them to and from parallel form, and detecting parity, overrun, and framing errors are called *Universal Asynchronous Receivers and Transmitters (UARTs)*. Many manufacturers have produced UARTs and their design

has been standardized—see McNamara [1]. The microprocessor manufacturers have tended to expand their designs to complete and versatile interfaces. In some cases the UART is only a small part of the interface, which may satisfy synchronous serial and parallel communication applications as well as asynchronous serial applications.

9-1-2 Synchronous Communications

A synchronously transmitted character also consists of from 5 to 8 bits optionally followed by a parity bit, but contains no start or stop bits. All characters have the same number of bits, say n , and the transmission time is divided into groups of n bits each. The same clock controls the sampling by the receiver as is used to generate the bits, thus guaranteeing the synchronization of the two processes. The transmitter must transmit a character during each n -bit interval. If a character is not available at the beginning of an interval, an *underrun* is said to occur and the transmitter will insert an idle character. Depending on the system, idle characters may be interpreted as errors, called *underrun errors*, by the receiver.

There is no problem in starting up or turning around an asynchronous system because the transmitter can simply put a mark on the line until the transmission is ready to begin. If noise causes an accidental 1-to-0 indication, the false signal is discovered when the first sample does not detect a start bit and the system will wait for another 1-to-0 transition. In a synchronous transmission the problem of starting up after a shutdown or other disruption is more difficult. All transmissions must begin with a series of *sync characters*, which should be selected so that confusion with the other characters is unlikely. Normally, the sync characters are the same as the idle characters, and in the ASCII code the sync character is 0010110.

The receiver, which must know the identity of the sync character, will examine each bit as it arrives, and when a sequence of bits exactly matches the bits in the sync character it will assume a transmission has begun. It may then treat the succeeding characters as information or may attempt also to match one or more of them with the sync character. Because noise may cause the false identification of a sync character most systems require that a transmission begin with a sequence of sync characters. The receiver will then not assume the beginning of a transmission until the proper number of sync characters have been input. It can be the responsibility of either the receiver or an input program to remove the unnecessary sync and idle characters.

As with the number of information bits and the existence and kind of parity, the number and identity of sync characters may be programmable. If this is so, then there would be special control registers for storing the sync character and the other formatting information.

9-1-3 Physical Communication Standards

As computer use grew, so did the need for equipment that would permit human-computer and computer-computer communication over both long and short dis-

tances. This meant the involvement of several parties in certain aspects of the computer industry. So that people in the telephone, communication equipment, and computer industries could converse with each other and build equipment that would function together, several definitions and standards have been established. These definitions and standards primarily affect three areas:

1. Transmission rates.
2. Electrical characteristics.
3. Line definitions and notations.

Transmission rates are measured in *bits per second (bps)* and in *baud*, which means the number of discrete conditions being transmitted per second. If there can be only one of two possible conditions at any point in time, then bps and baud are the same. This is almost always the case at the computer interface because it can normally input or output only 1s or 0s; therefore, this book will treat the two terms as being synonymous. However, it should be noted that many communication links may allow one of four or more conditions at a given time. For example, a phase-modulated signal may take on one of four phases, with each possible phase representing 2 bits, in which case the bps rate would be twice that of the baud rate.

The standard baud rates that are most often used are 110, 300, 600, 1200, 1800, 2400, 4800, 9600, and 19,200. Most CRT terminals are capable of handling any one of these rates up to 9600 baud, while printing terminals are limited by the speed of the print mechanism. Computer-driven typewriters operate at 110 baud and some dot-matrix printers can receive at up to 2400 baud. Most interfaces are such that the transmit and receive baud rates can be set separately, and frequently these rates are programmable.

As an example, consider an asynchronous transmission in which each character contains a start bit, 7 information bits, a parity bit, and 1 stop bit. If the baud rate of the line were 1200, then the maximum number of characters per second that could be transmitted would be $1200/10 = 120$. The maximum rate could be attained only when there is no dead time between characters. As a comparison, a synchronous line operating at 1200 baud and with no parity could transmit four sync characters and a 100-character message in

$$7(100 + 4)/1200 = 0.6067 \text{ second}$$

This would mean that up to $100/0.6067 = 165$ useful characters per second could be transmitted.

There are two organizations that are primarily responsible for establishing standards for electrical characteristics and line definitions. There is the Electrical Industry Association (EIA), which has set most of the standards used in the United States, and the Comité Consultatif International Téléphonique et Télégraphique (CCITT), which is part of the United Nations and is concerned with international standards. The standard that is of the most interest to us is the EIA RS-232-C

standard, which is essentially the same as the CCITT V.24 standard. It is the one that is concerned with sending serial bit streams between interfaces or terminals and communication equipment.

How far a bit stream output by an interface or terminal can be sent before it begins to seriously deteriorate depends on the baud rate and the electrical characteristics of the transmission line. A typical curve showing maximum distance versus baud rate for an unbalanced shielded pair with a 50-pF/ft capacitance is given in Fig. 9-6. It is evident that the maximum distance drops off rapidly for

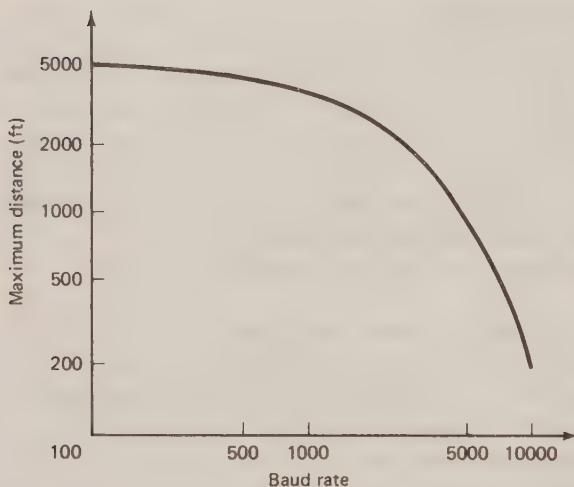
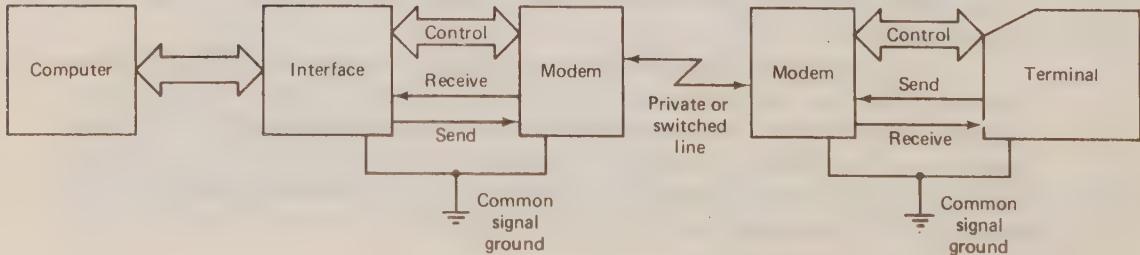


Figure 9-6 Maximum signal distance versus rate for EIA RS-232 voltage levels.

baud rates above 1000, with the maximum distance at 9600 baud being about 250 ft. For distances lying below the curve communication equipment is not required and a full duplex connection can consist of only three wires: a send line, a receive line, and a common signal ground.

For distances above the curve the transmission should be aided by inserting communication equipment as shown in Fig. 9-7. The usual configuration calls for

Figure 9-7 Long-distance serial communication.



identical circuits to be placed at the ends of the communication link for the purposes of modulating the bits going to the communication link and demodulating the bits coming from the link. These modulator/demodulator circuits are called *modems* or *data sets*. Ordinarily, the communication link is a telephone line, which may be either a direct (or private) line or a switched line.

An equivalent circuit of one of the lines between an interface or terminal and a modem is given in Fig. 9-8. The RS-232-C electrical standard is stated in terms

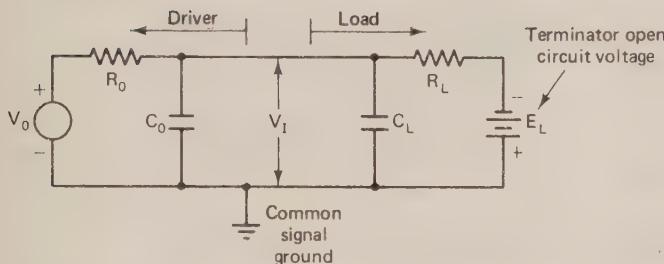


Figure 9-8 Equivalent circuit of a line between an interface or terminal and a modem.

of this circuit and is given in Fig. 9-9. If the interface or terminal is sending information to the modem, it is considered the driver and the modem is the load, and if it is receiving information, it is the load and the modem is the driver. In any case, the various components in the equivalent circuit must meet the specifications stated in Fig. 9-9. Note that the data signals are inverted in that a 1 is represented by the lower voltage. Whether or not the driver circuit for outputting data is part of the main interface device or is a separate circuit depends on the interface and the length of the line to be driven. Figure 9-10 shows a circuit for converting signals between TTL and the RS-232-C standard.

```

 $V_0 < 25 \text{ V}$ 
Maximum short circuit current to any wire in cable = 0.5 A
MARK signal at load < -3 V
SPACE signal at load > +3 V
MARK signal out of driver < -5 V and > -15 V
SPACE signal out of driver > +5 V and < +15 V
 $R_L < 7000 \text{ ohms}$  when measured with a voltage from 3 to 25 V,
but > 3000 ohms
 $C_L$  including line capacitance < 2500 pF
When  $E_L = 0$ ,  $5 \text{ V} < V_I < 15 \text{ V}$ 
 $R_0 > 300 \text{ ohms}$  under power off conditions
 $C_0$  is such that the slew rate of the driver's output
voltage is < 30 V/microsecond, but the
transition between -3 V and +3 V does not exceed
the smaller of 1 ms or 4% of the bit time

```

Figure 9-9 Principal RS-232-C electrical standards.

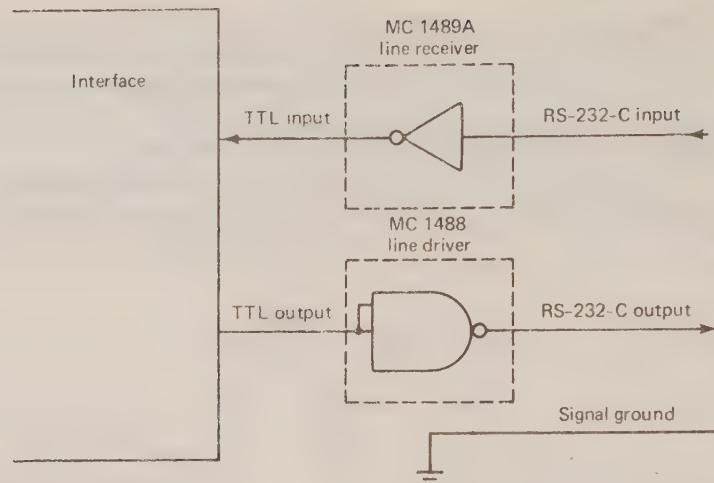


Figure 9-10 Driver circuit for RS-232-C send and receive lines.

Also part of the RS-232-C standard are the definitions and symbolic representations of the control lines that run between an interface or terminal and its modem. These definitions are summarized in Fig. 9-11, with the symbolic representation appearing in the first column, the standard name in the third column, and a brief description in the fourth column. Connections are most often made through 25-pin connectors and the second column gives the pin number of the pin to which each line is normally connected. Beside most symbols in the first column are numbers in parentheses. These numbers are the ones given to the lines by the CCITT standard.

Only the first eight lines in Fig. 9-11 are needed for communicating over a direct telephone line. For transmitting, a Request to Send (CA) signal is sent to the modem and this is acknowledged by a Clear to Send (CB). Then transmission may begin on the Transmitted Data (BA) line. For receiving, the Received Line Signal Detector (CF) line is activated by the modem to indicate that it is receiving a signal from the modem at the other end of the communication link, and the Received Data (BB) line is for sending the received data from the modem to the interface or terminal. The Data Set Ready (CC) indicates that the modem is turned on and is in its data mode. This line must be active when either transmitting or receiving.

If the communication link is part of a switched telephone system, at least two more lines are required. Signals are sent to the modem over the Data Terminal Ready (CD) line to control the switching of the modem and its related equipment so that a communication link is established. The Ring Indicator (CE) line is activated by the modem to indicate to the interface or terminal that the modem is receiving a ringing signal. To answer this signal the interface or terminal would respond by activating the Data Terminal Ready line. This would cause the modem and its switching equipment to "answer the ringing phone."

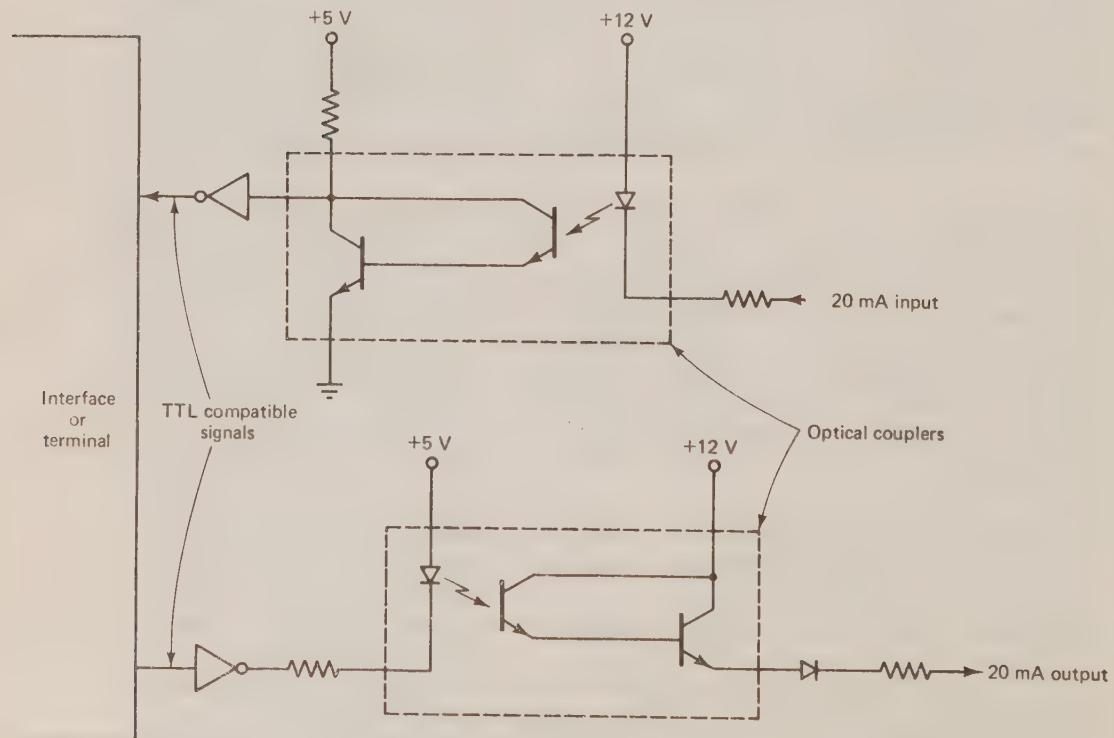
Figure 9-11 Summary of RS-232-C control line definitions.

| Symbol EIA (CCITT) | Pin | Name | Description |
|-----------------------|-----|--|--|
| AA (101) | 1 | Protective Ground | Used as an equipment ground |
| AB (102) | 7 | Signal Ground | Used as a common ground for all signals |
| BA (103) | 2 | Transmitted Data | For outputting data to modem |
| BB (104) | 3 | Received Data | For inputting data from modem |
| CA (105) | 4 | Request to Send | To modem - to turn the modem's transmitter on and off when operating in half duplex |
| CB (106) | 5 | Clear to Send | From modem - indicates modem is ready to transmit |
| CC (107) | 6 | Data Set Ready | From modem - indicates that modem power is on and the modem is not in test mode |
| CF (109) | 8 | Received Line Signal Detector | From modem - indicates the modem is receiving a signal from the modem at the other end of the link |
| CD (108/2) | 20 | Data Terminal Ready | To modem - prepares the modem to be connected to the communication link and begin transmitting |
| CE (125) | 22 | Ring Indicator | From modem - indicates ringing signal is detected on link |
| CG (110) | 21 | Signal Quality Detector | From modem - active when there is a low probability of error in the received data |
| CH (111) | 23 | Data Signal Rate Selector (DTE Source) | To modem - indicates to the modem one of two synchronous data rates or ranges of rates |
| CI (112) | 23 | Data Signal Rate Selector (DCE Source) | From modem - indicates to interface or terminal one of two synchronous data rates or ranges of rates |
| DA (113) | 24 | Transmitter Signal Element Timing (DTE Source) | To modem - provides modem transmitter with signal timing |
| DB (114) | 15 | Transmitter Signal Element Timing (DCE Source) | From modem - provides interface or terminal with transmitter signal timing |
| DD (115) | 17 | Receiver Signal Element Timing | From modem - provides interface or terminal receiver with signal timing |
| SBA (118) | 14 | Secondary Transmitted Data | To modem - for outputting low rate data |
| SBB (119) | 16 | Secondary Received Data | From modem - for inputting low rate data |
| SCA (120) | 19 | Secondary Request to Send | To modem - to turn on modem's secondary transmitter |
| SCB (121) | 13 | Secondary Clear to Send | From modem - indicates secondary transmitter is ready |
| SCF (122) | 12 | Secondary Received Line Signal Detector | From modem - indicates signal is detected on secondary link |

Synchronous communication requires that timing information be communicated along with the data. There are two lines for providing this information: the Transmitter Signal Element Timing lines. The DTE Source (DA) line is for sending timing information from the interface or terminal to the modem and the DCE Source (DB) line is for transmitting timing information in the other direction. Because modems for synchronous transmissions often accommodate two data rates or two ranges of data rates, the Data Signal Rate Selector (CH and CI) lines are for indicating rate information. CH is for sending a rate selection from the interface or terminal to the modem and CI is used by the modem to indicate to the interface or terminal the data rate it is currently using.

For both asynchronous and synchronous high-speed transmission it has been found to be more efficient to send the information in packets containing identification and error detection information as well as the data. Packets are normally used when computers are communicating with each other. In connection with the transmission of packets a certain amount of handshaking and verification must take place and this can best be done using a secondary low-rate communication link. The last five lines defined in Fig. 9-11 are for these secondary transmissions. Because a full discussion of packets and their associated protocols would take more

Figure 9-12 Circuits for driving and receiving 20-mA loop signals.



space than we have available, these lines are not considered further here. For more information the reader should refer to Reference 1.

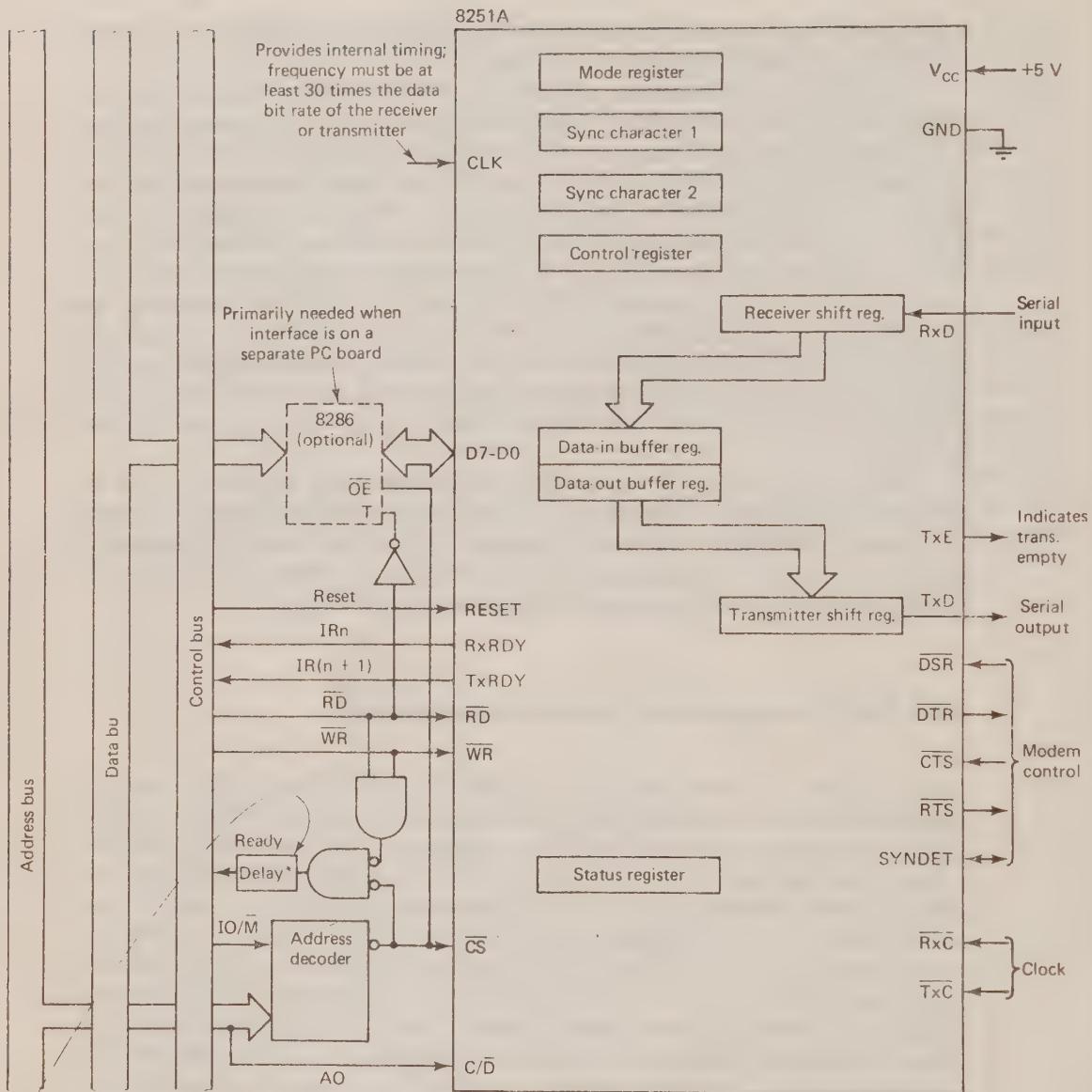
It should be mentioned that there is a second electrical standard that is used for transmitting data back and forth between an interface or terminal and a modem. It is called the *20-mA loop standard* and is implemented by having the line consist of a loop that carries a 20-mA current when it is in a mark (1) state and does not carry a current when it is in a space (0) state. In a 20-mA loop configuration one of the communicating devices must be the active device and provide the current source, while the other device would be passive. Historically, this standard began with mechanical teleprinters that used the current for driving their solenoids, but even most CRT terminals have the option of communicating over either an RS-232-C line or a 20-mA line. However, as the need for current signals recedes, so does the use of the 20-mA loop standard and RS-232-C links are far more prevalent today. Circuits for driving and receiving 20-mA loop signals are shown in Fig. 9-12.

Many of the details of the above standards were not discussed. There are several books on data communications, some of which are listed in the references at the end of the chapter. The above discussion was directed toward the connection between the computer interface and the communication system, and included primarily information that is needed to understand interface design.

9-1-4 8251A Programmable Communication Interface

As an example of a serial interface device let us consider Intel's 8251A programmable communication interface. The 8251A is diagrammed in Fig. 9-13. It is capable of being programmed for asynchronous or synchronous communication. The data-in buffer and data-out buffer registers share the same port address. For input, the serial bit stream arriving on the RxD pin is shifted into the receiver shift register and then the data bits are transferred to the data-in buffer register, where they can be input by the CPU. Conversely, on output the data bits put in the data-out buffer register by the CPU are transferred to the transmitter shift register and, along with the necessary synchronization bits, are shifted out through the TxD pin. Among other things the contents of the mode register, which are initialized by the executing program, determine whether the 8251A is in asynchronous mode or synchronous mode and the format of the characters being received and transmitted. The control register, which is also set by the program, controls the operation of the interface, and the status register makes certain information available to the executing program. Clearly, the sync character registers are for storing the sync characters needed for synchronous communication.

Even though all seven of the registers on the left side of Fig. 9-13 can be accessed by the processor, the 8251A is associated with only two port addresses. The C/D pin is connected to the address line A0 and A0 differentiates the two port



* A 1-wait state delay may be required if external transceivers are used in a minimum mode system.

Figure 9-13 8251A serial communication interface.

addresses. The 8251A internally interprets the C/D, RD, and WR signals as follows:

| C/D (= A0) | RD | WR | |
|------------|----|----|--|
| 0 | 0 | 1 | Data input from the data-in buffer |
| 0 | 1 | 0 | Data output to the data-out buffer |
| 1- | 0 | 1 | Status register is put on data bus |
| 1 | 1 | 0 | Data bus is put in mode, control, or sync character register |

where 1 means that the pin is high and 0 means that it is low. All other combinations cause the three-state D7-D0 pins to go into their high-impedance state.

Whether the mode, control, or sync character register is selected depends on the accessing sequence. A flowchart of the sequencing is given in Fig. 9-14. After a hardware reset or a command is given with its reset bit set to 1, the next output with A0=1 (i.e., with C/D=1, RD=1, and WR=0) is directed to the mode register. The formats of the mode register for both the asynchronous and synchronous cases are defined in Fig. 9-15. If the two LSBs of the mode are zero, then the interface is put in its synchronous mode and the MSB determines the number of sync characters. In the synchronous mode, the next 1 or 2 bytes output with A0=1 become the sync characters. If the two LSBs of the mode are not both 0, then the 8251A enters its asynchronous mode. In either case, all subsequent bytes prior to another reset go to the control register if A0=1 and the data-out buffer register if A0=0.

In the synchronous mode the baud rates of the transmitter and receiver, which are the shift rates of the shift registers, are the same as the frequencies of the signals applied to TxC and RxC, respectively, but in the asynchronous mode the three remaining possible combinations for the two LSBs in the mode register dictate the baud rate factor. The relationship between the frequencies of the TxC and RxC clock inputs and the baud rates of the transmitter and receiver is:

$$\text{Clock frequency} = \text{Baud rate factor} \times \text{Baud rate}$$

If 10 is in the LSBs of the mode register and the transmitter and receiver baud rates are to be 300 and 1200, respectively, then the frequency applied to TxC should be 4800 Hz and the frequency at RxC should be 19.2 kHz.

In both the asynchronous and synchronous modes, bits 2 and 3 indicate the number of data bits in each character, bit 4 indicates whether or not there is to be parity bit, and bit 5 specifies the type of parity (odd or even). For the asynchronous mode the two MSBs indicate the number of stop bits, but for the synchronous mode bit 6 determines whether the SYNDET pin is to be used as an input or as an output and, as mentioned above, bit 7 indicates the number of sync characters.

If the SYNDET pin is used as an output it becomes active when a bit-for-bit match has been found between the incoming bit stream and the sync character(s). If the search for sync characters is conducted by an external device, then SYNDET can be used to input a signal, indicating that a match has been found by the external device. The pin also has a meaning during asynchronous operation, but in this case it can only be an output. This output is called the break detect signal and goes high whenever a character consisting of all 0s is received.

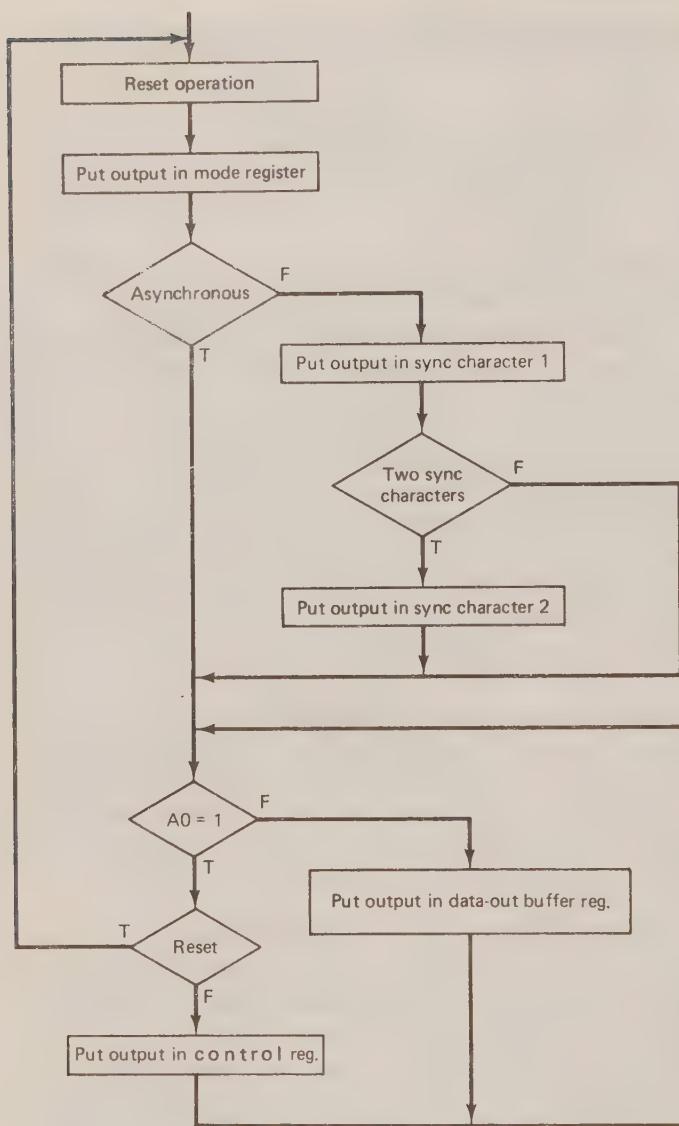


Figure 9-14 Flowchart of the disposition of output.

The format for the control register is given in Fig. 9-16. Bit 0 of this register must be 1 before data can be output and bit 2 must be 1 before data can be received. Programmed answering of a modem is accomplished by setting bit 1 to 1 since this forces the DTR pin to 0 and the complement of DTR is normally connected to the CD line from the modem. Bit 3 equal to 1 forces TxD to 0, thus causing break characters to be transmitted. Setting bit 4 to 1 causes all the error bits in the status register to be cleared (the bits that are set when framing, overrun, and parity errors occur). Bit 5 is used for sending a Request to Send signal to a modem. If the complement of the RTS pin is connected to a modem's CA line, then a 1 put in

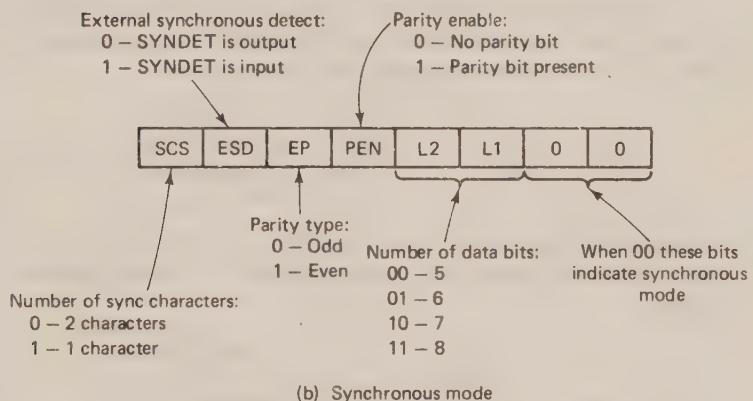
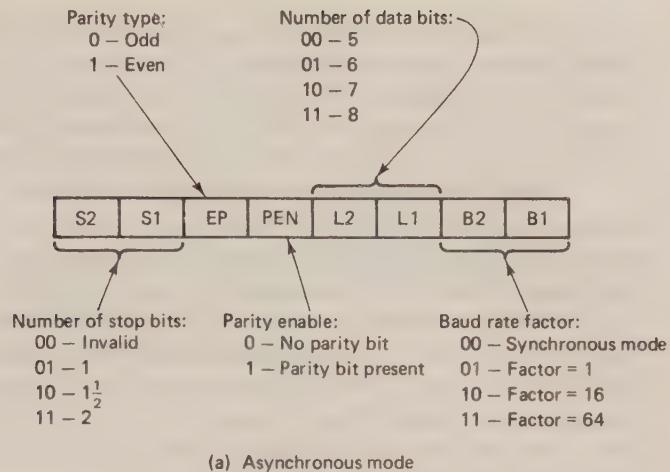
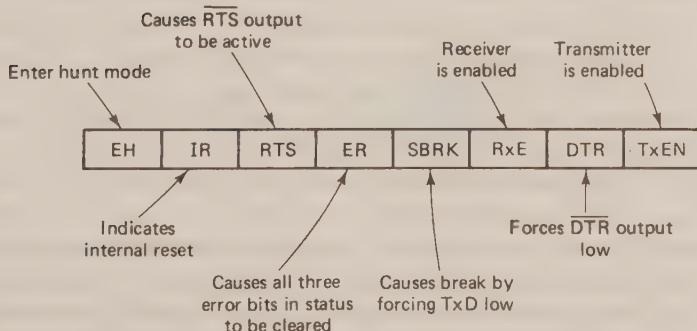


Figure 9-15 Format of the mode register.

Figure 9-16 Format of the control register.



Note: In all cases action is taken when the bit is set to 1.

bit 5 will cause the CA line to go high. Setting bit 6 causes the 8251A to be reinitialized and the reset sequence to be reentered (i.e., a return is made to the top of the flowchart shown in Fig. 9-14 and the next output will be to the mode register). Bit 7 is used only with the synchronous mode. When set, it causes the 8251A to begin a bit-by-bit search for a sync character or sync characters.

Typical connections to modems for asynchronous and synchronous transmissions are shown in Fig. 9-17. With regard to the synchronous connections it is assumed that the timing is controlled by the modem and its related communications equipment. Also, if this equipment is used to detect the sync character(s) at the beginning of a received message, then it can inform the 8251A of its success over the SYNDET line. On the other hand, if 8251A searches for the sync character(s), then it can use the SYNDET line as an output to tell the modem that the sync character(s) has been found. To satisfy the RS-232-C standard, drivers and receivers are needed to convert the TTL-compatible signals at TxD and RxD to the proper voltage levels (see Fig. 9-10).

A program sequence which initializes the mode register and gives a command to enable the transmitter and begin an asynchronous transmission of 7-bit characters followed by an even-parity bit and 2 stop bits is:

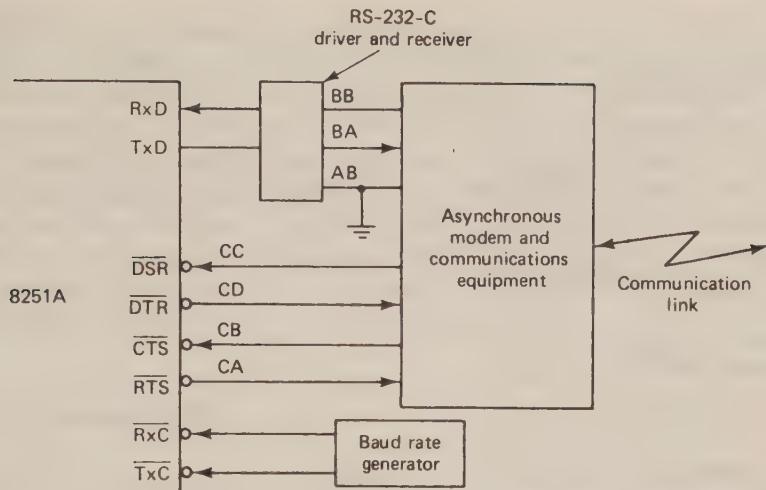
```
MOV AL,11111010B
OUT 51H,AL
MOV AL,00110011B
OUT 51H,AL
```

This sequence assumes that the mode and control registers are at address 51H and the clock frequencies are to be 16 times the corresponding baud rates. The sequence:

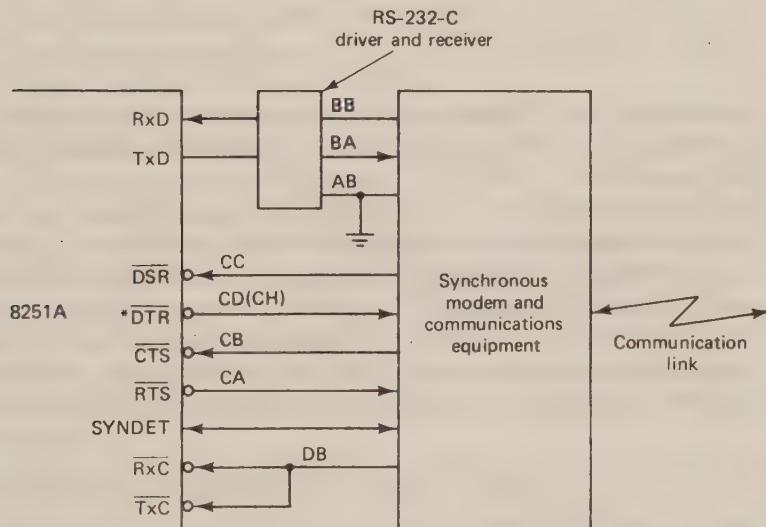
```
MOV AL,00111000B
OUT 51H,AL
MOV AL,16H
OUT 51H,AL
OUT 51H,AL
MOV AL,10010100B
OUT 51H,AL
```

would cause the same 8251A to be put in synchronous mode and to begin searching for two successive ASCII sync characters. As before, the characters are to consist of 7 data bits and an even parity bit, but there will, of course, be no stop bits.

The format of the status register is given in Fig. 9-18. Bits 1, 2, and 6 reflect the signals on the RxRDY, TxE, and SYNDET pins. TxRDY indicates that the data-out buffer is empty. Unlike the TxRDY pin, this bit is not affected by the CTS input pin or the TxEN control bit. RxRDY indicates that a character has been received and is ready to be input to the processor. Either the TxRDY and RxRDY bits can be used for programmed I/O or the signals on the corresponding pins can be connected to interrupt request lines to provide for interrupt I/O. The TxRDY bit is automatically cleared when a character is made available for transmitting and the RxRDY bit is automatically cleared when the character that set it is input by the processor. Bit 2 indicates that the transmitter shift register is waiting



(a) Asynchronous modem connection



*May also be used for rate selection — modem line CH.

(b) Synchronous modem connection

NOTE: Control line interfacing depends on modem specifications.

Figure 9-17 8251A modem connections.

to be sent a character from the data-out buffer register. During synchronous transmissions, while this bit is set, the transmitter will take its data from the sync character registers until data are put in the data-out buffer register. Bits 3, 4, and 5 indicate parity, overrun, and framing errors, respectively. When an error is detected, the bit having the corresponding error type will be set to 1. If the com-

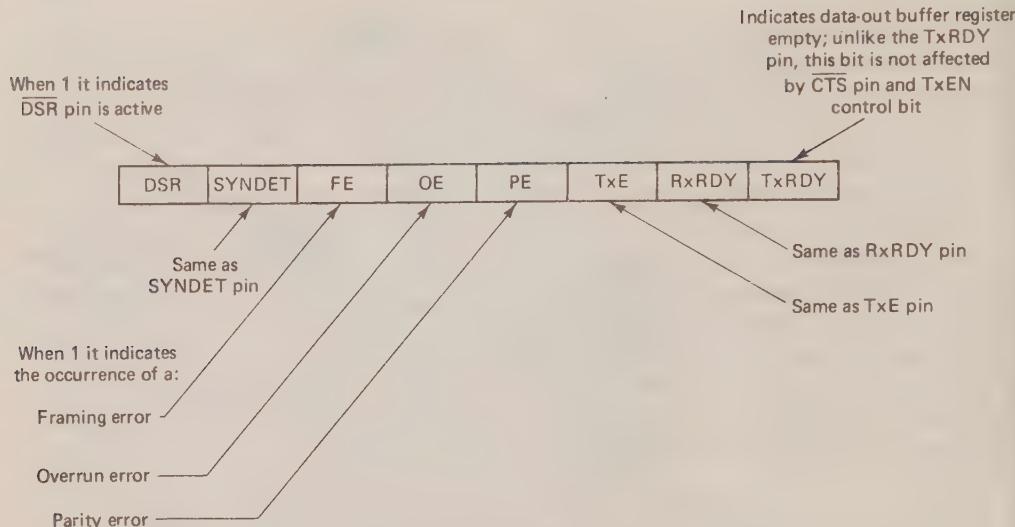


Figure 9-18 Format of the status register.

plement of the DSR pin is connected to the Data Set Ready (CC) line, then bit 7 reflects the state of the modem and is 1 when the modem is turned on and is in its data mode.

Figure 9-19 gives a typical program sequence which uses programmed I/O to input 80 characters from the 8251A, whose data buffer register's address is 0050, and put them in the memory buffer beginning at LINE. The inner loop continually tests the RxRDY bit until it is set by a character being put in the data-in buffer register. Then the newly arrived character is moved to the buffer and the error bits are checked. If the present character arrived before the previous character was input or a parity or framing error occurred during transmission, then the input

Figure 9-19 Inputting a line of characters through an 8251A.

```

MOV AL, 00110101B      ;ENABLE TRANSMITTER AND RECEIVER
OUT 51H, AL             ;AND CLEAR ERROR BITS
MOV DI, 0                ;INITIALIZE INDEX
MOV CX, 80               ;PUT COUNT IN CX
BEGIN: IN AL, 51H         ;WAIT FOR INPUT
TEST AL, 02H
JZ BEGIN
IN AL, 50H               ;INPUT CHARACTER AND
MOV LINE[DI], AL          ;PUT IN LINE BUFFER
INC DI
IN AL, 51H               ;CHECK ERROR
TEST AL, 00111000B
JNZ ERROR
LOOP BEGIN               ;IF NO ERROR IS FOUND
                           ;CONTINUE INPUTTING
JMP SHORT EXIT
ERROR: CALL NEAR PTR ERR_ROUT ;ELSE CALL ERR_ROUT
EXIT:
.
.
.

```

ceases and a call is made to an error routine that would presumably examine the individual error bits, print an appropriate message, and clear the error bits.

Because inputting a character automatically resets the RxRDY bit, unless another character is received before the inner loop is reentered, the inner loop must cycle until the RxRDY bit is reset to 1 by the next incoming character. If the incoming characters have fewer than 8 bits, the unused MSBs in the data buffer register are always zeroed. Also, the parity bit is not passed to the processor and checks for parity errors can only be made by examining the parity error bit in the status register. On output, if a character is less than 8 bits long, the unneeded MSBs in the data-out buffer register are ignored.

9-2 PARALLEL COMMUNICATION

Parallel communication is accomplished by simultaneously transferring bits over separate lines. Its advantage over serial communication is that for lines with a given maximum bit rate, higher information rates can be attained due to the use of several lines. The disadvantage is, of course, the cost of the extra lines and, because these costs increase with distance, parallel communication is used over long distances only if high data rates are required.

Unlike serial communication, parallel communication has not been well standardized. Parallel transfers may be performed by simply putting data in the interface's data-in buffer register or taking data out of the interface's data-out buffer register, or they may be conducted under the control of handshaking and/or timing signals. Normally, one character (or other piece of information) is transferred at a time and there is no problem in determining the end of a character or in finding the beginning of a transmission. There is no well-defined format for synchronous or asynchronous transmissions, but if a timing signal is to coordinate the activity between the device and the interface, the transmission would be considered synchronous. If only handshaking signals are employed, the transmissions would be asynchronous.

An interface may be designed for only outputting, only inputting, inputting and outputting over separate sets of lines, or performing I/O over a single set of bidirectional lines. If an interface is connected to a line printer, it would only output data, and if it services a card reader, it would only input data. An interface that services both a paper tape reader and a paper tape punch would require one set of input lines and one set of output lines, but an interface for a device that does not input and output data simultaneously could use only one set of bidirectional lines.

A typical parallel output interface which uses no control lines is given in Fig. 9-20(a). In this case the contents of the data buffer register would continually be maintained on the data lines connecting the interface and the output device. If the output device consisted of a latch and a set of relays that are controlled by the bit settings in the latch, then the computer could control the relays by simply outputting data to the data buffer register.

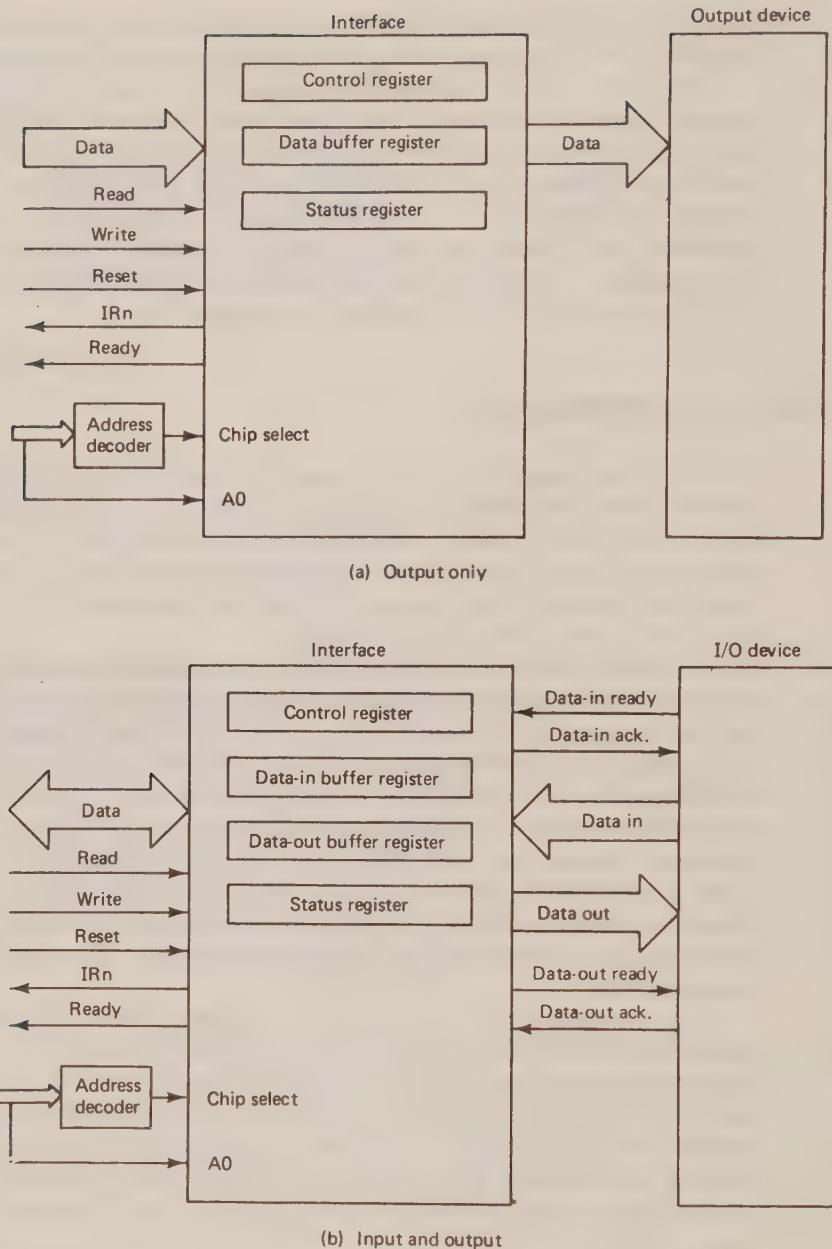


Figure 9-20 Representative parallel communication interfaces.

Figure 9-20(b) shows a representative parallel interface with handshaking control lines and separate input and output connections to an I/O device. In this case, an input would be carried out by first putting the data on the data bus and a 1 on the data-in ready line. The interface would respond by latching the data into the data-in buffer and putting a 1 on the data-in acknowledge line. Upon receiving the acknowledgment the device would drop the data and ready signals. When the data are received by the interface, it would set a "ready" status bit and perhaps send out an interrupt request. After the CPU takes the data the interface would clear the "ready" status bit and put the data lines in their high-impedance state. If an interrupt request were made, it would be handled in the usual manner.

For an output, the interface would set a "ready" status bit and, perhaps, make an interrupt request when the data-out buffer is available. After the CPU outputs the data, the interface would clear the output "ready" status bit, put the data on the data-out bus, and signal the I/O device over the data-out ready line. When the device is ready to accept the data, it would latch the data and then return an acknowledgment. The interface would then drop its data-out ready signal and once again set the output "ready" status bit.

9-2-1 8255A Programmable Peripheral Interface

Intel's 8255A programmable peripheral interface provides a good example of a parallel interface. As shown in Fig. 9-21, the interface contains a control register and three separately addressable ports, denoted A, B, and C. Whether or not an 8255A is being accessed is determined by the signal on the \overline{CS} pin and the direction of the access is according to the \overline{RD} and \overline{WR} signals. Which of the four registers is being addressed is determined by the signals applied to the pins A1 and A0. Therefore, the lowest port address assigned to an 8255A must be divisible by 4. A summary of the 8255A's addressing is:

| A1 | A0 | \overline{RD} | \overline{WR} | \overline{CS} | Transfer Description |
|----|----|-----------------|-----------------|-----------------|---|
| 0 | 0 | 0 | 1 | 0 | Port A to data bus |
| 0 | 1 | 0 | 1 | 0 | Port B to data bus |
| 1 | 0 | 0 | 1 | 0 | Port C to data bus |
| 0 | 0 | 1 | 0 | 0 | Data bus to port A |
| 0 | 1 | 1 | 0 | 0 | Data bus to port B |
| 1 | 0 | 1 | 0 | 0 | Data bus to port C |
| 1 | 1 | 1 | 0 | 0 | Data bus to control register if D7=1; if D7=0, input from the data bus is treated as a Set/ Reset instruction |
| x | x | x | x | 1 | D7-D0 go to high-impedance state |
| 1 | 1 | 0 | 1 | 0 | Illegal combination |
| x | x | 1 | 1 | 0 | D7-D0 go to high-impedance state |

where 0 is low and 1 is high.

Because the bits in port C are sometimes used as control bits, the 8255A is designed so that they can be output to individually using a Set/Reset instruction. When the 8255A receives a byte that is directed to its control register, it examines

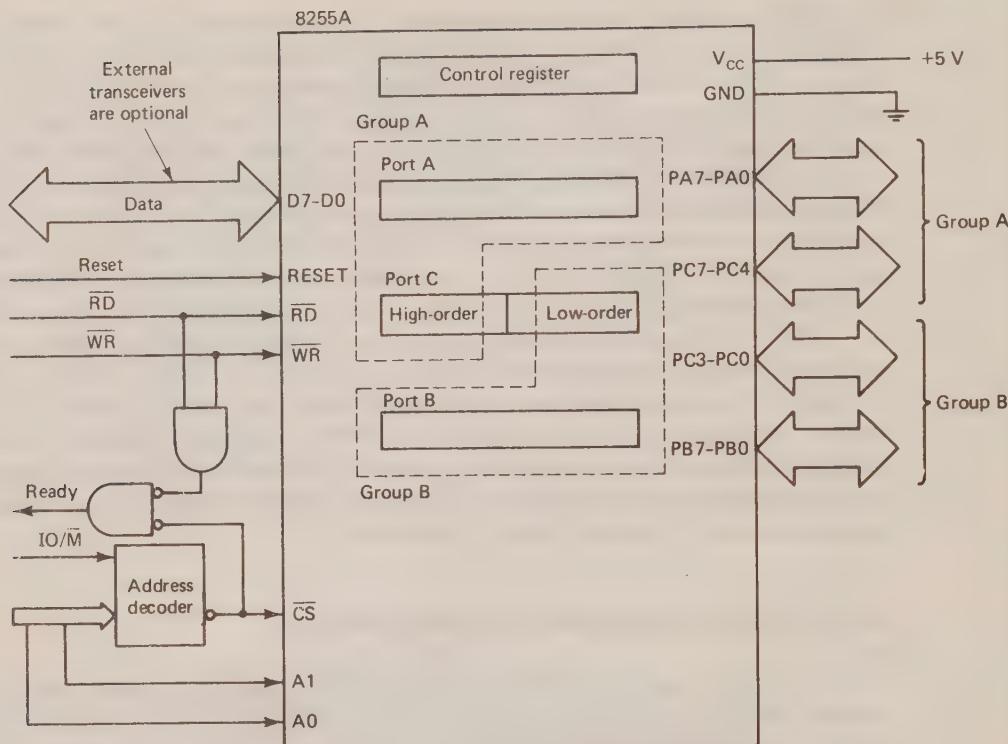


Figure 9-21 Diagram of the 8255A.

the data bit 7. If this bit is 1, the data are transferred to the control register, but if it is 0, the data are treated as a Set/Reset instruction and is used to set or clear the port C bit specified by the instruction. Bits 3-1 give the bit number of the bit to be changed and bit 0 indicates whether it is to be set or cleared. The remaining bits are unused.

The bits in the three ports are attached to pins that may be connected to the I/O device. These bits are divided into groups A and B, with group A consisting of the bits in port A and the four MSBs of port C and group B consisting of port B and the four LSBs of port C. The use of each of the two groups is controlled by the mode associated with it. Group A is associated with one of three modes, mode 0, mode 1, and mode 2, and group B with one of two modes, mode 0 and mode 1. The modes are determined by the contents of the control register, whose format is given in Fig. 9-22. These modes are:

Mode 0—If a group is in mode 0, it is divided into two sets. For group A these sets are port A and the upper 4 bits of port C, and for group B they are port B and the lower 4 bits of port C. Each set may be used for inputting or outputting, but not both. Bits D4, D3, D1, and D0 in the control register

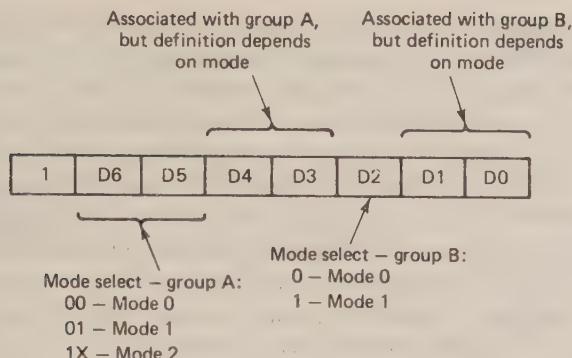


Figure 9-22 Format of the 8255A's control register.

specify which sets are for input and which are for output. These bits are associated with the sets as follows:

D4—Port A.

D3—Upper half of port C.

D1—Port B.

D0—Lower half of port C.

If a bit is 0, then the corresponding set is used for output; if it is 1, the set is for input.

Mode 1—When group A is in this mode port A is used for input or output according to bit D4 (D4=1 indicates input), and the upper half of port C is used for handshaking and control signals. For *inputting*, the four MSBs of port C are assigned the following symbols and definitions:

| | | |
|----------------|------------------------|---|
| PC4 | STB_A | A 0 applied to this pin causes PA7-PA0 to be latched, or "strobed," into port A. |
| PC5 | IBF_A | Indicates that the input buffer is full. It is 1 when port A contains data that have not yet been input to the CPU. When a 0 is on this pin the device can input a new byte to the interface. |
| PC6,PC7 | | May be used to output control signals to the device or input status from the device. If D3 of the control register is 0, they are for outputting control signals; otherwise, they are for inputting status. |

For *outputting*:

| | | |
|----------------|------------------------|--|
| PC4,PC5 | | Serve the same purpose as described above for PC6 and PC7. |
| PC7 | OBF_A | Indicates that the output buffer is full. It outputs a 0 to the device when port A is outputting new data to be taken by the device. |
| PC6 | ACK_A | Device puts a 0 on this pin when it accepts data from port A. |

In mode 1, PC3 is denoted INTR_A and is associated with group A. It is used as an interrupt request line and is tied to one of the IR lines in the system

bus. When inputting to port A, this pin becomes 1 when new data are put in port A (i.e., it is controlled by PC4) and is cleared when the CPU takes the data. For output, this pin is set to 1 when the contents of port A are taken by the device and is cleared when new data are sent from the CPU. If group B is in mode 1, port B is input to or output from according to bit D1 of the control register (D1 = 1 indicates input). For input, PC2 and PC1 are denoted \overline{STB}_B and IBF_B , respectively, and serve the same purposes for group B as \overline{STB}_A and IBF_A do for group A. Similarly, for output PC1 and PC2 are denoted \overline{OBF}_B and \overline{ACK}_B . PC0 becomes $INTR_B$ and its use is analogous to that of $INTR_A$. The interrupt enable for group A is controlled by setting or clearing internal flags. Setting or clearing these flags is simulated by setting or clearing PC4, for input, or PC6, for output, using a Set/Reset instruction. Similarly, the interrupt enable for group B is controlled by set/clear of PC2 for both input and output.

Mode 2—This mode applies only to group A, although it also uses PC3 for making interrupt requests. In mode 2, port A is a bidirectional port and the four MSBs of port C are defined as follows:

| | | |
|------------|--------------------------------------|---|
| PC4 | STB_A | A 0 on this line causes the data on PA7-PA0 to be strobed into port A. |
| PC5 | IBF_A | Becomes 1 when port A is filled with new data from lines PA7-PA0 and is cleared when these data are taken by the CPU. |
| PC6 | \overline{ACK}_A | Indicates that the device is ready to accept data from PA7-PA0. |
| PC7 | \overline{OBF}_A | Becomes 0 when port A is filled with new data from the CPU and is set to 1 when data are taken by the device. |

While group A is in mode 2, group B may be in either mode 0 or mode 1. However, if group B is in mode 0, only PC2-PC0 can be used for input or output because group A has borrowed PC3 to use as an interrupt request line. Normally, if group A is in mode 2, PC2-PC0 would be connected to control and status pins on the device attached to the port A lines. Port B may also be used for this purpose.

In all three modes port C reflects the signals on PC7-PC0 and can be read with an IN instruction.

9-2-2 A/D and D/A Example

Figure 9-23 shows how an 8255A could be connected to an A/D and D/A subsystem. Since during an A/D conversion the analog voltage must remain unchanged, a sample-and-hold circuit is needed to keep the analog signal constant while the conversion is being performed. Group A is configured as an input in mode 1. A conversion is initiated by a signal from the 8255A's PC7 pin, which prompts the converter to output a busy signal. The busy line is connected to both a sample-

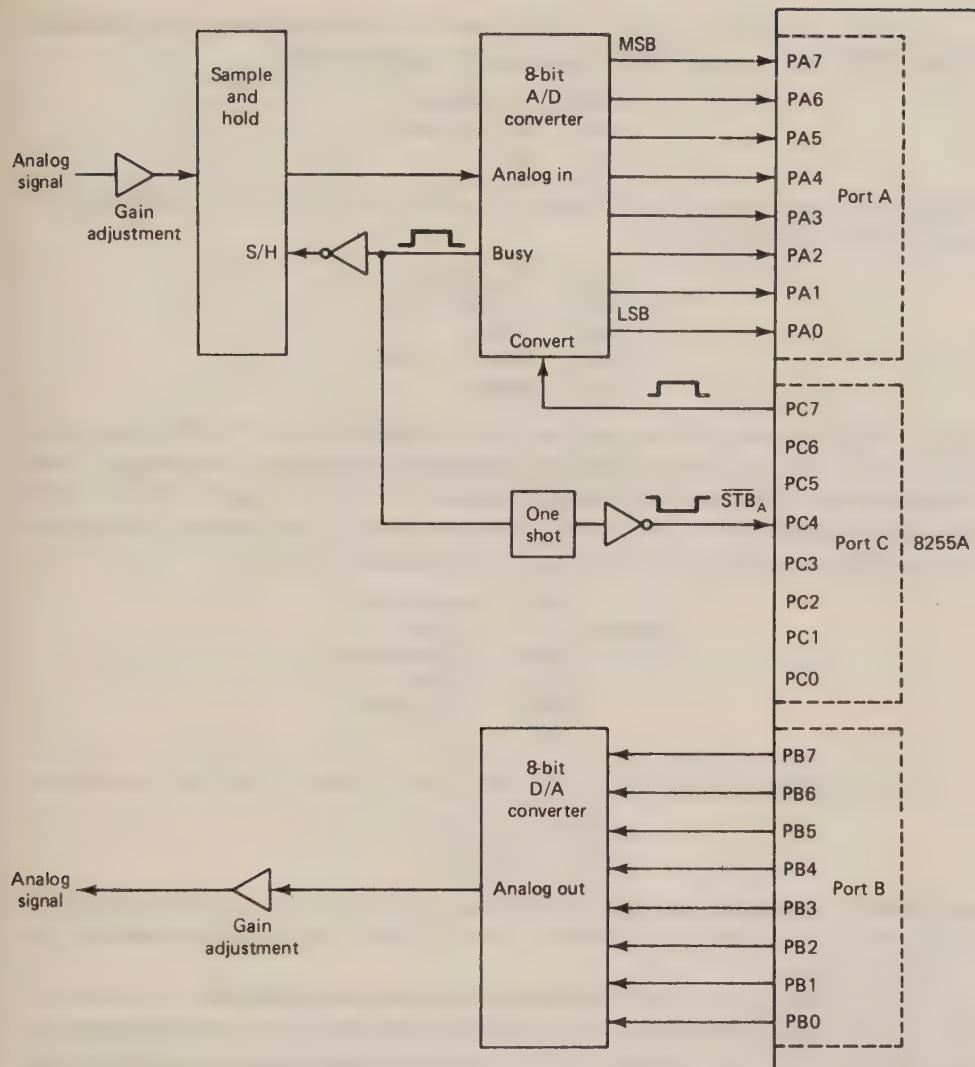


Figure 9-23 Interfacing an A/D and D/A subsystem using an 8255A.

and-hold control pin (S/H) and a negative edge-triggered one-shot. While the busy signal is high the sample-and-hold circuit maintains a constant output, and when the busy signal goes down at the end of the conversion the one-shot is triggered. The output from the one-shot is complemented and applied to the \overline{STB}_A (PC4) input on the 8255A. This causes the digitized sample to be strobed into port A. For the D/A portion of the subsystem, port B is configured as an output port in mode 0, which is directly connected to the binary input of the D/A converter. No handshaking is used.

Given that port A, port B, port C, and the control register have addresses FFF8, FFF9, FFFA, and FFFF, respectively, the sequence:

```
MOV DX,0FFFFBH
MOV AL,10110000B
OUT DX,AL
```

would cause port A to be put in mode 1, port B to be put in mode 0, and PC7 to be an output. The sequence:

```
MOV DX,0FFFFBH
MOV AL,00001111B
OUT DX,AL
MOV AL,00001110B
OUT DX,AL
```

would output a pulse to the convert pin of the A/D converter. The first instruction of the latter sequence puts the address associated with Set/Reset instruction, which is the same as the address of the control register, in the DX register. The next two instructions cause PC7 to be set and the last two cause it to be cleared. A sequence for providing a programmed I/O input of the converted data is:

```
AGAIN: MOV DX,0FFFAH
       IN  AL,DX
       TEST AL,00100000B
       JZ  AGAIN
       MOV DX,0FFF8H
       IN  AL,DX
```

For outputting a byte from AL to the D/A converter, only the instructions

```
MOV DX,0FFF9H
OUT DX,AL
```

are needed. As soon as the byte arrives at port B its bits are immediately applied to the input pins of the D/A converter, which, in turn, immediately converts it to an analog signal.

In this example it has been assumed that the timing of the conversions is provided by the program and that the gains of the input and output analog amplifiers are manually adjusted. To obtain even spacing between input or output samples from the program the execution times of the instructions must be used. This means that exactly the same instructions be executed between samples and that the total execution time of these instructions be accurately known. Interrupts would need to be disabled because they could randomly insert the execution of different numbers of instructions. The sample time could be adjusted by including a "do nothing" loop, such as:

```
IDLE:  MOV CX,N
       NOP
       LOOP  IDLE
```

between the inputs or outputs. A flowchart for inputting a block of A/D samples using programmed timing is given in Fig. 9-24.

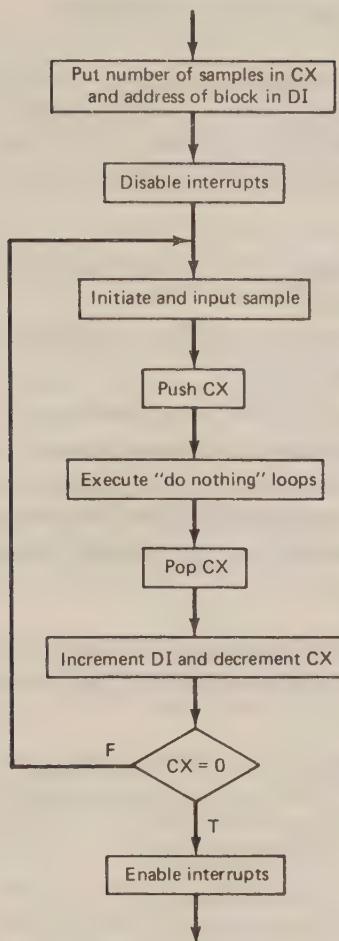


Figure 9-24 Programmed sample timing.

Frequently, programmable clocks and gain control devices are associated with A/D and D/A subsystems so that the timing of the sampling and D/A outputs are more precisely controlled and the gains can be dynamically changed. Also, a DMA controller is often used in conjunction with an A/D and D/A subsystem so that high-speed input and output can be attained.

Only 8-bit A/D and D/A converters are included in the design shown in Fig. 9-23. This limits the resolution to only 1 part in 256. If the voltage range of the input or output were -10 V to $+10\text{ V}$, the resolution would be:

$$\frac{20}{256} = 0.078\text{ V}$$

For higher resolutions, 10-, 12-, or 14-bit converters are required. To accommodate the greater number of bits, a combination of ports A and C or ports B and C could be used (see Exercise 14) or two 8255As could be used in parallel (see Sec. 9-7).

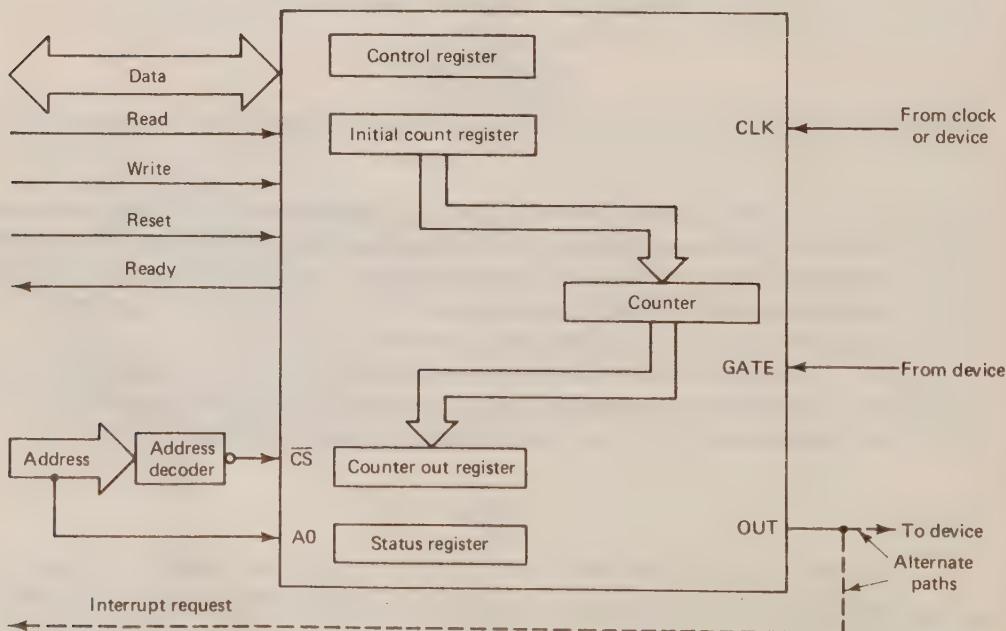
9-3 PROGRAMMABLE TIMERS AND EVENT COUNTERS

Quite often a device is needed to mark intervals of time for both the processor and external devices, count external events and make the count available to the processor, and provide external timing that can be programmed from the processor. Such a device is called a *programmable interval timer/event counter* and some of its uses are to:

1. Interrupt a time-sharing operating system at evenly spaced intervals so that it can switch programs.
2. Output precisely timed signals with programmed periods to an I/O device (e.g., an A/D converter).
3. Serve as a programmable baud rate generator.
4. Measure time delays between external events.
5. Count the number of times an event occurs in an external experiment and provide a means of inputting the count to the computer.
6. Cause the processor to be interrupted after a programmed number of external events have occurred.

A representative design of an interval timer/event counter is given in Fig. 9-25. In this design the four registers on the left are accessible by the computer, with

Figure 9-25 Typical interval timer/event counter.



the upper two being output ports and the lower two input ports. The counter itself is not directly available to the processor, but must be initialized from the initial count register and can be read only by first transferring its contents to the counter out register. The counter operates by starting at an initial value and counting backward to 0. The CLK input determines the count rate, GATE is for enabling and disabling the CLK input and perhaps other purposes, and the OUT output becomes active when the count reaches 0 or, possibly, when a GATE signal is received. OUT may be connected to an interrupt request line in the system bus so that an interrupt will occur when the count reaches 0, or to an I/O device which uses it to initiate specific I/O activity.

The operation is basically to enter a count in the initial count register, transfer the count to the counter, and cause the counter to count backward as pulses are applied to the CLK input. The current contents of the counter can be input at any time without disturbing the count by transferring them to the counter out register and then reading them. By buffering the count through the counter out register it does not have to be input to the processor immediately. The zero count indication would normally be applied to both the OUT pin and one of the bits in the status register. Thus either programmed I/O or interrupt I/O could be used to detect the zero count.

Among other things the control register includes the mode of operation. The mode determines exactly what happens when the count becomes 0 and/or a signal is applied to the gate input. Some possible actions are:

1. The GATE input is used for enabling and disabling the CLK input.
2. The GATE input may cause the counter to be reinitialized.
3. The GATE input may stop the count and force OUT high.
4. The count will give an OUT signal and stop when it reaches 0.
5. The count will give an OUT signal and automatically be reinitialized from the Initial Count Register when the count reaches 0.

The modes could be defined by combinations of these possibilities.

As an example, consider the application of an interval timer to a time-sharing operating system. In this case a clock would be connected to the CLK input and OUT to an interrupt request line, possibly to a nonmaskable line. The GATE input would not be needed. When the system is brought up the initial count register would be filled with

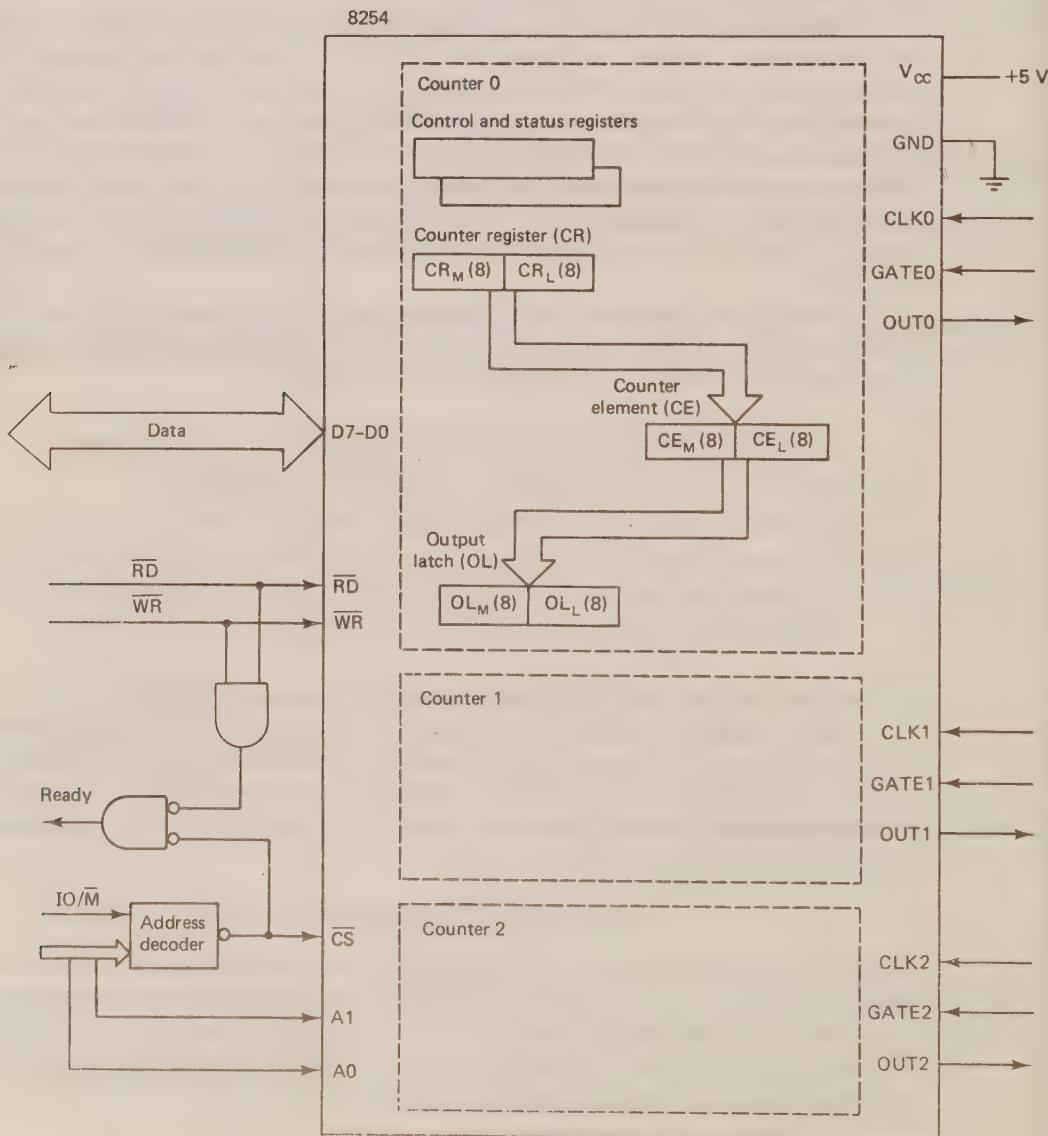
$$\text{Initial count} = \text{Clock frequency} \times T$$

where T is the length of each time slice in seconds, and the mode would be set so that each time the count reaches 0 the contents of the initial count register would be transferred to the counter and OUT would become active. Since OUT is used as an interrupt request, an interrupt routine for switching programs would be entered at the end of each period of T seconds.

9-3-1 Intel's 8254 Programmable Interval Timer

A diagram of Intel's 8254 interval/event counter is given in Fig. 9-26. The 8254 consists of three identical counting circuits, each of which has CLK and GATE inputs and an OUT output. Each can be viewed as containing a Control and Status Register pair, a Counter Register (CR) for receiving the initial count, a Counter Element (CE) which performs the counting but is not directly accessible from the

Figure 9-26 Diagram of the 8254.



processor, and an Output Latch (OL) for latching the contents of the CE so that they can be read. The CR, CE, and OL are treated as pairs of 8-bit registers. (Physically, the registers are not exactly as depicted, but to the programmer the figure is conceptually accurate.)

The registers can be accessed according to the following table:

| CS | RD | WR | A1 | A0 | Transfer |
|----|----|----|----|----|--|
| 0 | 1 | 0 | 0 | 0 | To counter 0 CR |
| 0 | 1 | 0 | 0 | 1 | To counter 1 CR |
| 0 | 1 | 0 | 1 | 0 | To counter 2 CR |
| 0 | 1 | 0 | 1 | 1 | To a control register or indicates a command |
| 0 | 0 | 1 | 0 | 0 | From counter 0 OL or status register |
| 0 | 0 | 1 | 0 | 1 | From counter 1 OL or status register |
| 0 | 0 | 1 | 1 | 0 | From counter 2 OL or status register |

where 0 means low and 1 means high. All other combinations result in the data pins being put into their high-impedance state. When A1 = A0 = 1, whether a control register is being written into or a command is being given depends on the MSBs of the byte being output. For the last three combinations, whether an OL or status register is read is determined by a previous command.

There are two types of commands, the counter latch command, which causes the CE in the counter specified by the two MSBs of the command to be latched into the corresponding OL, and the read back command, which may cause a combination of the CEs to be latched or "prepare" a combination of status registers to be read. To prepare a status register means to cause it to be read the next time a read operation inputs from the counter. When the two MSBs are 00, 01, or 10 a counter latch command is indicated, but if they are 11 a read back is to be performed. In a latch command bits 5 and 4 must be 0 and the remaining bits are unused. The read back command has the format:

| | | | | | | | |
|---|---|-------|------|------|------|------|---|
| 1 | 1 | COUNT | STAT | CNT2 | CNT1 | CNT0 | 0 |
|---|---|-------|------|------|------|------|---|

If the COUNT bit is 0, then the CEs for all of the counters whose CNT bits are 1 are latched. If CNT0 = CNT2 = 1 but CNT1 = 0, then the CEs in counters 0 and 2 are latched but the CE in counter 1 is not latched. Similarly, STAT = 0 causes the counters' status registers to be prepared for input. CEs can be latched and status registers can be prepared in the same command.

The formats of the control and status registers are given in Fig. 9-27. If the two MSBs of an output are both 1, they indicate that the output is to be a read back command; otherwise, they specify a counter. If they specify a counter and bits 4 and 5 are both 0, then a latch command is indicated and it is directed to the control register of the counter specified by the top 2 bits, but if they are not both 0, then they indicate the type of the input from OL or output to CR. The combination 01 indicates that the Read/Write operations are from/to the OL_L/CR_L , 10 indicates that they are from/to the OL_M/CR_M , and 11 indicates that these operations are to occur in pairs, with the first byte coming from/going to OL_L/CR_L and the second from/to OL_M/CR_M . A 1-byte write to CR will cause the other byte to be

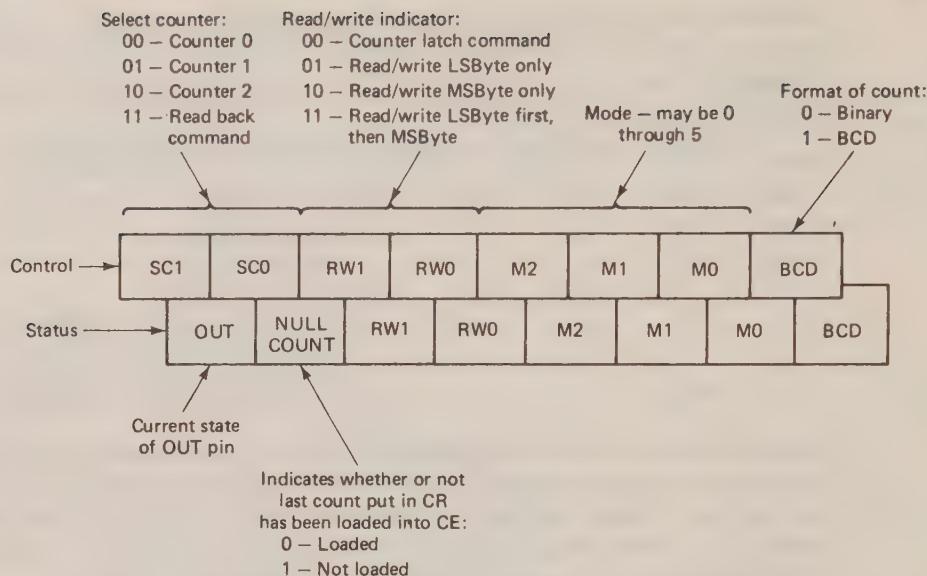


Figure 9-27 Control and status registers for 8254 counters.

zeroed. Bits 1, 2, and 3 determine the mode and bit 0 specifies the format of the count.

Given that N is the initial count, the modes are:

Mode 0 (Interrupt on Terminal Count)—GATE = 1 enables counting and GATE = 0 disables counting, and GATE has no effect on OUT. The contents of CR are transferred to CE on the first CLK pulse after CR is written into by the processor, regardless of the signal on the GATE pin. The pulse that loads CE is not included in the count. OUT goes low when there is an output to the control register and remains low until the count goes to 0. Mode 0 is primarily for event counting.

Mode 1 (Hardware Retriggerable One-Shot)—After CR has been loaded with N, a 0-to-1 transition on GATE will cause CE to be loaded, a 1-to-0 transition at OUT, and the count to begin. When the count reaches 0 OUT will go high, thus producing a negative-going OUT pulse N clock periods long.

Mode 2 (Periodic Interval Timer)—After loading CR with N, a transfer is made from CR to CE on the next clock pulse. OUT goes from 1 to 0 when the count becomes 1 and remains low for one CLK pulse; then it returns to 1 and CE is reloaded from CR, thus giving a negative pulse at OUT after every N clock cycles. GATE = 1 enables the count and GATE = 0 disables the count. A 0-to-1 transition on GATE also causes the count to be reinitialized on the next clock pulse. This mode is used to provide a programmable periodic interval timer.

Mode 3 (Square-Wave Generator)—It is similar to mode 2 except that OUT

goes low when half the initial count is reached and remains low until the count becomes 0. Hence the duty cycle is changed. As before, GATE enables and disables the count and a 0-to-1 transition on GATE reinitializes the count. This mode may be used for baud rate generation.

Mode 4 (Software-Triggered Strobe)—It is similar to mode 0 except that OUT is high while the counting is taking place and produces a one-clock-period negative pulse when the count reaches 0.

Mode 5 (Hardware-Triggered Strobe—Retriggerable)—After CR is loaded, a 0-to-1 transition on GATE will cause a transfer from CR to CE during the next CLK pulse. OUT will be high during the counting but will go low for one CLK period when the count becomes 0. GATE can reinitialize counting at any time.

For all modes, if the initial count is 0, it will be interpreted as 2^{16} or 10^4 depending on the format of the count. The above descriptions were only to provide an overall idea of the operation of the 8254 in the various modes. For a detailed description of the modes, the reader should look in Reference 2.

9-3-2 Interval Timer Application to A/D

Figure 9-28 shows how an 8254 could be used to provide a programmable sample rate generator for an A/D subsystem. By using all three of the 8254's counters, not only can the program set the sample rate, but it can also determine the period over which the samples are taken. Suppose that counter 0 is put in mode 2, counter 1 in mode 1, and counter 2 in mode 3 and that L, M, and N are their initial counts. If F is the frequency of the clock, then the frequency applied to CLK1 will be F/N . This will result in OUT1 producing a pulse having a period of MN/F . Therefore, pulses will occur at OUT0 at a frequency of F/L for a period of MN/F seconds. By applying OUT0 to the Convert input of the A/D converter, F/L samples per second will be taken for MN/F seconds beginning after the three counters have been initialized and the relay or manually operated switch has been closed. After each converted sample has been transmitted to port A of the 8255A, an interrupt request is made through the INTR_A and an interrupt routine is used to input the sample. An initialization sequence for the system is given in Fig. 9-29. The sequence assumes that the addresses associated with the 8254 are 0070 through 0073; LCNT, MCNT, and NCNT contain L, M, and N; and L and N are less than 256. It sets the counters to their modes and puts the initial counts in the CRs. The initial counts L and N are to be in binary and M is to be in BCD.

9-4 KEYBOARD AND DISPLAY

For low-cost small systems, especially single-board microcomputers and microprocessor-based instruments, the front panel (or *console*) is often implemented by using simple keyboard and display units as input and output devices. Through a

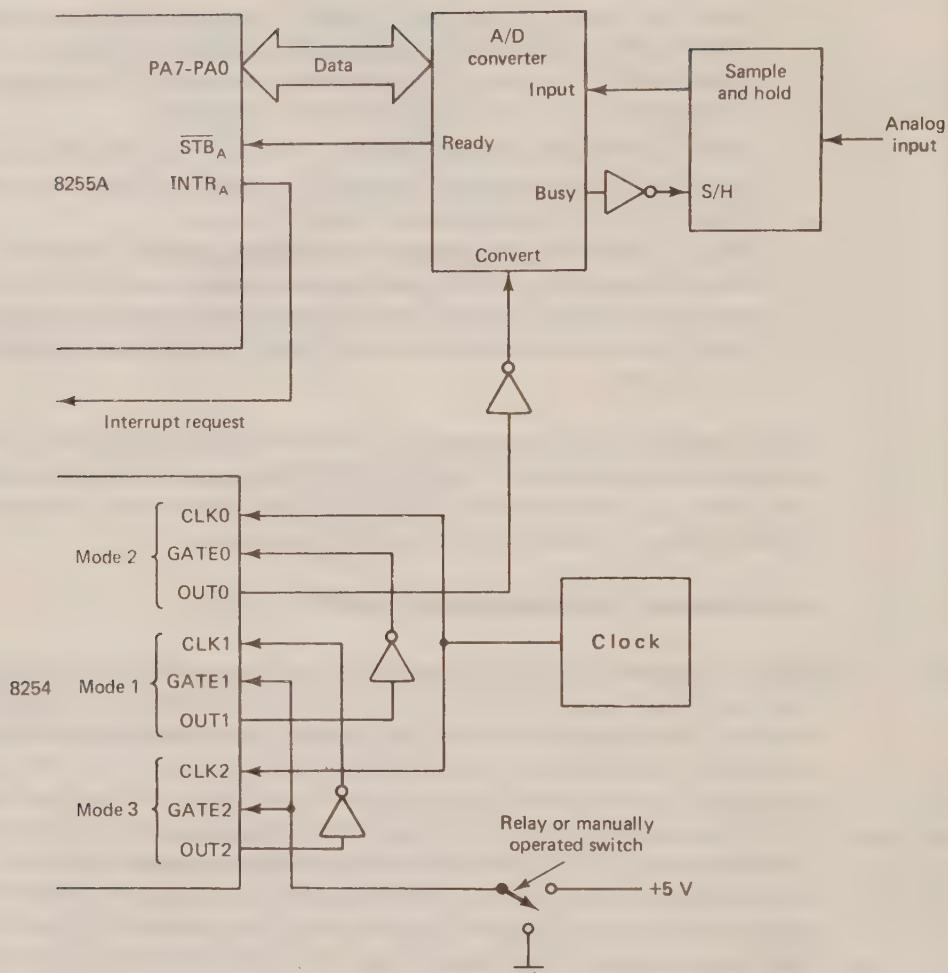


Figure 9-28 Interval timer example.

```

MOV AL,00010100B ;OUTPUT COUNTER 0
OUT 73H,AL ;CONTROL - MODE 2
MOV AL,LCNT ;OUTPUT COUNTER 0
OUT 70H,AL ;INITIAL COUNT - BINARY
MOV AL,01110011B ;OUTPUT COUNTER 1
OUT 73H,AL ;CONTROL - MODE 1
MOV AX,MCNT ;OUTPUT COUNTER 1
OUT 71H,AL ;INITIAL COUNT - BCD
MOV AL,AB
OUT 71H,AL ;OUTPUT COUNTER 2
MOV AL,10010110B ;CONTROL - MODE 3
OUT 73H,AL ;OUTPUT COUNTER 2
MOV AL,NCNT ;INITIAL COUNT - BINARY
OUT 72H,AL
.
.
.

```

Figure 9-29 Initialization of counters for the A/D example.

keyboard, data, memory addresses, and machine code can be entered in the hexadecimal form. In addition to the numeric keys, a keyboard may include functional keys to enter monitor and control commands. For output, memory addresses and data can be displayed on light-emitting diode (LED) display devices.

9-4-1 Keyboard Design

Unlike a terminal, a mechanical contact keyboard, for which the key switches are organized in a matrix form, does not include any electronics. Figure 9-30 illustrates how a 64-key keyboard can be interfaced to a microcomputer through two parallel I/O ports such as those provided by an 8255A. When a key is depressed, the corresponding row and column are shorted to form a path. By detecting the row and column positions of the contact closure, the code word representing the depressed key is determined. This process is called keyboard scanning and is accomplished as follows. The output port sends a 0 to row 0 and 1s to all 7 of the other rows. The column lines are then read and checked. If a 0 is not found in row 0, then the process is repeated for row 1, then for row 2, and so on. When a 0 is found, a depressed key is detected whose row position is known from the combination that was output and column position is known from the input. By combining the row and column positions of the 0, a unique word which indicates the position of the depressed key can be found.

There are two major problems associated with keyboards; they are contact bounce and striking keys at about the same time. When a key is depressed or released, the contact may bounce between its open and close position several times before it settles to a close or open position. The duration of the bouncing varies and is normally less than 10 ms. Contact closures due to bouncing must be discarded to prevent false key detection and this operation is called *debouncing*. Debouncing can be accomplished through either hardware or software, but the software approach requires too much processor time for most applications. Multiple key closings are most easily handled by scanning the entire array and inputting the closures in the order they are detected.

9-4-2 Display Design

Various types of devices are available for numeric and alphanumeric displays. Seven-segment LED displays such as the one shown in Fig. 9-31 are typically used for hexadecimal digit display. A digit in seven-segment code is fed to the input pins of a through g and DP. The input can be represented by active low or active high signals, depending on whether the display unit is of a common anode or a common cathode type. An individual segment is lit when the corresponding LED is forward biased.

Figure 9-32 shows a multiple-digit display that is configured from eight seven-segment display units. In order to reduce the device count by eliminating external data latches from the display units, they can be connected to two 8-bit parallel

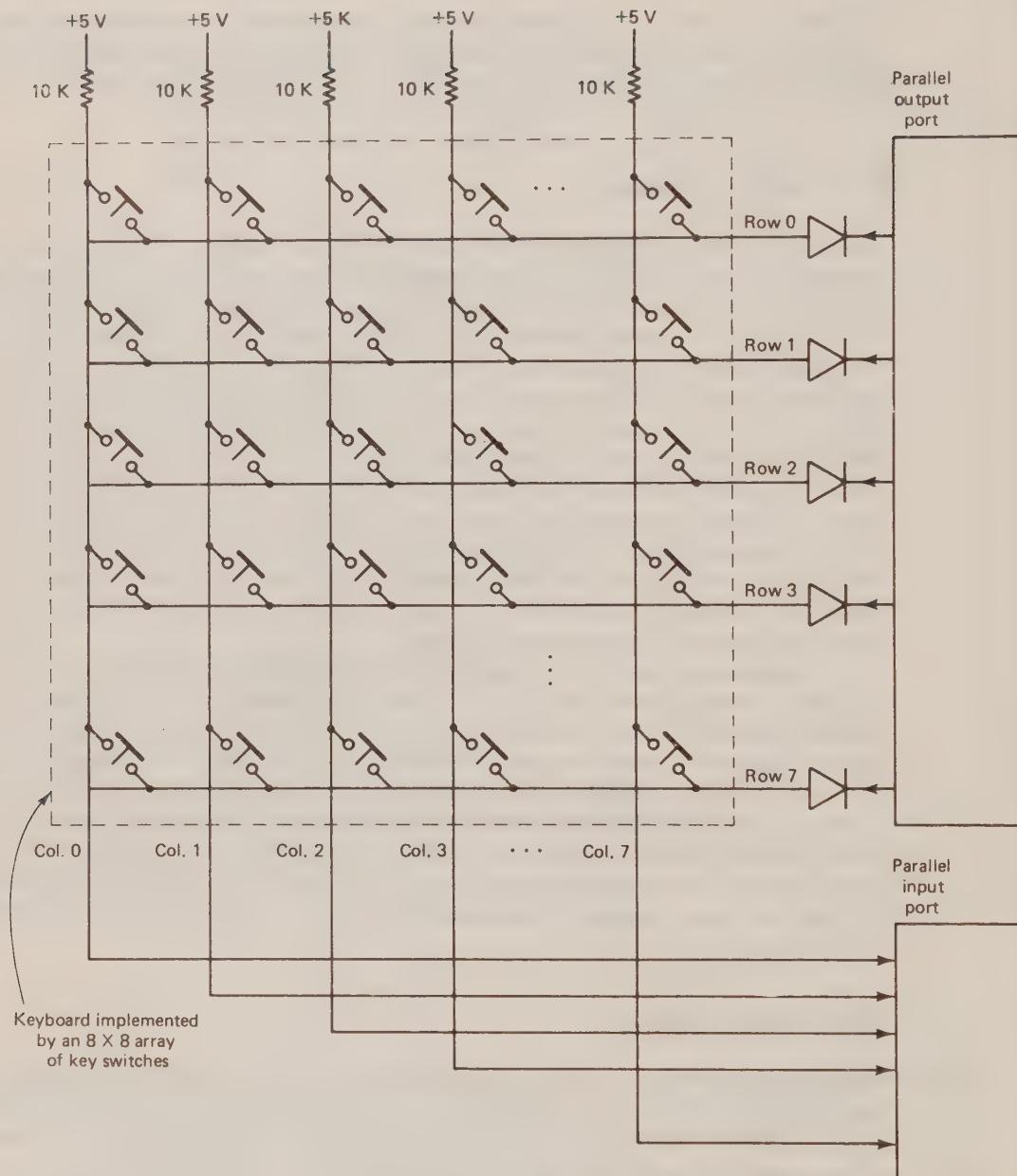
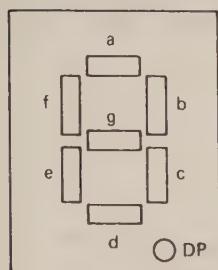
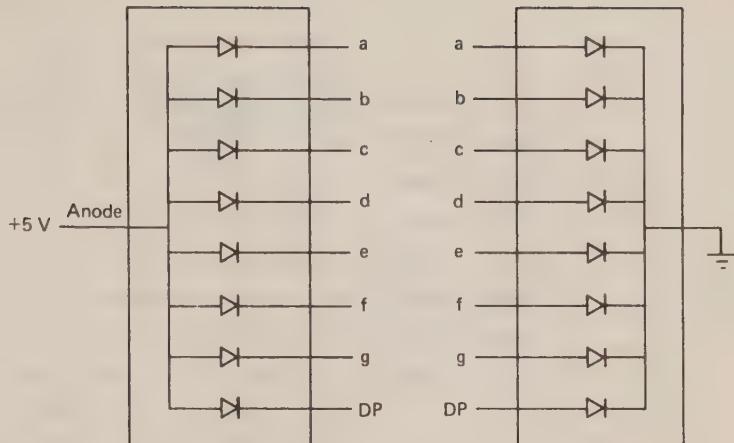


Figure 9-30 Organization of a mechanical keyboard.

output ports and operated in a multiplexed mode. All display units share the common segment lines and only one unit is selected at a time through the digit select lines. Each display is driven for 1 ms and after rotating through all eight units, the first one is returned to and the sequence is repeated. Thus, the displays



(a) Typical 7-segment LED display unit



(b) Common anode

(c) Common cathode

Figure 9-31 Seven-segment LED display unit.

are repeatedly refreshed to give the illusion of a continuously lit multiple-digit display.

Since a typical average current through each turned-on segment could be as high as 20 mA and more than one segment may be lit per digit, the resulting current exceeds the driving capability of a regular TTL gate. Therefore, digit drivers and segment drivers are needed. A low on the base of a digit driver transistor turns the transistor on and effectively connects that display unit to +5 V. A low on the base of a segment driver transistor turns on the transistor, thus letting current flow through the segment. In addition, since the voltage drop across a forward-biased segment diode is constant, a resistor in series with each segment is required to limit the current, thus protecting the display and drivers. The design resistance of these resistors depends on the desired display brightness.

Another type of hexadecimal digit display is the Texas Instruments (TI) TIL311 shown in Fig. 9-33, which uses a matrix-dot array of 20 LEDs. It inputs a 4-bit binary number and internally decodes the digit input to light the LEDs corresponding to the equivalent hexadecimal digit. Since it has built-in latches and constant current drivers, the input is TTL compatible, thus eliminating a need for external drivers and current-limiting resistors. Also, the input latch holds data for constant display and refresh is no longer required.

9-4-3 Keyboard/Display Controller

Although interfacing a keyboard and multiple-digit display to parallel I/O ports is simple from a hardware standpoint, while inputting and outputting the processor would be tied up by keyboard scanning and display refreshing routines. The Intel 8279 keyboard/display controller is an LSI device designed to release the processor from performing the time-consuming scan and refresh operations.

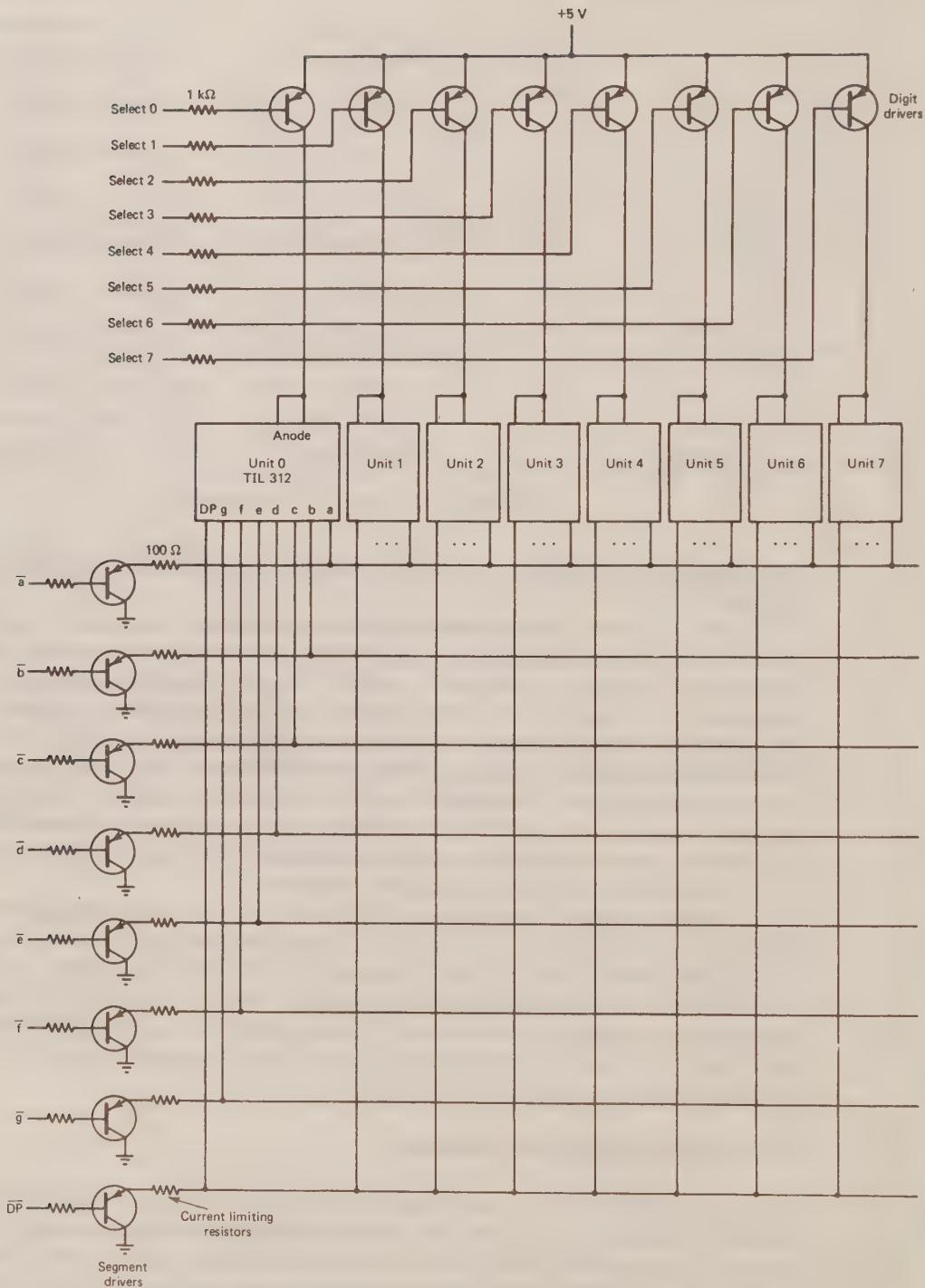


Figure 9-32 Eight-digit display.

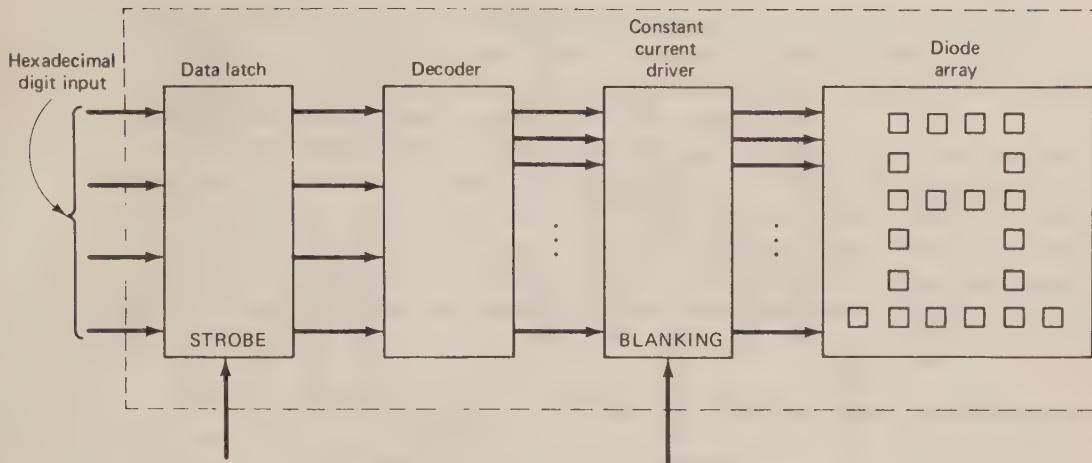


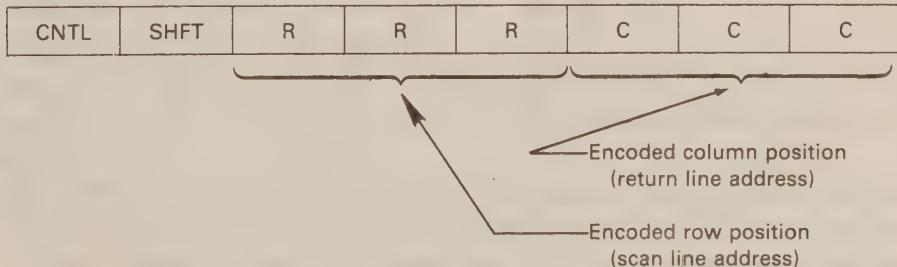
Figure 9-33 TI TIL311 matrix-dot hexadecimal display.

Figure 9-34 shows the general structure of the 8279 and its interface to the bus. The control and status registers share the odd address and the data buffer register uses the even address. The addressing is according to the following table:

| CS | RD | WR | A0 | Transfer Description |
|----|----|----|----|----------------------------------|
| 0 | 1 | 0 | 0 | Data bus to data buffer register |
| 0 | 1 | 0 | 1 | Data bus to control register |
| 0 | 0 | 1 | 0 | Data buffer register to data bus |
| 0 | 0 | 1 | 1 | Status register to data bus |

where 1 means high and 0 means low.

For keyboard control, the 8279 constantly scans each row of the keyboard by sending out row addresses on SL2-SL0 and inputting signals on the return lines RL7-RL0, which represent the column addresses. Note that the SL3-SL0 lines are used for both keyboard scanning and display refreshing and will accommodate up to 16 display units. When a depressed key is detected, the key is automatically debounced by waiting 10 ms to check if the same key remains depressed. If a depressed key is detected an 8-bit code word corresponding to the key position is assembled by combining the encoded column position, row position, shift status, and control status as shown below.



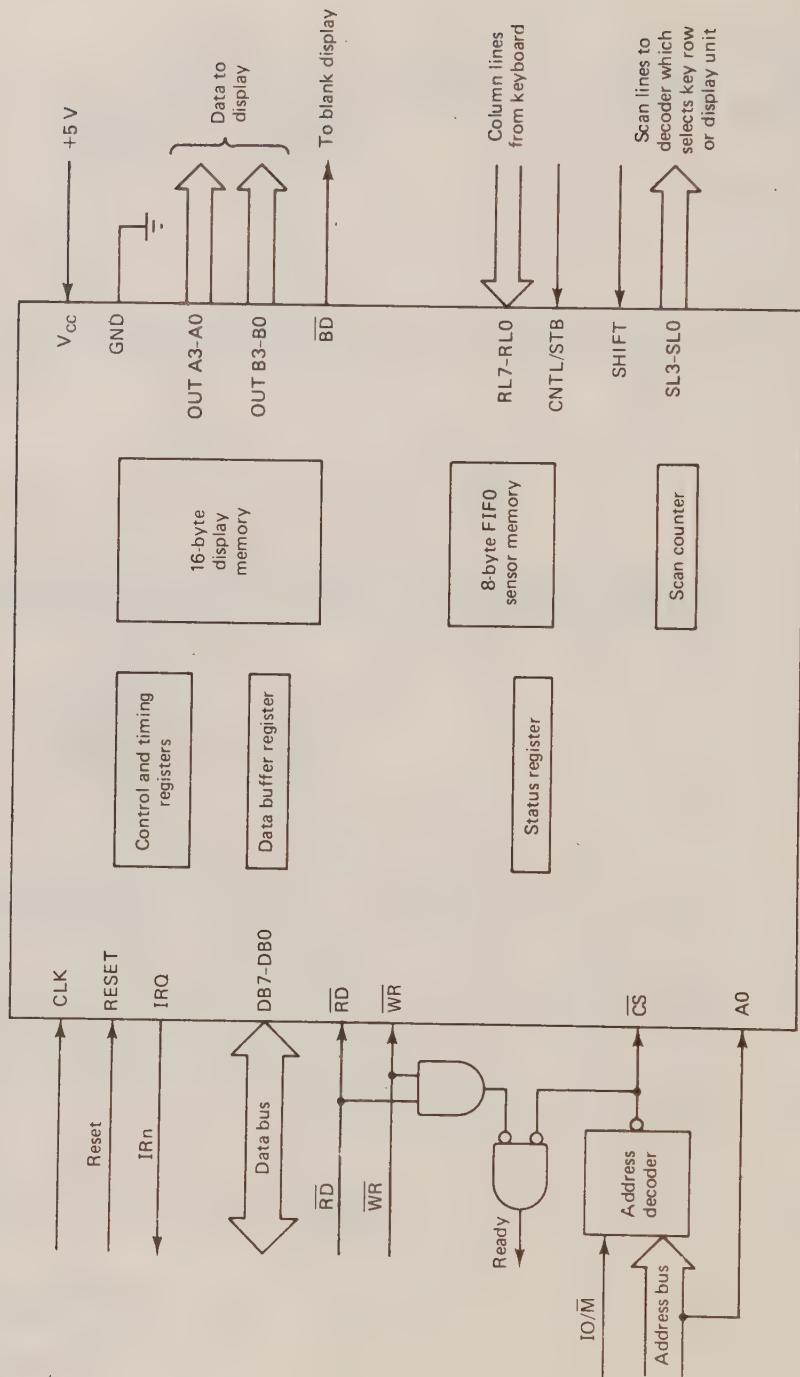
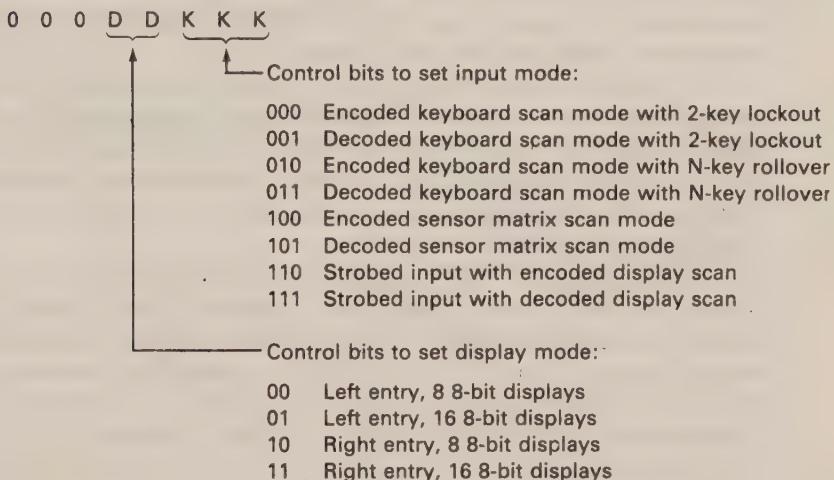


Figure 9-34 Structure of the 8279.

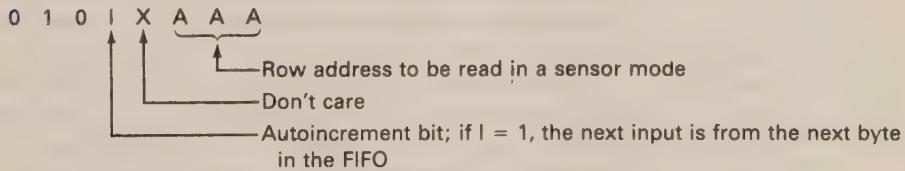
The SHFT and CNTL pins are used primarily to support typewriter-like keyboards which have shift and control keys. The key position is then entered into the 8×8 first-in/first-out (FIFO) sensor memory, and the IRQ (interrupt request) line is activated if the sensor memory was previously empty.

The control and timing registers are physically a collection of flags and registers that are accessed by commands which are sent to the 8279's odd address. The three MSBs of a command determine its type and the meaning of the remaining 5 bits depends on the type. Although there are eight types, only three of them are considered here. (For the other types as well as other detailed information about the 8279, see Reference 2.) The formats of the three commands of interest to us are:

Keyboard Display Mode Set—Specifies the input and display modes and is used to initialize the 8279. Its format is:



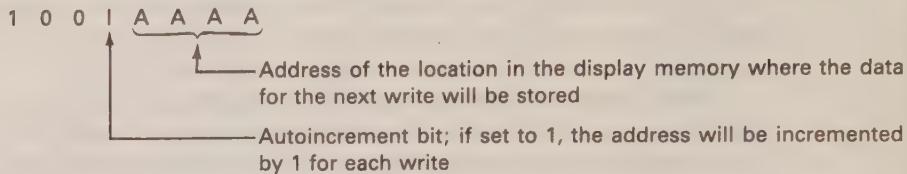
Read FIFO Sensor Memory—Specifies that a read from the data buffer register will input a byte from the FIFO memory and, if the 8279 is in the sensor mode, it indicates which row is to be read. This command is required before inputting data from the FIFO memory. Its format is:



Note that if the input mode is a keyboard scan mode, a read is always from the byte which first entered the FIFO, hence the I and AAA bits are ignored.

Write to Display Memory—Indicates that a write to the data buffer register will put data in the display memory. This command must be given before the

CPU can send the characters to be displayed to the 8279. Its format is:



The 8279 provides two options for handling the situation in which more than one key is depressed at about the same time. With the two-key lockout option, if another key is depressed while the first key is being debounced, the key which is released last will be entered into the FIFO. If the second key is depressed within two scan cycles after the first is debounced and the first key remains depressed after the second one is released, then the first depressed key is recognized. Otherwise, both are ignored. The N-key rollover option treats each key depression independently. If more than one is depressed, after they are depressed they are all entered in the order they were sensed.

In addition, the 8279 has a sensor matrix mode under which signals on the return lines are stored into the FIFO at the row corresponding to the scan address. Unlike a keyboard scan mode, the SHIFT and CNTL status and the scan address are not entered. This mode keeps an image of the status of the sensor matrix and is useful when information from several devices is input by polling each device through the scan lines.

The status of the FIFO is kept in the status register. Bits 0, 1, and 2 of this register give the number of data bytes currently in FIFO memory and bit 3 = 1 indicates that this memory is full. Bits 4 and 5 indicate underflows, which occur when an attempt is made to read from an empty FIFO memory, and overflows, which occur when an input to a full FIFO memory is attempted. In both cases a 1 signifies the presence of the error. A 1 in bit 6 reflects a closure when the 8279 is in its sensor matrix mode and a multiple closure when it is in special error mode. Bit 7 shows whether or not the display is available.

For display control, the 8279 provides a 16-byte display memory and refresh logic. Each address in the display memory corresponds to a display unit, with address 0 representing the leftmost display unit. Once the CPU loads the characters to be displayed into this memory, the 8279 needs no further instructions and the processor is released from refreshing the display units. The output is accomplished by the 8279 repeatedly sending out characters over the lines OUT A3-A0 and OUT B3-B0 and unit select addresses over SL3-SL0. Although the display memory can be directly addressed, the processor may sequentially enter data to the display memory either from the left or from the right. For the autoincrement left entry, after each write to the display the address is incremented by 1, so that the next character appears in the next display unit to the right. The autoincrement right entry allows characters to be displayed in the form used by many electronic calculators. It causes the display to be shifted left one character and stores the next character from the right.

Figure 9-35 illustrates one way in which a 64-key keyboard and an eight-digit

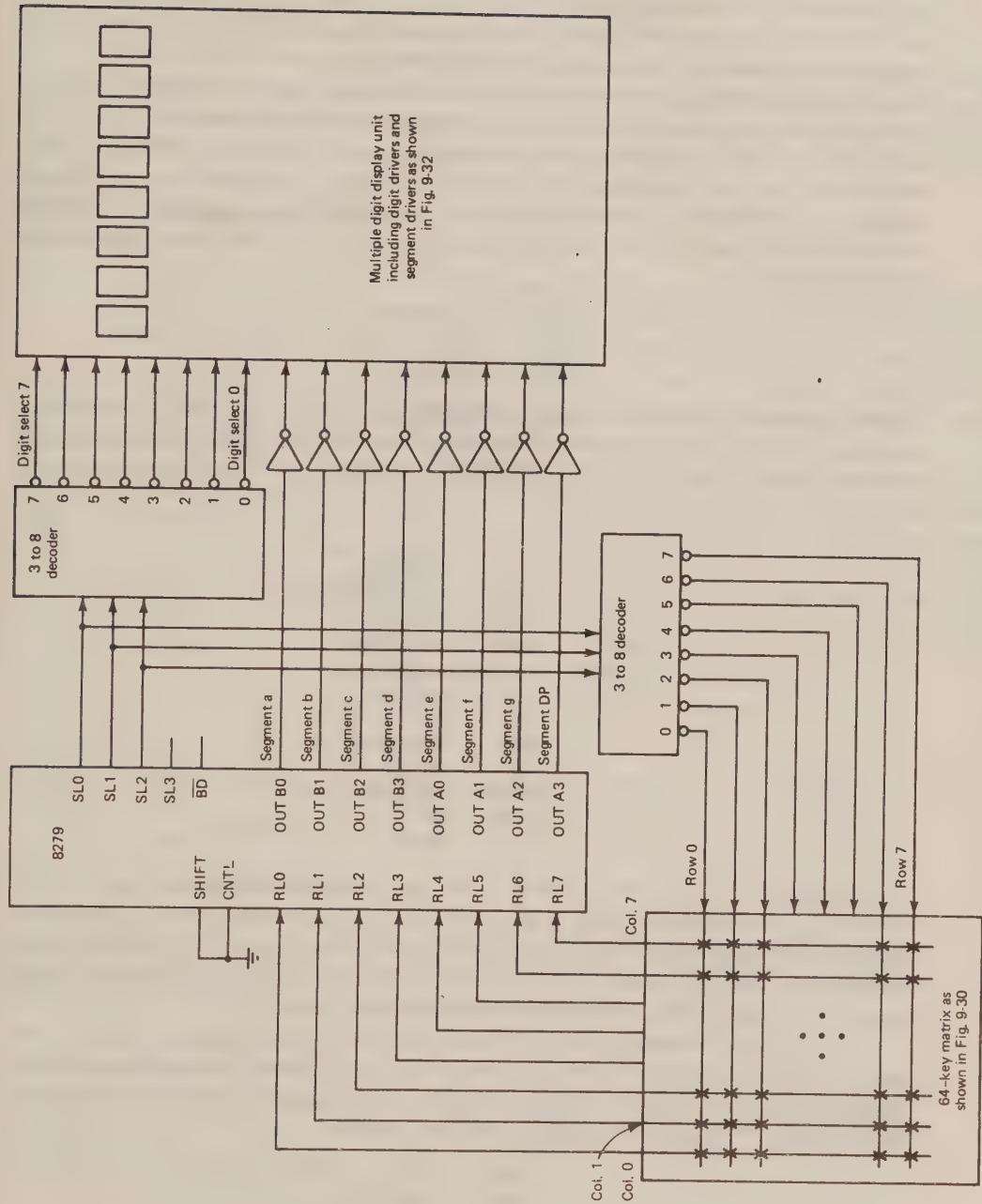


Figure 9-35 Use of an 8279 to interface a keyboard and a multiple-digit display.

seven-segment display can be connected to an 8279. Both the keyboard and display are scanned and refreshed under the control of the select signals SL2–SL0. The SL3 pin is not connected because there are only eight display units. Both 3-to-8 decoders have active low outputs. One decoder selects one row of keys while the other decoder enables one of the eight digit drivers.

To demonstrate how to program an 8279, let us assume that the device is connected to a keyboard and multiple-digit display as shown in Fig. 9-35, the 8279's addresses are FFE8 and FFE9, and the interrupt request pin IRQ is not used. First, the device must be initialized by sending a mode set command to the control register. The following instructions set the keyboard/display controller to its encoded keyboard scan mode, with two-key lockout, and its left entry eight 8-bit displays mode:

```
MOV DX,0FFE9H
MOV AL,0
OUT DX,AL
```

Then, characters generated by the depressed keys can be read through the FIFO memory. A program segment that uses programmed I/O to input eight keywords and store them in an 8-byte array KEYS with the first byte at the highest address is:

```
MOV SI,8
MOV DX,0FFE9H
MOV AL,01000000B
OUT DX,AL
NEXT: MOV DX,0FFE9H
IDLE: IN AL,DX
TEST AL,0FH
JZ IDLE
MOV DX,0FFE8H
IN AL,DX
MOV KEYS[SI - 1],AL
DEC SI
JNZ NEXT
```

The first instruction puts the count in SI, the next three instructions cause the input from the even address to come from the FIFO, the three instructions beginning at IDLE force the processor to idle until an input is ready, and the following three instructions transfer the input data to KEYS. The last two instructions cause the sequence to repeat until eight characters have been read.

To display characters, the CPU must first give a write display memory command and then output to the display memory. The following instruction sequence displays eight seven-segment digits which are stored beginning at DIGITS with the least significant digit being stored at the low address:

```
MOV SI,8
MOV DX,0FFE9H
MOV AL,10010000B
OUT DX,AL
MOV DX,0FFE8H
```

```

AGAIN:    MOV    AL,DIGITS[SI-1]
          OUT    DX,AL
          DEC    SI
          JNZ    AGAIN

```

The first instruction puts the digit count in SI, the following three instructions output the write display memory command, and the next instruction puts the address of the data buffer register in DX. The loop outputs the digits to the display memory.

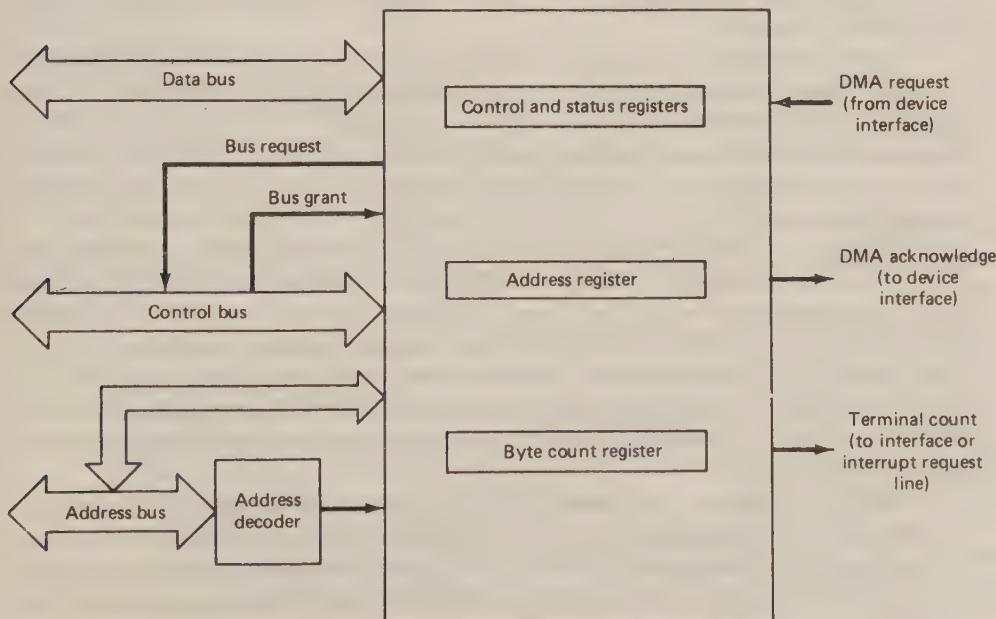
9-5 DMA CONTROLLERS

As discussed in Chap. 6, a DMA controller is capable of becoming the bus master and supervising a transfer between an I/O or mass storage interface and memory. While making a transfer, it must be able to place memory addresses on the bus and send and receive handshaking signals in a manner similar to that of the bus control logic. The purpose of a DMA controller is to perform a sequence of transfers (i.e., a block transfer) by stealing bus cycles.

A DMA controller is designed to service one or more I/O or mass storage interfaces, and each interface is connected to the controller by a set of conductors. A portion of a DMA controller for servicing a single interface is called a *channel*.

The general organization of a one-channel DMA controller and its principal connections is shown in Fig. 9-36. In addition to the usual control and status registers, each channel must contain an address register and a byte (or word) count

Figure 9-36 General organization of a DMA controller.



register. Initializing the controller consists of filling these registers with the beginning (or ending) address of the memory array that is to be used as a buffer and the number of bytes (words) to be transferred. For an input to memory, each time the interface has data to transfer it makes a DMA request. The controller then makes a bus request and, when it receives a bus grant, it puts the contents of the address register on the address bus, sends an acknowledgment back to the interface, and issues I/O read and memory write signals. The interface then puts the data on the data bus and drops its request. When the memory accepts the data it returns a ready signal to the controller, which then increments (or decrements) the address register, decrements the byte (word) count, and drops its bus request. Upon the count reaching zero, the process stops and a signal is sent to the processor as an interrupt request or to the interface to notify it that the transfers have terminated. An output is similarly executed, except that the controller issues I/O write and memory read signals and the data are transferred in the other direction.

As an example, let us consider the Intel 8237 DMA controller, whose overall organization along with that of the associated logic needed in an 8088 system is given in Fig. 9-37. A minimum mode 8088 configuration containing an 8237 is shown in Fig. 9-38.

When data are being put in or taken out of the 8237's registers, the 8237 is a slave just like the system's I/O interfaces. As a slave it receives 16-bit addresses with the 12 MSBs of these addresses determining whether or not the chip is selected and the 4 LSBs being used for internal addressing. When both HRQ and CS are low, the 8237 becomes a slave with the IOR and IOW being the input control pins. The CPU can read from or write to the internal registers of the controller by activating IOR or IOW. The AEN, which is active when the controller is a master and is outputting an address, is 0 while the system is communicating with the controller's registers.

If the controller is the master, then it must supply the bus address. When it is master it puts the low-order byte of the address on the pins A7-A0 and the high-order byte on DB7-DB0, and sets AEN to 1. With AEN = 1, the outputs of the external address latch are enabled, thereby allowing the upper address byte to be put on the A15-A8 lines. The AEN line is also used to disable the 8282 address latches connected to the CPU's A19-A8 and AD7-AD0 pins. Shortly after the AEN signal is activated a pulse is sent out over the Address Strobe (ADSTB) pin. This pulse is used to strobe bits 8 through 15 of the address into the 8282 address latch connected to A15-A8. The upper 4 bits of the address, A19-A16, cannot be obtained from the 8237, but must be programmed separately before the block transfer begins. A 4-bit I/O port, which can be output to just like any other I/O port, is needed to hold the high-order 4 bits of the address. During a block transfer, the contents of this port should not be changed; therefore, any single transfer is limited to 2^{16} bytes.

While it is master the controller must also output the necessary read/write commands. These commands are IOR, MEMR, IOW, and MEMW and indicate an I/O read, a memory read, an I/O write, and a memory write, respectively. Because these signals do not match the RD, WR, and IO/M signals output by a

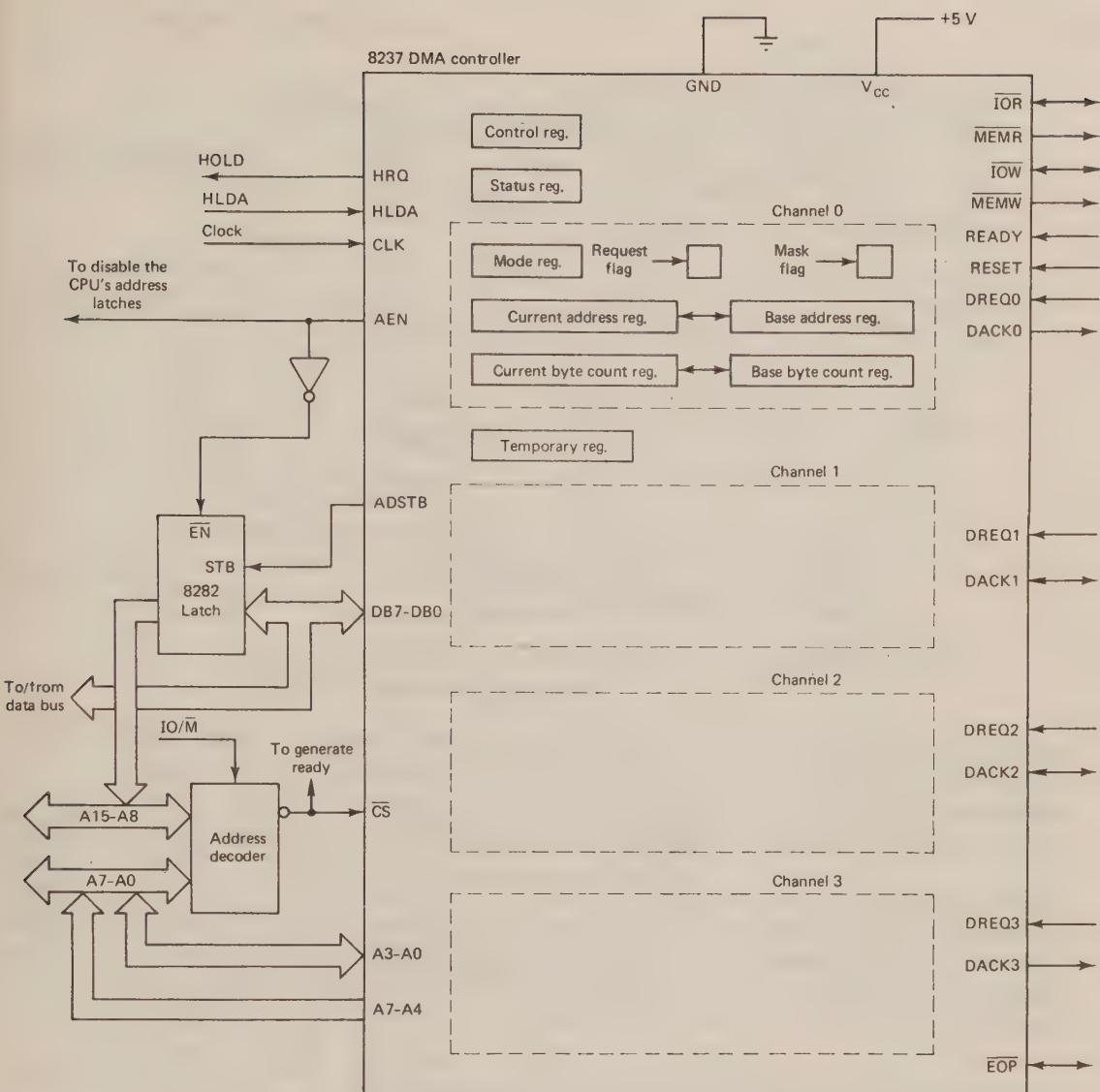


Figure 9-37 Organization of an 8237 and its associated logic.

minimum mode-8088, a read/write encoding circuit such as the one shown in Fig. 9-39 is needed to perform the translation between the two sets of signals. During a DMA transfer, the controller disables the outputs of the read/write encoder from the command lines by activating the AEN signal. Of course, memory and I/O interfaces in the system must be designed so that they respond to MEMR, MEMW, IOR, and IOW instead of RD, WR, and IO/M. As with the processor timing, the READY signal is used to extend the bus cycles by inserting wait states when servicing slow devices. A RESET signal clears the control, status, and temporary registers and the request flags, and sets the mask flags. The EOP pin is bidirectional. The count going to zero causes a negative pulse to be output through EOP, which

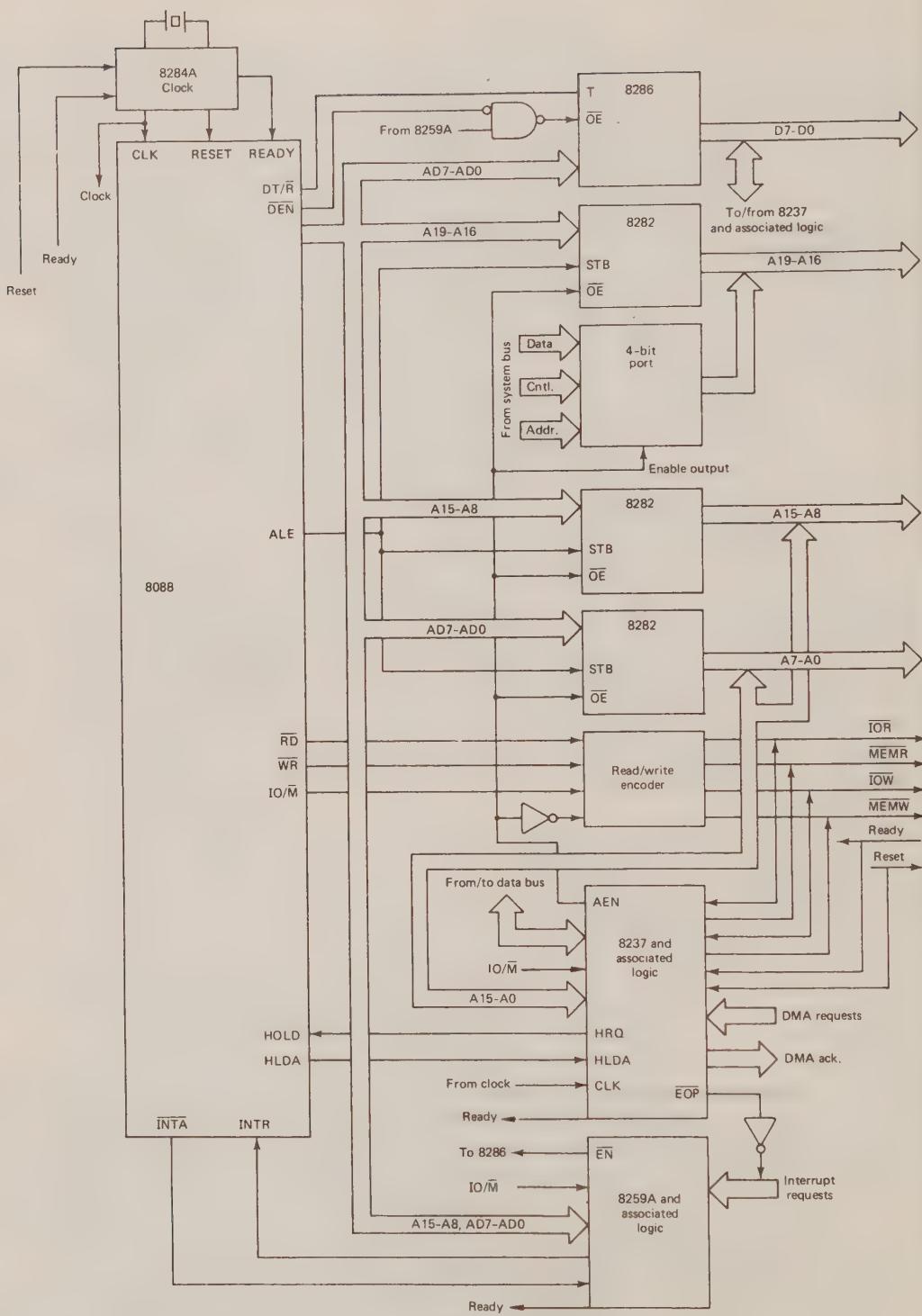
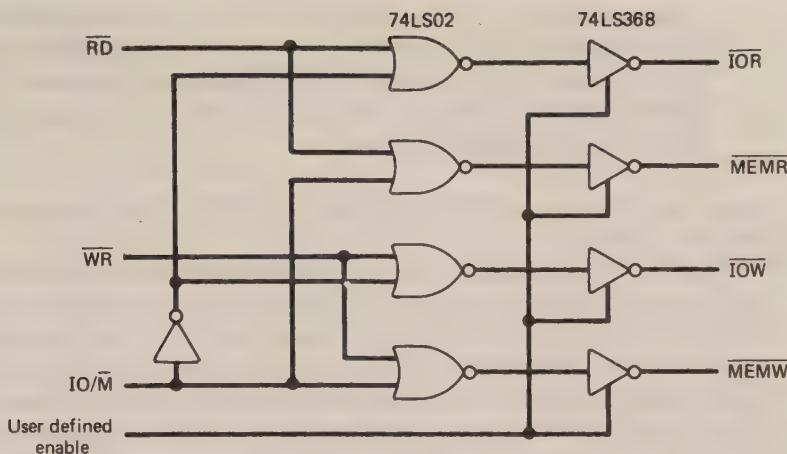


Figure 9-38 Minimum mode 8088 configuration that includes an 8237.



Note: If it is not necessary to three-state the command lines, a decoder (8205 or 74S138) could be used. The 74LS257 is not recommended since the outputs may experience voltage spikes when entering or leaving three state.

Figure 9-39 Encode logic for \overline{RD} , \overline{WR} , and $\overline{IO/M}$ signals. (Reprinted by permission of Intel Corporation. Copyright 1979.)

can be used either as an interrupt request or by the device interface. On the other hand, the interface may send an active low signal to the 8237 through this pin. In either case, a low on the \overline{EOP} will result in the block transfer being suspended. After the suspension the block transfer may resume according to the mode of the 8237.

An 8237 includes control, status, and temporary registers, and four channels, each containing a mode register, current address register, base address register, current byte count register, base byte count register, request flag, and mask flag. Each channel may be put in one of four modes, with its current mode being determined by bits 7 and 6 of the channel's mode register. The four possible modes are:

Single Transfer Mode (01)—After each transfer the controller will release the bus to the processor for at least one bus cycle, but will immediately begin testing for DREQ inputs and proceed to steal another cycle as soon as a DREQ line becomes active.

Block Transfer Mode (10)—DREQ need only be active until DACK becomes active, after which the bus is not released until the entire block of data has been transferred.

Demand Transfer Mode (00)—This mode is similar to the block mode except that DREQ is tested after each transfer. If DREQ is inactive, transfers are suspended until DREQ once again becomes active, at which time the block transfer continues from the point at which it was suspended. This allows the interface to stop the transfer in the event that its device cannot keep up.

Cascade Mode (11)—In this mode 8237s may be cascaded so that more than four channels can be included in the DMA subsystem. In cascading the controllers, those in the second level are connected to those in the first level by joining HRQ to DREQ and HLDA to DACK. To conserve space, this mode will not be considered further.

In all cases the count going to zero will cause $\overline{\text{EOP}}$ to become active and the transfer process to cease.

Bit 5 of the mode register specifies whether the contents of the address register are to be incremented (0) or decremented (1) after each transfer, thus determining the order in which the data are stored in memory. If bit 4 is 1, then autoinitialization is enabled. When the current address and current byte count registers are initially loaded their contents are also put in the base address and base byte count registers. If autoinitialization is enabled, then the current registers are automatically reloaded from the base registers whenever the count goes to zero. Bits 2 and 3 indicate the type of transfer to be made. There are three types: verify (00), write (01), and read (10). The verify type is for verifying information concerning the previous input or output operation and is not actually associated with a current transfer. It is of little interest to us and will not be discussed further. The two LSBs of an output to a mode register direct the output to the indicated channel, i.e., select the mode register that is to receive the output.

In addition to block transfers between I/O or mass storage devices and memory, the 8237 can supervise memory-to-memory transfers. Such transfers are conducted by bringing bytes from the source memory area into the 8-bit temporary register in the 8237 and then outputting them to the destination memory area. Therefore, two bus cycles are required for each memory-to-memory transfer. The channel 0 current address register is used for source addressing. The channel 1 current address and current byte count registers provide the destination addressing and counting. The destination address must increment or decrement as usual, but by setting the proper bit in the control register the source address may be held constant. This allows the same data byte to be transferred into the entire destination array.

With regards to the control register, a memory-to-memory transfer is enabled by setting bit 0 to 1, in which case bit 1 = 1 indicates that the source address is to be held constant. Bit 2 is used for enabling (0) or disabling (1) the controller and bit 3 specifies the type of timing. If the speed characteristics of the system permit, then bit 3 can be set to 1 to indicate compressed timing. With compressed timing only two clock cycles are needed to perform most transfers. (Timing is discussed below.) Bit 4 determines whether the priority is fixed or rotating. Normally, channel 0 has the highest priority and channel 3 the lowest; however, if bit 4 is 1, the priority will rotate after each transfer, e.g., if the priority before a transfer is 2-3-0-1, then after the transfer it will be 3-0-1-2. By rotating the priority the controller can prevent one channel from dominating the bus. The 8237 permits the program to specify the length of the $\overline{\text{IOW}}$ and $\overline{\text{MEMW}}$ signals when normal

timing is being used. A 1 in bit 5 indicates that these signals are to be extended over two clock cycles. The program can also specify whether the DREQ and DACK pins are to be active high or active low by setting or clearing bits 6 (DREQ) and 7 (DACK), respectively. Bit 6 = 1 indicates that DREQ is active low and bit 7 = 1 indicates that DACK is active high. How these bits should be set depends on the characteristics of the associated interfaces.

The format of the status register is such that the lower 4 bits indicate the states of the terminal counts of the four channels and the upper 4 bits show the current presence or absence of DMA requests. For the lower 4 bits a 1 in bit n indicates that the terminal count for channel n is active, and for the upper four bits a 1 in bit $n + 4$ signals the presence of a request on channel n .

Each channel also has associated with it a request flag and a mask flag. A DMA request can be programmed as well as input through the DREQ pin. Setting the request flag for a channel has the same effect as the DREQ pin becoming active, and it is cleared when \overline{EOP} goes active. When set to 1, the mask flag disables the channel so that DMA requests (either hardware or software) are not recognized. If a channel is not programmed for auto-initialization, this flag is automatically set by EOP going active. The request and mask flags are programmed using commands in which bit 2 determines the setting of the flag and bits 1 and 0 give the channel number of the flag. The remaining bits are unused. There is also a command for adjusting all four of the mask flags at once. For this command, bit n being set or cleared causes the mask bit for channel n to be set or cleared.

Besides the commands for setting the flags, there are a master clear command and a clear first/last flip-flop command. A master clear command has the same effect as a RESET signal. The first/last flip-flop is for loading the 16-bit address and count registers. Because an 8237 can be sent only one byte at a time, these registers must be loaded using two outputs. Assuming that the first/last flip-flop is initially 0, the first byte output to one of these registers is put in the low-order byte and the flip-flop is set to 1. When the second byte is sent, the flip-flop being 1 directs it to the high-order byte of the register. Then the flip-flop is reset to 0. The purpose of the clear first/last flip-flop command is to initialize this flip-flop before outputting to the address and count registers. Neither of the two commands discussed in this paragraph involves the transfer of data over the data bus. They are automatically executed by the controller when a write is made to the appropriate address.

The addressing of the various registers and commands associated with the controller is done via the \overline{CS} , \overline{IOR} , \overline{IOW} , and A3–A0 lines. $\overline{CS} = 0$, of course, indicates that the controller is being accessed. A3 equals 0 when an address or count register is addressed and is 1 when the control or status register is accessed or a command is being given. When addressing an address or count register A2–A1 gives the channel number and A0 = 0 specifies the current address register and A0 = 1 the current count register. For a read, IOR is low and IOW is high, and for a write, IOR is high and IOW is low. A write to either the current address or current count register will also write to the associated base register. Addressing

the control and status registers and giving commands are summarized as follows:

| A3 | A2 | A1 | A0 | IOR | IOW | Transfer or Command |
|----|----|----|----|-----|-----|----------------------------|
| 1 | 0 | 0 | 0 | 0 | 1 | Read from status register |
| 1 | 0 | 0 | 0 | 1 | 0 | Write to control register |
| 1 | 0 | 0 | 1 | 1 | 0 | Write to a request flag |
| 1 | 0 | 1 | 0 | 1 | 0 | Write to a mask flag |
| 1 | 0 | 1 | 1 | 1 | 0 | Write to mode register |
| 1 | 1 | 0 | 0 | 1 | 0 | Clear first/last flip-flop |
| 1 | 1 | 0 | 1 | 0 | 1 | Read temporary register |
| 1 | 1 | 0 | 1 | 1 | 0 | Master clear |
| 1 | 1 | 1 | 0 | 1 | 0 | Clear mask flags |
| 1 | 1 | 1 | 1 | 1 | 0 | Write to all mask flags |

where 0 means low and 1 means high. (Note that the lowest port address assigned to an 8237 must be divisible by 16_{10} .) The temporary register can be read only after the completion of the entire memory-to-memory block transfer. All combinations not shown are illegal.

The 8237 timing can be divided into the SI, S0, S1, S2, S3, S4, and SW states. A flowchart of the timing as seen by the 8237 is given in Fig. 9-40. Between transfers the controller is in a succession of idle, SI, states. During each SI state the DREQ lines are tested to determine if a channel is requesting a DMA transfer. If $\overline{CS} = 0$ and all of the DREQs are inactive, the controller becomes a slave and the CPU can communicate with it. If there is an active DREQ input, HRQ is activated and an S0 state is entered. S0 states are repeated until a HLDA signal is returned, and then a sequence of the S1 through S4 states occurs. S1 may be skipped if the high-order byte of the transfer address does not need to be changed. When a transfer is made to or from a device that cannot respond quickly enough, wait (SW) states are inserted. If normal timing is used, S3 states are needed, but if the bus and its interfaces can handle compressed timing, the S3 states are deleted. During S4 the transfer mode is tested and, except for incomplete block transfers in the block and demand modes, a return is made to SI. A timing diagram for interface to memory transfers is shown in Fig. 9-41.

9-6 DISKETTE CONTROLLERS

The primary means of storing large quantities of data are magnetic tape and disk units, although magnetic bubble memory (MBM) is currently having some impact in this area. Magnetic tapes and disks have two major advantages over MBMs: portability between systems and capacity. Magnetic tapes tend to cost less per byte of stored information and be the most durable, but disks have much lower access times. Tapes and disks come in many sizes and shapes and the designs of the equipment for reading from and writing onto them vary correspondingly. It is not possible to give a reasonable presentation of the numerous types of tape and disk units in the space available here; therefore, we will concentrate on what is by far the most popular mass storage units in microcomputer systems, the *diskette*, or *floppy disk*, units.

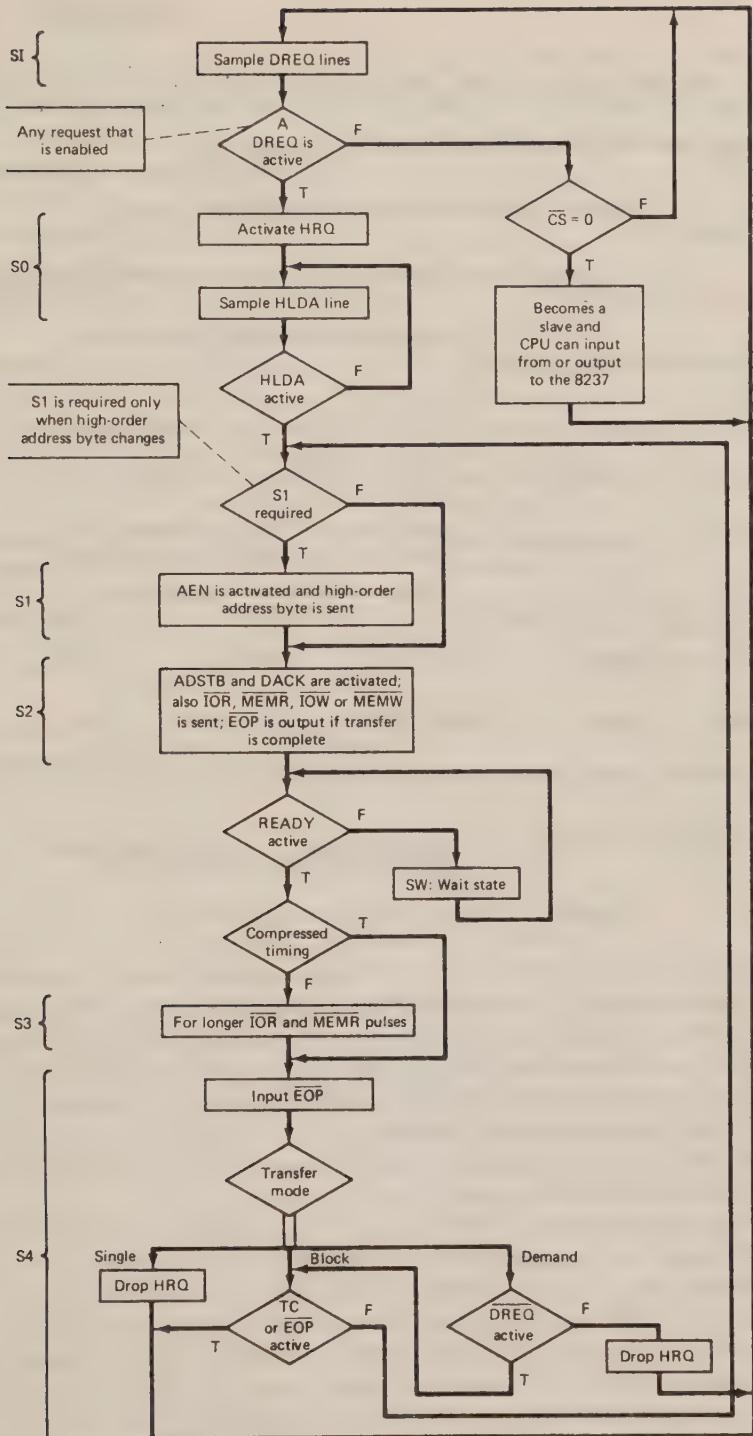


Figure 9-40 Flowchart of 8237 activity by states.

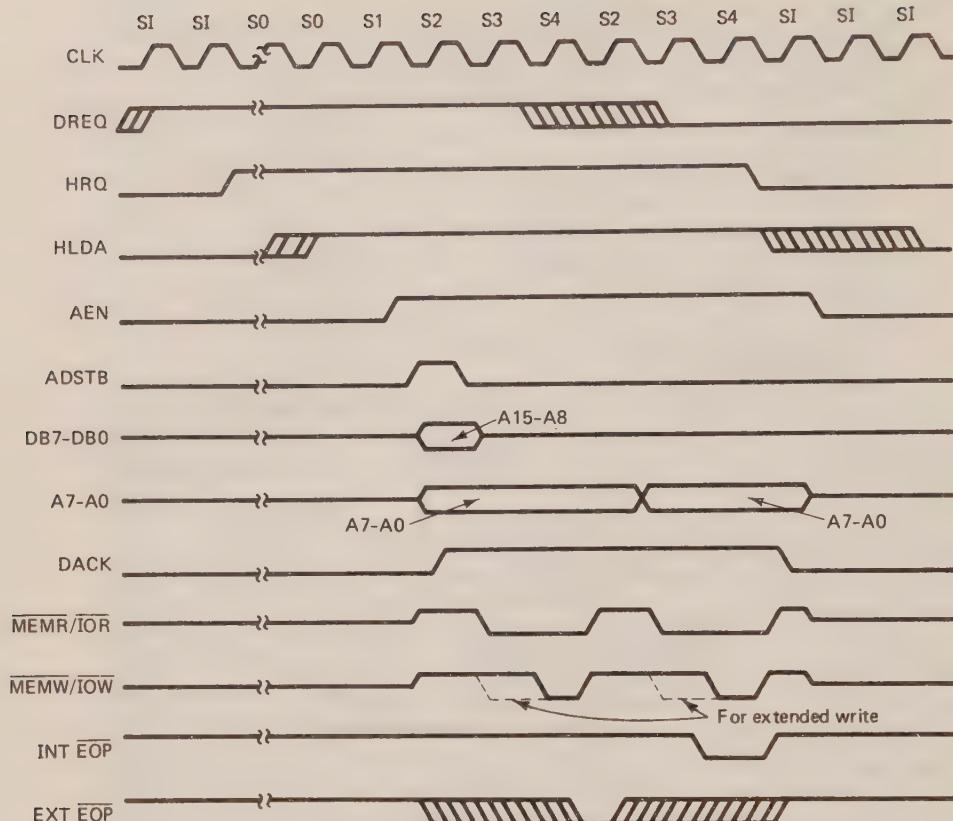


Figure 9-41 Typical timing diagram for an 8237.

The physical construction and overall data storage format of a diskette is shown in Fig. 9-42. The diskette is a Mylar disk that is coated with a magnetic substance. It has a spindle hole in the center and at least one index hole that is slightly offset from the center. The purpose of the index hole(s) is to give the diskette drive a reference point(s) while the drive is storing or retrieving data. The diskette is sealed in a square jacket that also has a spindle hole and index hole. In addition, the jacket has a slot so that the drive's read/write head can contact the diskette's surface, and a write protect notch.

The data are bit serially stored (i.e., as a succession of bits) in concentric circles called *tracks* and are grouped into arcs known as *sectors*. Some diskette drives have only one read/write head and can only store and retrieve data from one surface of the diskette, while others have two read/write heads and can utilize both surfaces. If both surfaces can be accessed, then the pairs of tracks that are the same distance from the center of the diskette are referred to as *cylinders*.

There are two standard sizes for diskettes, ones with 8-in. jackets and ones with 5.25-in. jackets (called *mini-diskettes*). Some have only one index hole and are said to be *soft sectored*, while others have an index hole for each sector and

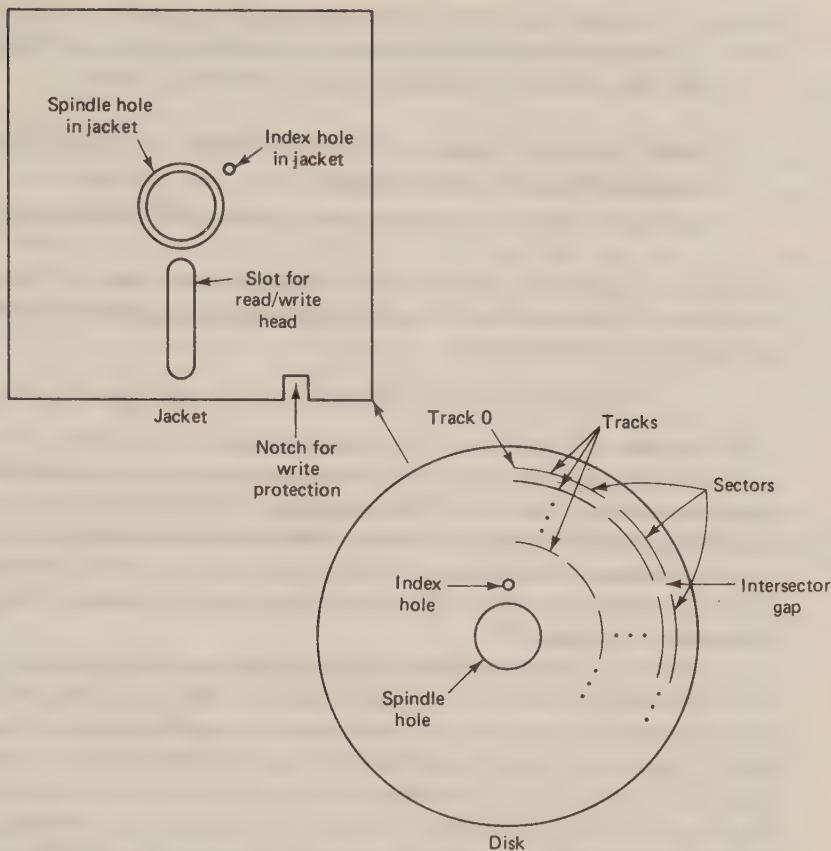


Figure 9-42 Construction of a diskette.

are said to be *hard sectored*. Each sector on a soft-sectored diskette must start with formatting information that marks the beginning of the sector and only the first sector of a track is marked by an index hole. For a hard-sectored diskette the beginning of all sectors are marked by index holes.

The tracks (and cylinders) are numbered, with the outermost track being given the number 0. The sectors are also numbered and on a soft-sectored diskette, the first sector after the index hole is assigned the number 1. When a diskette is inserted in a drive, the spindle passes through the spindle hole and the diskette is clamped to the spindle and is continuously rotated within its jacket at 360 rpm. The read/write head moves in and out so that it can be positioned over any desired track. Once positioned, the rotational motion causes the sectors within the selected track to pass under the head, thus permitting the individual sectors to be accessed. When a read or write is to be performed, the head is lowered, or *loaded*, onto the surface of the diskette through the slot in the jacket, and is then positioned over the desired track. For a soft-sectored diskette, the drive waits until the index hole is encountered and then begins to read the sector formatting information. Upon

finding the needed sector the drive may begin its read or write operation. In the case of a hard-sectored disk the sector is found by means of the index holes.

The time needed to access a sector is subdivided into:

Load Time—For bringing the head in contact with the diskette.

Position Time—For positioning the head over the track.

Rotational Time—For rotating the diskette until it is over the desired sector.

Typical average load, position, and rotational times are 16, 225, and 80 ms, respectively. Once a sector is found the average information transfer rate in bytes per second is approximately

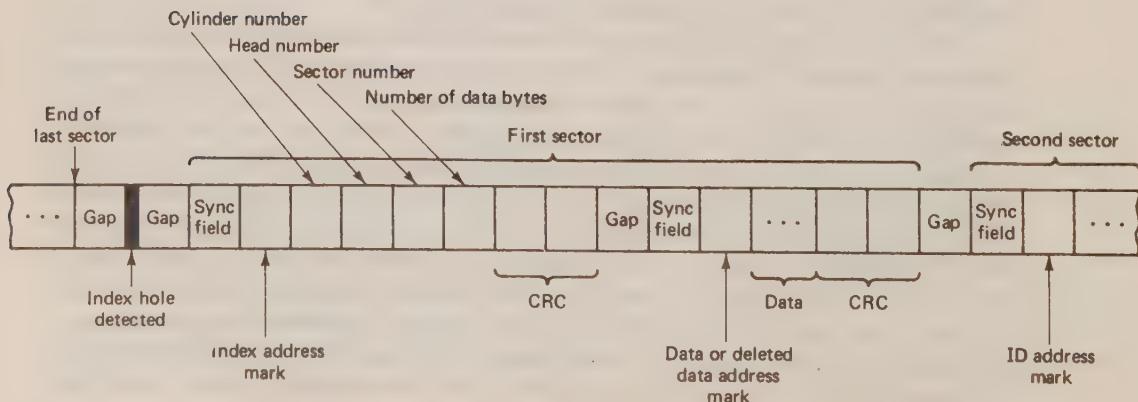
$$\text{Bytes per sector} \times \text{Sectors per track} \times \text{Speed in rpm}/60$$

(This includes the time wasted in traversing gaps in the data.)

In order to make our discussion more specific, let us now limit it to a single type of diskette, the IBM 3740-compatible, soft-sectored, 8-in. diskette. These diskettes contain 77 tracks and either 15 sectors (single density) or 26 sectors (double density), although our examples will permit from 8 to 26 sectors. Normally, only 75 of the tracks are used, thus allowing for two bad tracks. The good tracks are numbered 0 through 74.

The track and sector format of these diskettes is illustrated in Fig. 9-43. There is a gap between each of the sectors and the index hole indicator must become active during the gap between the last sector and first sector of each track. Each sector contains an identification field and a data field, both of which begin with a sync field. There is a gap between the two fields which gives the head time to be turned on or off, depending on whether the previous or following data field was or is to be accessed. The identification field contains an address mark followed by the cylinder (or track) number, head number, sector number, number of data bytes

Figure 9-43 Track and sector format for a soft-sectored diskette.



in the sector, and two error detection bytes. The data field consists of an address mark followed by the data bytes and two error detection bytes.

The address mark in the identification field of the first sector is usually different from those in the other sectors and the address marks in the identification fields are different from those in the data fields. Also, the address marks for the data field differ depending on whether the field contains useful information or is occupied by filler bytes.

There are two principal methods for generating the error detection bytes. One is to obtain a *checksum* by summing all of the bytes in the field that occur before the checksum (and ignoring any overflow). The other is to consider the bits in the field as coefficients of one big binary polynomial and let the error detection bytes be the remainder, called the *cyclic redundancy check (CRC)*, that results from dividing this polynomial by a fixed polynomial of degree 16. It is the latter method that will be assumed here.

The bit pattern of the serially stored information is shown in Fig. 9-44. The information is grouped into cells, each of which is divided into four intervals, with the first interval containing a clock pulse. The third interval is for indicating a data bit and will contain a pulse if the data bit is 1. A representative value for the cell period for a 360-rpm drive is 4 μ s.

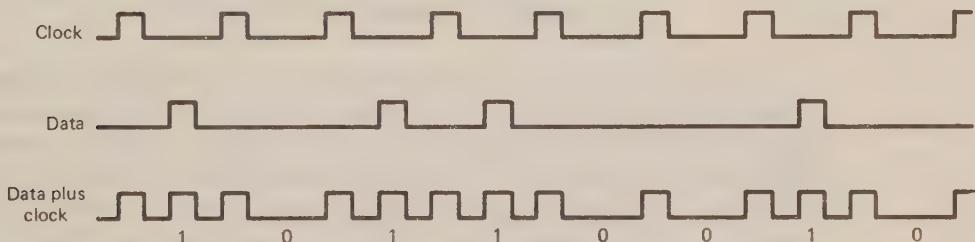


Figure 9-44 Bit pattern of stored data.

Figure 9-45 gives the organization of a typical diskette controller and Fig. 9-46 shows how the controller would appear in a diskette subsystem. Because the information is serially stored on the diskette the inputs from and outputs to the diskette drive are bit streams. Therefore, the input bytes are received by a shift register before they are transferred to the data buffer register, and the output bytes are put in a shift register so that they can be shifted out in serial form. The output electronics must be such that the pulses from the write clock input are interspersed with the data bits to produce the format given in Fig. 9-44.

Because most disk controllers are designed to service more than one drive, some of the pins on the controller must be reserved for drive selection lines. Other control outputs to the drive should include a step pulse line for stepping through the tracks, a step direction signal, and a head load signal. There should be control inputs for inputting signals indicating when the head is positioned at track 0, a fault or error has occurred, or the diskette is write protected, and a pulse each time the index hole is sensed.

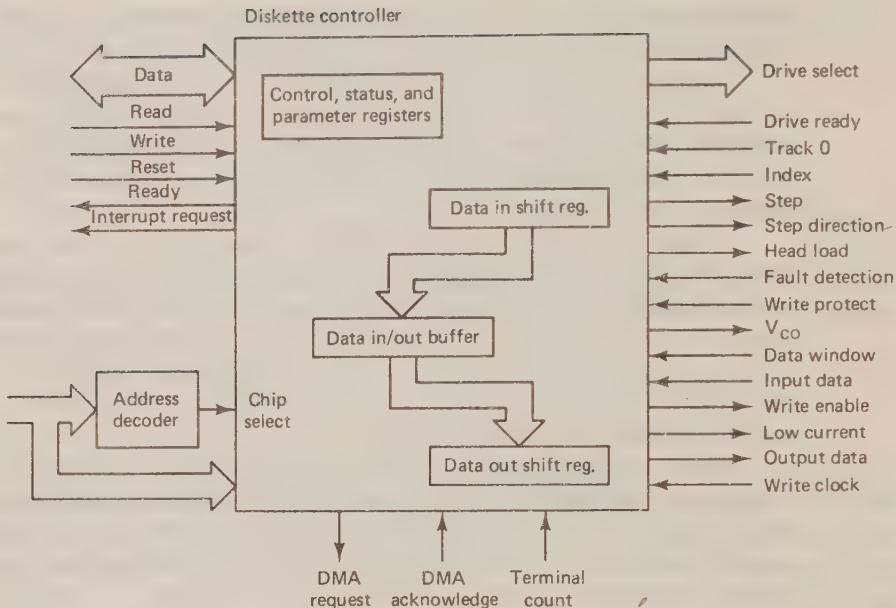


Figure 9-45 Organization of a diskette controller.

For reading data from the disk the data must be separated from the clock pulses. This is done by using a phase-locked-loop circuit to provide a data window signal. The controller must supply a V_{CO} sync signal to the phase-locked loop. For writing data there should be a write enable and low-current lines in addition to the output data line. The low-current line is needed because less current is used when writing to the inner tracks, those tracks with numbers greater than 43.

If a diskette having 26 sectors and 128 data bytes per sector is rotated at 360 rpm, the average transfer rate is approximately

$$26 \times 128 \times 360/60 = 19,968 \text{ bytes/second}$$

which gives an average period of about 50 μs . Taking into account the gaps, the actual period for transferring a byte is normally 32 μs (or 4 μs per bit). Although it would be possible to perform transfers at this rate without a DMA controller, it is much more efficient to use one. Consequently, disk controllers normally have DMA request, DMA acknowledge, and terminal count pins. Another reason for using a DMA controller with a disk controller is that disk transfers always involve blocks of data, not single bytes.

As an example of a diskette controller let us consider the Intel 8272, whose organization is diagrammed in Fig. 9-47. It has two registers that can be addressed by the processor, a status register that is read when $A_0 = 0$ and a data in/out register that is accessed when $A_0 = 1$. The format of the status register is given in Fig. 9-48. There are several other control, status, and parameter registers and flags that can be addressed via the data in/out register. Whether these registers

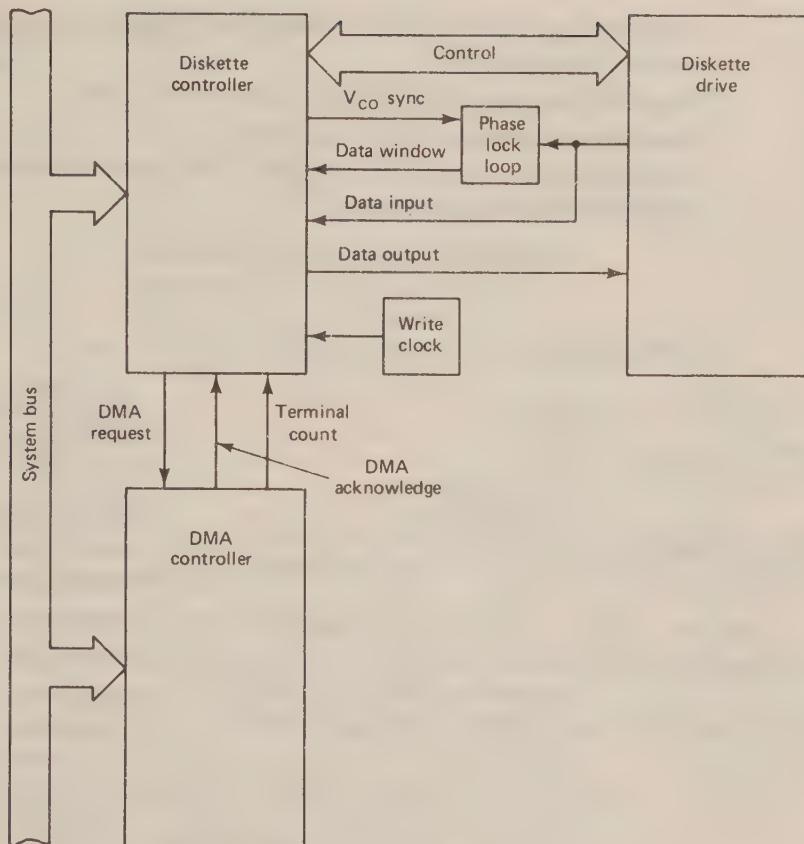


Figure 9-46 Diskette subsystem.

and flags are being accessed or data I/O is being performed with the diskette drive depends on the sequencing.

Operations executed by the controller are divided into a command phase, an execution phase, and a result phase. During the command phase bytes are sent to the control registers and flags via the data in/out register. Then the requested operation is performed during the execution phase and, upon completion of the operation, the result phase is entered and status information is read by the processor. The outputs during the command phase and inputs during the result phase are performed using single-byte transfers, even though any data transfer that takes place in the execution phase is normally supervised by a DMA controller.

The possible 8272 commands are:

Read Data—Reads data from a data field on a diskette.

Write Data—Writes data to a data field on a diskette.

Read Deleted Data—Reads data from a field marked as being deleted.

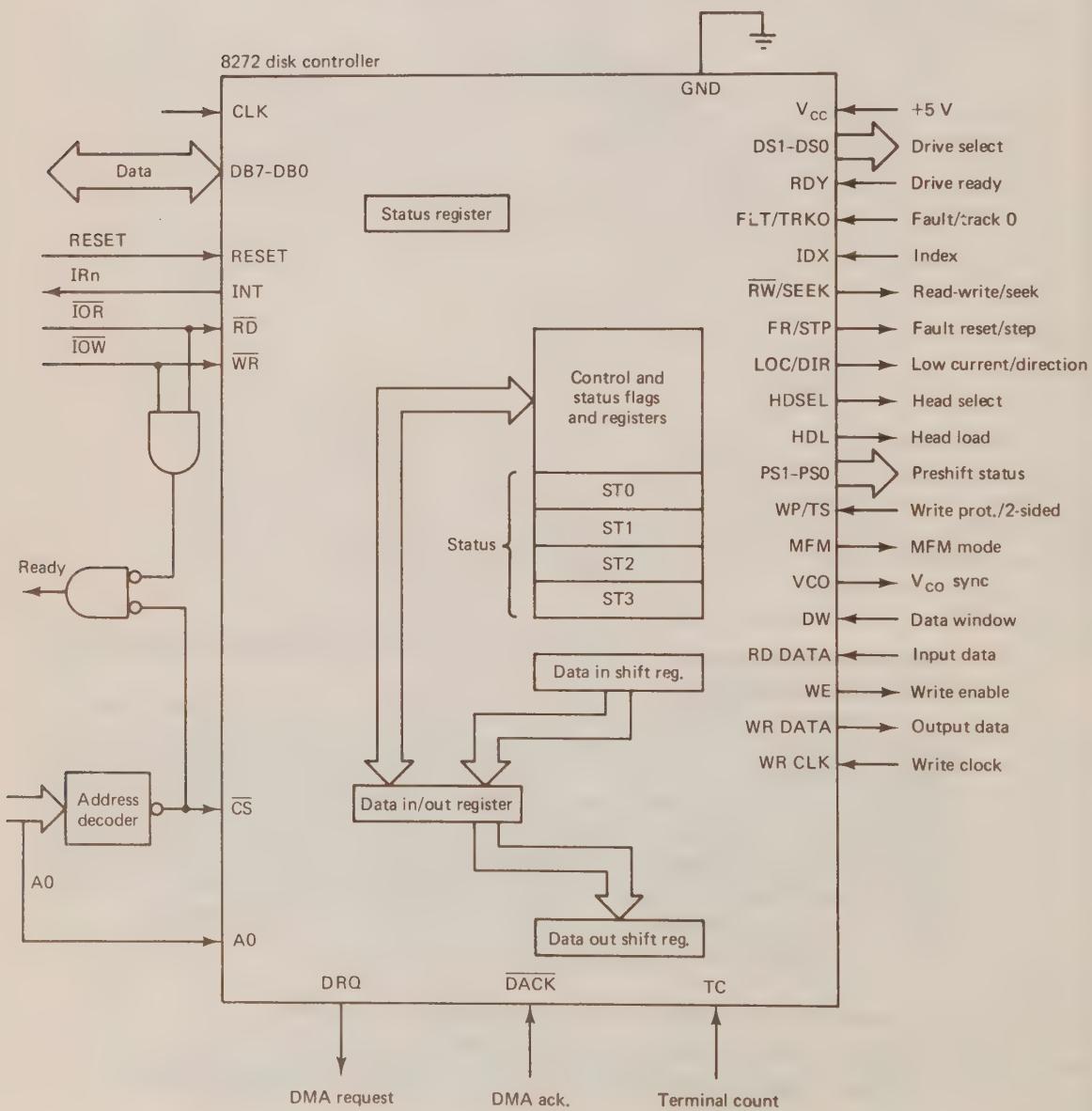
Write Deleted Data—Writes a deleted data address mark and puts filler characters in the data field.

Read A Track—Reads an entire track of data fields.

Read ID—Reads the identification field.

Format A Track—Writes the formatting information to a track using program supplied formatting parameters.

Figure 9-47 Diagram of an 8272 diskette controller.



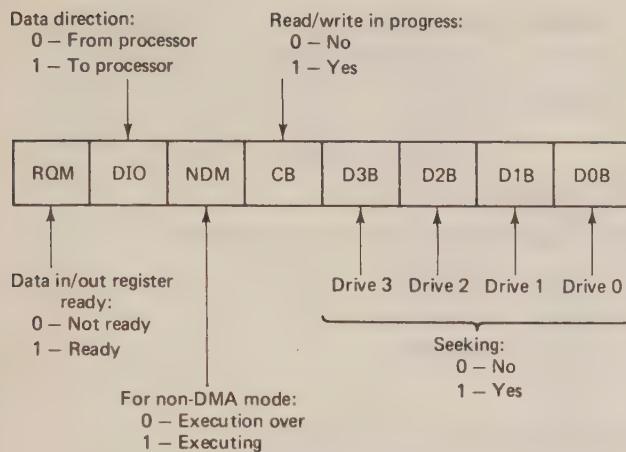


Figure 9-48 Format of the 8272's status register.

Scan Equal
Scan Low or Equal
Scan High or Equal } Scans the data for specified condition and makes an interrupt request when the condition is satisfied.

Recalibrate—Causes the head to be retracted to track 0.

Sense Interrupt Status—Reads status information from ST0 after interrupts caused by ready line changes and seek operations.

Specify—Sets the step rate, head unload time, head load time, and DMA mode.

Sense Drive Status—Inputs the drive status from ST3.

Seek—Positions head over a specified track.

The sequence needed to complete four of these commands is given in Fig. 9-49 as an example. The read data command includes all three phases, the seek command includes only the command and execution phases, the sense drive status command includes only the command and result phases, and the specify command consists only of the command phase. In all cases, the commands must be performed in their entirety (including the result phase, if applicable) or they will be considered incomplete. If an invalid command is given, the 8272 will return the status byte ST0 in response to the next input from the data in/out register.

Several of the commands require that some of the status bytes ST0, ST1, ST2, and ST3 be read during their result phases. These registers contain bits for indicating such things as CRC errors, overrun errors, drive selected, end of seek, whether or not the diskette is two-sided, and so on.

As an example, consider an 8272 whose even address is 002A. The 8272 could be initialized to a step rate of 6 ms per track, a head unload time of 48 ms, a head load time of 16 ms, and the DMA mode by the specify command sequence

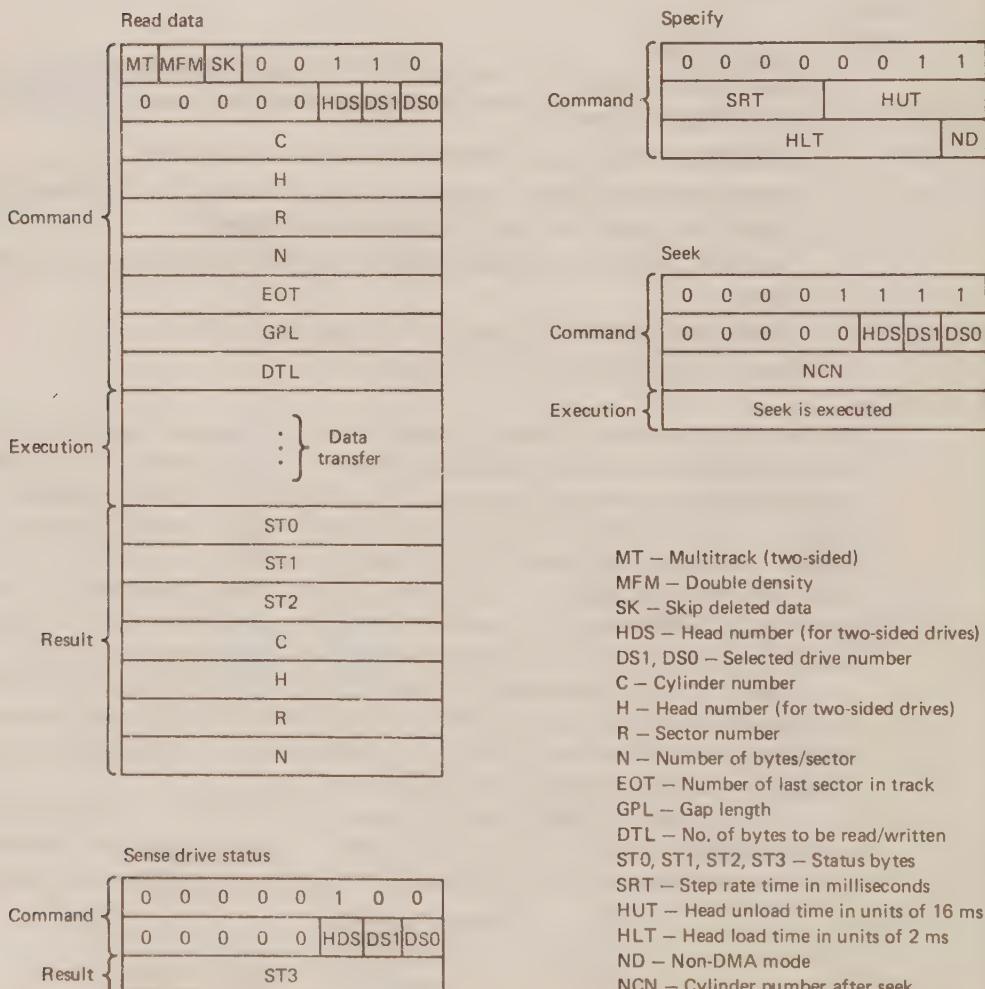
```
%CHECK (2AH,80H)
        MOV     AL,03H
        OUT    2BH,AL
```

```
%CHECK (2AH,80H)
    MOV AL,63H
    OUT 2BH,AL
%CHECK (2AH,80H)
    MOV AL,10H
    OUT 2BH,AL
```

where the macro CHECK is defined as follows:

```
/*DEFINE (CHECK (PORT,STAT)) LOCAL AGAIN
(%AGAIN: IN AL,%PORT
        AND AL,0C0H
        XOR AL,%STAT
        JNE %AGAIN
)
```

Figure 9-49 Typical 8272 commands.



The reason the %CHECK macro is needed before each output is that the two MSBs of the status register must be 10 before each command byte is written into the 8272. Also during the result phase, these 2 bits must be 11 before each byte is read from the 8272's data register.

The seek command sequence

| | |
|------------------|--------|
| %CHECK (2AH,80H) | |
| MOV | AL,0FH |
| OUT | 2BH,AL |
| %CHECK (2AH,80H) | |
| MOV | AL,02H |
| OUT | 2BH,AL |
| %CHECK (2AH,80H) | |
| MOV | AL,30 |
| OUT | 2BH,AL |

would cause the head on drive 2 to be moved to cylinder 30 and head 0 to be selected. Figure 9-50 defines two macros for executing the read data command. It is assumed that a sequence such as

| | | |
|-------|------|--------|
| IDLE: | IN | AL,2AH |
| | TEST | AL,12H |
| | JNZ | IDLE |

would appear just prior to a READ_COM macro call to make certain that the desired drive (which in this example is drive 1) is available. Also, the associated DMA must be initialized before the READ_COM macro call is made so that DMA transfers may begin as soon as the read operation is performed. It is assumed that a call to the READ_STAT macro would be made only after the execution phase, and the read is known to be complete (e.g., after an interrupt on completion interrupt has occurred).

9-7 MAXIMUM MODE AND 16-BIT BUS INTERFACE DESIGNS

As noted in the chapter's introduction, all of the Intel examples in the first six sections of this chapter are based on a minimum mode 8088 processor. To convert the designs to a maximum mode system two primary changes are necessary. First of all, an 8288 bus controller must be connected into the system as shown in Figs. 8-9 and 8-10. For a system that contains an 8237 DMA controller, the 8288 would replace the circuit for encoding the RD, WR, and IO/M signals shown in Fig. 9-38 and detailed in Fig. 9-39. In any event, with the inclusion of an 8288 the RD and WR pins on the interfaces would be attached to the IORC and IOWC outputs from the 8288 and the IO/M lines shown entering the address decoders would no longer be required.

The other major change concerns the HRQ and HLDA signals used to make bus requests and grants. The 8237 is designed to output a continuous request HRQ signal until it is ready to relinquish the bus, at which time it drops the signal. Also, the processor outputs a continuous HLDA signal. This is compatible with an 8086/

```

%*DEFINE (READ_COM (ADDR,HDSDS,C,H,R,N,DTL))
    PUSH    AX          ;SAVE REGISTERS
    %CHECK (2AH,80H)
        MOV     AL,46H      ;OUTPUT COMMAND
        OUT    %ADDR,AL    ;CODE
    %CHECK (2AH,80H)
        MOV     AL,%HSDS   ;OUTPUT HEAD AND
        OUT    %ADDR,AL    ;DRIVE NOS.
    %CHECK (2AH,80H)
        MOV     AL,%C       ;OUTPUT CHANNEL NO.,
        OUT    %ADDR,AL
    %CHECK (2AH,80H)
        MOV     AL,%H       ;HEAD NO.,
        OUT    %ADDR,AL
    %CHECK (2AH,80H)
        MOV     AL,%R       ;SECTOR NO.,
        OUT    %ADDR,AL
    %CHECK (2AH,80H)
        MOV     AL,%N       ;NO. OF DATA BYTES,
        OUT    %ADDR,AL
    %CHECK (2AH,80H)
        MOV     AL,26       ;NO. OF SECTORS,
        OUT    %ADDR,AL
    %CHECK (2AH,80H)
        MOV     AL,17       ;GAP LENGTH, AND
        OUT    %ADDR,AL
    %CHECK (2AH,80H)
        MOV     AL,%DTL     ;DATA LENGTH
        OUT    %ADDR,AL
        POP    AX
    )
%*DEFINE (READ_STAT (ADDR,RDSTAT)) LOCAL LP
(
    PUSH    AX          ;SAVE REGISTERS
    PUSH    CX
    PUSH    SI
    MOV    SI,0          ;INPUT STATUS TO
    MOV    CX,7          ;ARRAY RDSTAT
    %LP:
    NOP
    %CHECK (2AH,0COH)
        IN     AL,%ADDR
        MOV    %RDSTAT[SI],AL
        INC    SI
        LOOP  %LP
        POP    SI          ;RESTORE REGISTERS
        POP    CX
        POP    AX
)

```

Figure 9-50 Macros for executing read data commands.

8088 processor in minimum mode, but in maximum mode the processor uses a single RQ/GT line to both receive requests and issue grants and expects to see only a pulse at the time the request is made. The request is acknowledged by outputting a pulse and a second pulse must be sent to the processor from the DMA controller at the conclusion of the DMA activity. A circuit for converting between the two-line continuous signals associated with the 8237 and the one-line pulses of a maximum mode processor is given in Fig. 9-51.

The problems associated with connecting the 8-bit interface devices to a 16-bit bus of an 8086 are related to the need to transfer even-addressed bytes over the lower half of the data bus and odd-addressed bytes over the upper half. For interfaces that communicate only with the processor (i.e., do not utilize DMA), the problem can be solved quite simply. Instead of connecting the address lines for selecting individual registers internal to the interface, say An-A0, to the interface's pins An-A0, attach lines A(n+1)-A1 to those pins as shown in Fig. 9-52.

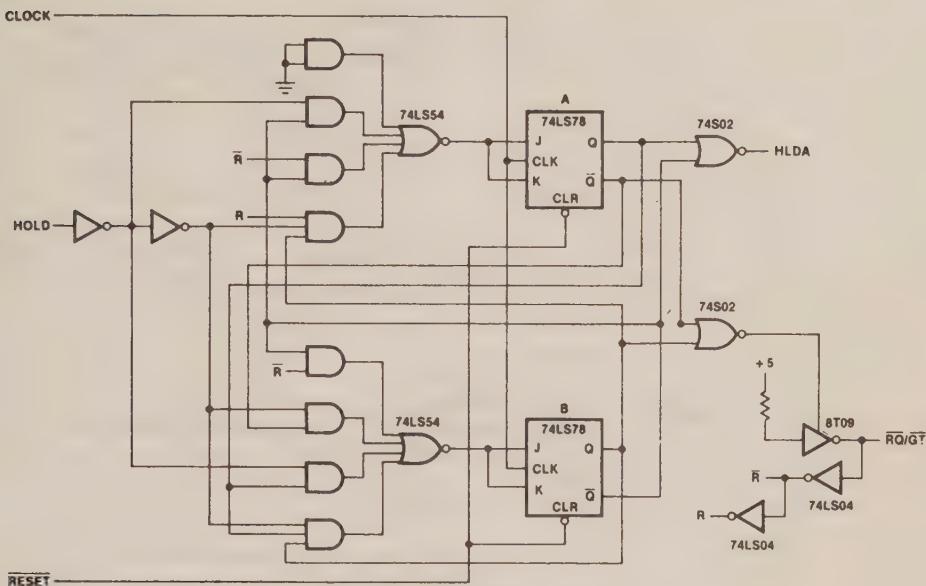


Figure 9-51 Bus request and grant conversions needed when connecting an 8237 DMA controller to a maximum mode 8086/8088. (Reprinted by permission of Intel Corporation. Copyright 1979.)

This means that the interface will be assigned only even addresses in the I/O address space beginning with an address divisible by 2^{n+1} and the interspersed odd addresses would not be used. For example, if the A1 and A0 pins on the 8255A were connected to the A2 and A1 address lines and the beginning address of the 8255A ports were 08F8, then all transfers to and from the 8255A would be made over the low-order byte of the bus. The ports A, B, and C would have the addresses 08F8, 08FA and 08FC, respectively, and the control register would be assigned the address 08FE. Likewise, consecutive odd addresses can be assigned to an interface if the interface is connected to the high-order byte of the bus.

To use successive addresses and both halves of a 16-bit data bus, the bus could be connected as shown in Fig. 9-53. To access a register the 8086 uses its BHE and A0 as follows:

| BHE | A0 | Transfer |
|-----|----|--|
| 0 | 0 | Not useful |
| 0 | 1 | Odd-addressed byte on upper half of bus |
| 1 | 0 | Even-addressed byte on lower half of bus |
| 1 | 1 | Not possible |

where 0 is low and 1 is high. By using these signals and two 8286 transceivers data can be transferred between the interface and the data bus lines D7-D0 when $\overline{\text{BHE}} = 1$ and $\text{A}0 = 0$, and the bus lines D15-D8 when $\overline{\text{BHE}} = 0$ and $\text{A}0 = 1$. The RD signal is used to determine the direction of the data flow. The ready logic is not shown.

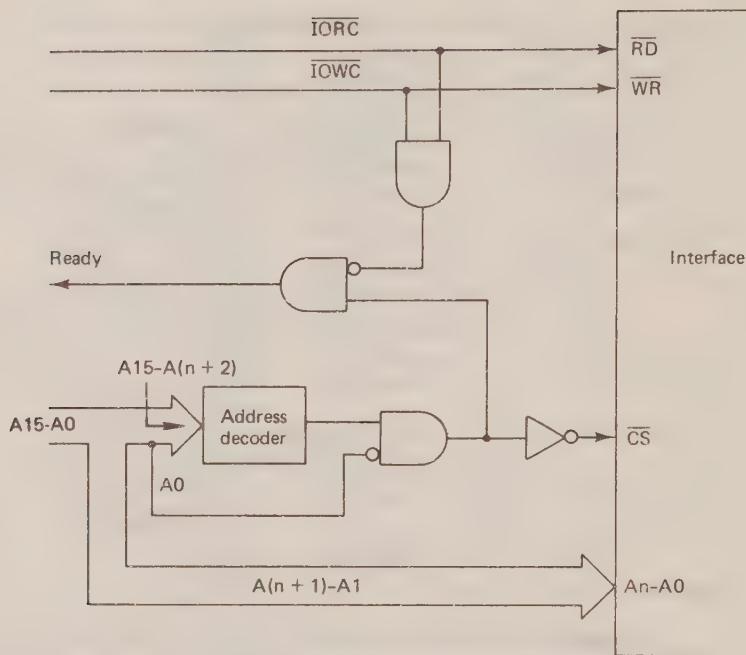


Figure 9-52 Accessing only even-addressed I/O ports.

For an 8237-based DMA system, the bus control logic shown in Fig. 9-38 must be altered as indicated in Fig. 9-54. The \overline{BHE} signal coming out of the 8086 would be latched by the same 8282 as is used by the A19–A16 lines. The A0 line would be connected to the \overline{BHE} line through a tristate inverter which is controlled by the 8237's AEN signal. When AEN is active and $A_0 = 0$, \overline{BHE} is high, indicating that the transfer is to be made over the lower byte of the bus. If AEN is active and $A_0 = 1$, \overline{BHE} is low and the transfer is made over the upper byte. In addition, both the interface and the 8237 must be connected to the bus through extra logic similar to that given in Fig. 9-53.

Although the above paragraphs have been concerned with the communication between an 8-bit interface and a 16-bit data bus, some attention should be given to the design of a 16-bit interface. Such an interface would transfer entire words to and from the data bus and would tend to double the utilization of the available bus cycles. A 16-bit interface design based on two 8255As is given in Fig. 9-55. The A2 and A1 lines in the address bus are connected to the A1 and A0 pins of both 8255As; thus the 16-bit ports are formed from the pairs of ports A, B, C, and the control/status registers. The lower 8255A occupies 4 consecutive even addresses and the upper 8255A occupies 4 consecutive odd addresses. If bits A15–A3 match the address designed into the address decoder, then the decoder will emit a 0 chip select signal. For the lower 8255A, if both the chip select and A0 signals are 0, then a 0 is applied to CS. For the upper 8255A, both the chip select and \overline{BHE} signals must be 0 in order for a 0 to be sent to the CS. (Therefore, it is

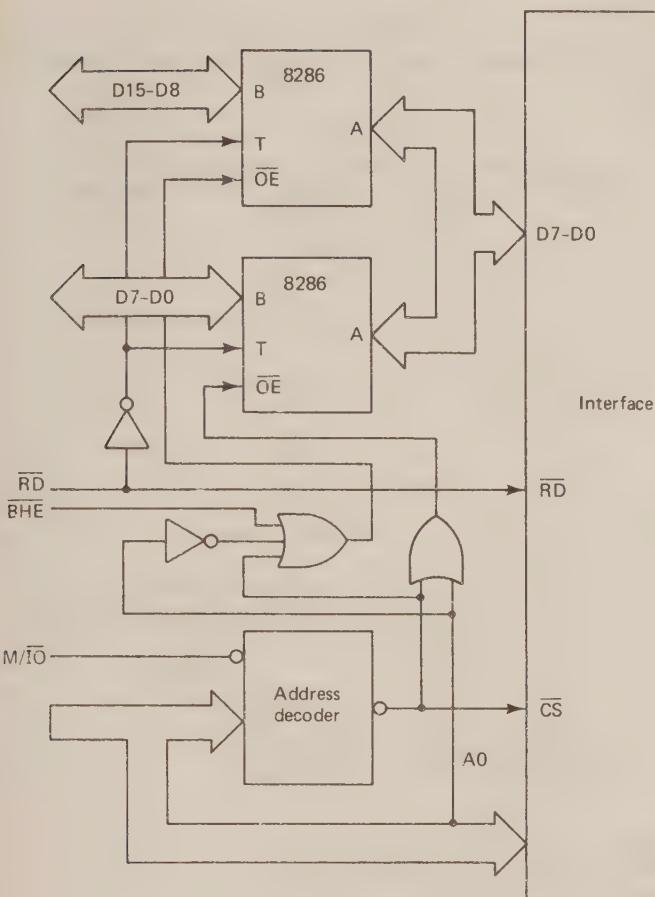


Figure 9-53 Connection of an 8-bit interface to a 16-bit data bus.

possible to address the 8255As individually.) The read, write, and reset control lines are connected to the \overline{RD} , \overline{WR} , and RESET pins of both of the 8255As and a ready signal is returned if either \overline{CS} signal is active.

One other alternative in interface design is to treat registers of an I/O device as memory locations. These locations are then referred to as *memory-mapped I/O*. This scheme requires the interface to respond to memory read and memory write commands. The chief advantage of memory-mapped I/O is that the device registers can be accessed and manipulated with any instruction or addressing mode that references memory operands, thus providing programming flexibility and reducing coding. However, if the control and status registers share the same address, as they do for several of the Intel interface devices, one must not use an instruction such as

OR CONTROL,00000001B

which would input from the status register and output to the control register. Because the same address may not be assigned to both memory and an I/O device,

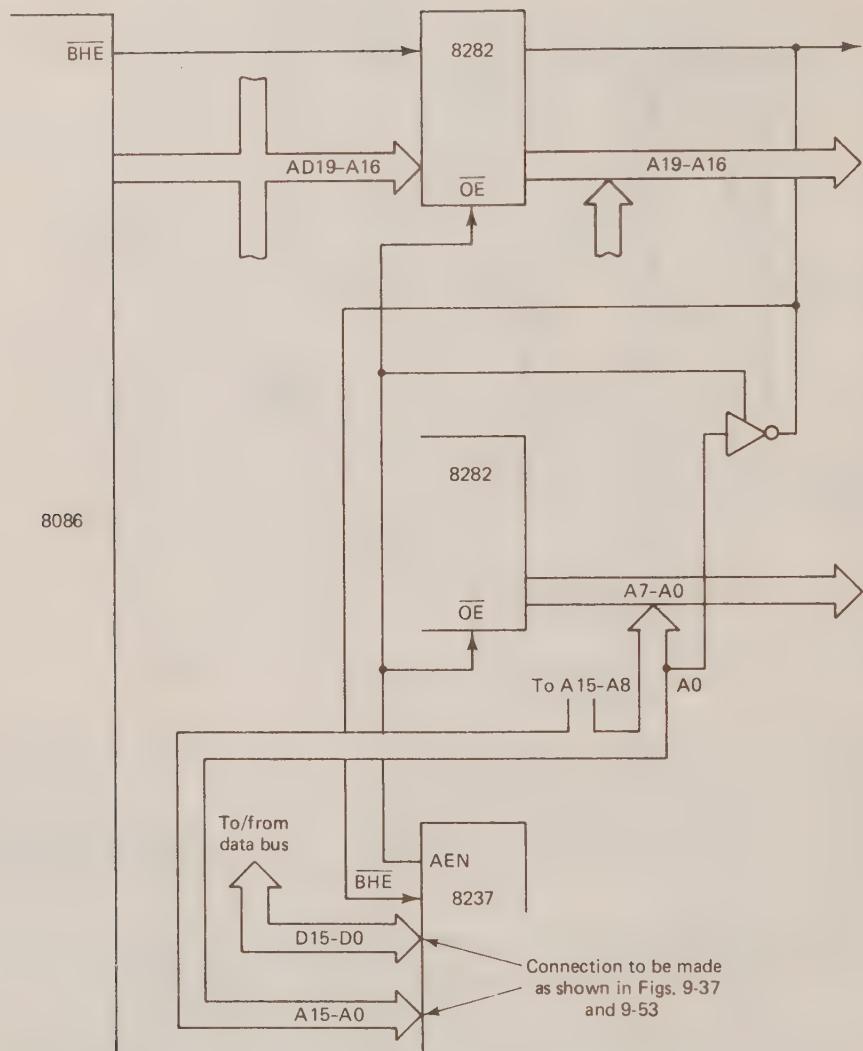


Figure 9-54 Using an 8237 DMA controller with a 16-bit data bus.

memory-mapped I/O causes the maximum number of available memory locations to be reduced.

BIBLIOGRAPHY

1. McNamara, John E., *Technical Aspects of Data Communications* (Bedford, Mass.: Digital Equipment Corporation, 1978).
2. *1981 Data Component Catalog* (Santa Clara, Calif.: Intel Corporation, 1981).

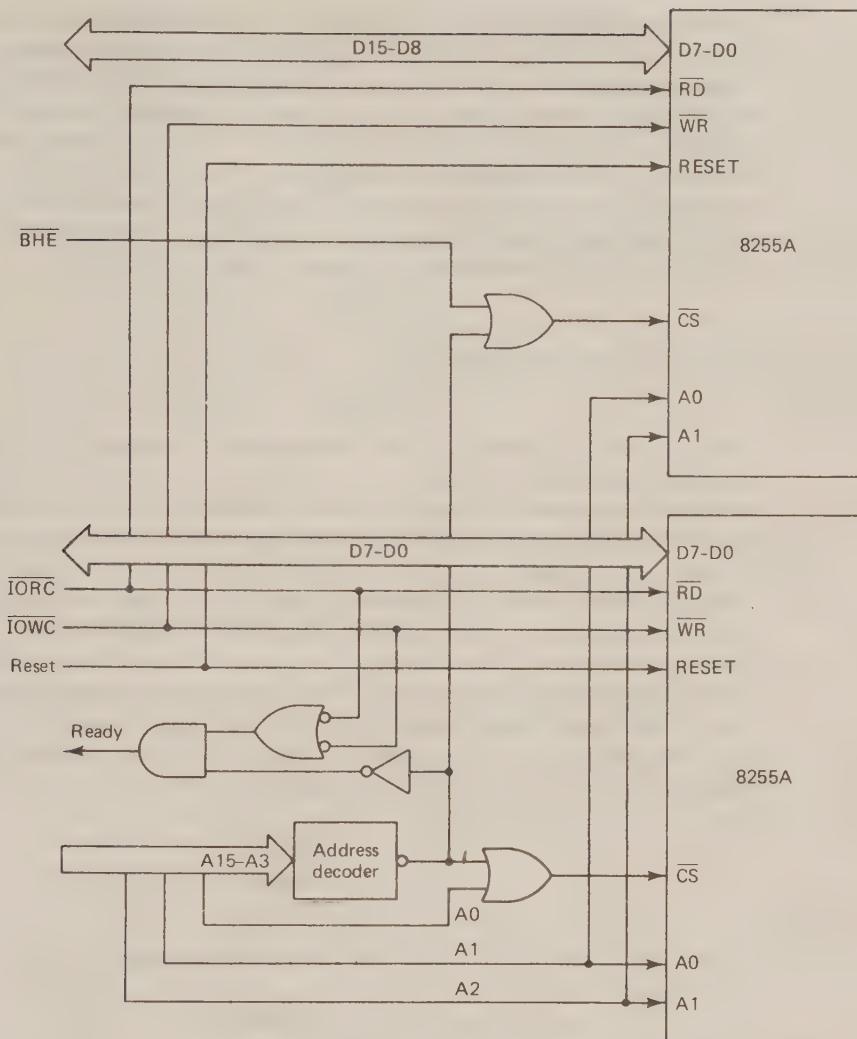


Figure 9-55 Example of a 16-bit parallel I/O interface.

3. Peatman, John B., *Microcomputer-Based Design* (New York: McGraw-Hill Book Company, 1977).
4. Hall, Douglas V., *Microprocessors and Digital Systems* (New York: McGraw-Hill Book Company, 1980).

EXERCISES

1. Assume the 7-bit ASCII code, even parity, and 2 stop bits and sketch the timing diagram of the letter C as it is transmitted over an asynchronous serial communication link.

2. If there is no dead time between characters, characters contain 7 information bits, there is a parity bit and 2 stop bits with each character, and the bit rate is 600 bits per second, what is the information transfer rate in characters per second? Compare this figure with the information transfer rate of a synchronous transmission having the same bit rate, a message format consisting of two sync characters followed by 200 seven-bit information characters, and no imbedded idle characters.
3. According to the discussion in Sec. 9-1-3, what is the approximate maximum distance a 4800-baud signal could be transmitted without the aid of communications equipment?
4. If the equivalent circuit shown in Fig. 9-8 is such that

$$R_o = 500 \Omega \quad C_o = 2 \mu F$$

C_L is due to 100 ft of a 20-pF/ft line

$$R_L = 4000 \Omega \quad E_L = 0 V$$

- what would V_t be when $V_o = 7 V$? Determine the maximum of the derivative of V_t as V_o instantaneously changes from a steady state $-7 V$ to $+7 V$. How long would it take for V_t to go from $-3 V$ to $+3 V$ if V_o switches from $-7 V$ to $+7 V$? Would all of the RS-232-C electrical specifications be satisfied if the line is operated at 1200 baud?
5. If the mode register of an 8251A contains 01111011, what will be the format of a transmitted character? In order for the receiver and transmitter baud rates to be 300 and 1200, respectively, what would the frequencies applied to RxC and TxC need to be?
 6. Assuming that the contents of the mode register of an 8251A are 00010100, determine the character and message formats of the 8251A's serial transmission.
 7. Given that the sequence shown below is output with $A0 = 1$ following a reset, describe the action taken as each output is received by the 8251A.

| |
|----------|
| 00010100 |
| 00010110 |
| 00010110 |
| 00110011 |
| ⋮ |
| 01000000 |

} Output with $A0 = 0$

(See the flowchart in Fig. 9-14.)

8. Write a program sequence that will initialize an 8251A so that it will transmit asynchronous characters containing 7 information bits, 1 stop bit, and no parity bit. The baud rate factor is to be 64. The even address associated with the 8251A is to be 008A. Extend the sequence so that it outputs a command that activates Request to Send and Data Terminal Ready signals and enables the transmitter.
9. Extend the program sequence in Exercise 8 so that it will use programmed I/O to output the message

AN OVERRUN ERROR HAS OCCURRED

and then clear the overrun error bit.

10. Write an interrupt routine that will output the 20 bytes beginning at MESSAGE to the

8251A whose even address is 25C0. Each time the routine is entered it is to output 1 byte, and when it has output 20 bytes it is to put a 1 in FLAG. CNT is to be used to store the byte count.

11. Suppose that the beginning address of an 8255A is 0500 and write a program sequence that will:
 - (a) Put both groups A and B in mode 0 with ports A and C being input ports and port B being an output port.
 - (b) Put group A in mode 2 and group B in mode 1 with port B being an output.
 - (c) Put group A in mode 1 with port A being an input and PC6 and PC7 being outputs, and group B in mode 1 with port B being an input.
12. Assume an 8255A, whose beginning address is 0030, is such that group A is in mode 2 and group B is in mode 0 with PB7–PB0 as inputs and PC2–PC0 as outputs. Write a program that will output 100 bytes from an array beginning at OUT_ARRAY to the device connected to port A. Before outputting each byte the program must loop until PB1, PB2, and PB3 are all active and after outputting the byte the program must set PC1 to 1.
13. Using an 8255A, an 8282 latch, an 8286, and other logic, design a circuit for setting and reading a bank of eight relays. Assume that the 8286 drivers are sufficient to drive the output circuits which supply the current to the relay coils. Specify the 8255A modes and all of the necessary control signals. The relays and their associated circuitry need not be included in the design.
14. By also using port B as a mode 1 input, modify the design in Fig. 9-23 so that a 12-bit A/D converter can be serviced. Give a program sequence for inputting a sample and storing it in the word whose location is represented by SAMPLE.
15. Describe how an 8254 could be used to count the pulses input to it for a period of time that is controlled by a second input. At the end of the overall counting time an interrupt request is to be made on an IR2 line, and after every 20 of the pulses being counted an interrupt request is to be made on an IR3 line. Also write a program sequence for initializing the 8254.
16. Suppose that an 8254 connected to a 1-kHz clock is to be used to keep the time of day in BCD format to the nearest second. It is to start its timekeeping immediately after the bytes HOURS, MINUTES, SECONDS, and AMPM have been loaded with the current time. Write a program sequence that will initialize the 8254 and an interrupt routine that will update the time at the end of each second.
17. Assume that in Fig. 9-30, ports A and B of an 8255A are connected to the keyboard. Draw a flowchart for the keyboard scanning routine. (Do not consider multiple key closures.)
18. Suppose that an 8255A is used to drive the multiple-digit display in Fig. 9-32 in the multiplexed mode.
 - (a) Show the connections between the 8255A and the display.
 - (b) Draw a flowchart for a display routine.
19. For the design in Exercise 18, draw a block diagram to show how the refresh operations can be eliminated by using external data latches.
20. Suppose that eight devices are to be monitored and that each device has an 8-bit register for storing its status. Draw a logic diagram to illustrate how this can be accomplished using an 8279.

21. Write a program sequence that will cause an 8237 DMA controller to perform a memory-to-memory transfer of the 1000 bytes beginning at SOURCE to an array beginning at DST. The port addresses of the 8237 are to be 0600 through 060F.
22. Explain exactly how the 8286s in Fig. 9-53 are controlled during an:
 - (a) Output to the control register.
 - (b) Input from the status register.
 - (c) I/O interface to memory transfer.
 - (d) Memory-to-memory transfer.given that the interface is an 8237.
23. Write a program sequence that will cause an 8237 to perform a demand mode block transfer of 500 bytes to an array beginning at BUFFER. The bytes are to be stored starting with the highest address (i.e., BUFFER + 499). The port addresses of the 8237 are to be 1020 through 102F.
24. Give a set of instructions that will:
 - (a) Mask channels 0 and 3 of an 8237.
 - (b) Cause a program-initiated DMA transfer on channel 2 of an 8237 according to the current contents of its control, address, and byte count registers.
25. Draw a timing diagram for an 8237 block mode transfer of 4 bytes from an I/O device to memory.
26. Given that the head load time is 32 ms, seek time is 8 ms per track and there are 40 tracks, and the rotational speed is 360 rpm, determine the average access time of a soft-sectorized diskette. Next determine the average time needed to first recalibrate the head (i.e., move the head to track 0) and then access an arbitrary sector.
27. Write a program sequence that will use the sense drive status command of an 8272 to put a 1 in location FLAG2 if drive 2 is write protected (bit 6 of result is 1).
28. Give a set of instructions that will loop until one of the drives in an 8272-based diskette subsystem is not busy and will then branch to ROUTn, where n is the number of the drive that is first found to be idle.
29. Draw a flowchart of the complete sequence needed to output a block of data to an 8272-based diskette subsystem that includes an 8237 DMA controller. The flowchart should take into account the initialization steps for both the diskette and DMA controllers as well as the steps to execute the phases of write data command.
30. Describe in detail the bus activity that takes place while two consecutive bytes are transferred from memory to an I/O device over a 16-bit 8086 bus by an 8237 DMA controller.
31. Consider the 16-bit parallel I/O device shown in Fig. 9-55. Give a program sequence that will put both groups in both 8255As in mode 0 with ports A and C as inputs and port B as an output. Then write a pair of instructions that will output 1720 through port B.
32. Modify Fig. 9-47 so that the data buffer register and control/status registers are assigned to consecutive even addresses.

Semiconductor Memory

In its most general sense the word *memory* refers to any device that stores information for later use. Under this definition, the memory of a computer can be divided into two categories. One category pertains to that part of the computer that holds the instructions and data that are presently being operated on, i.e., the part that the processor can access directly. The other category consists of the facilities that can store information, but the information must be transferred to the memory in the first category before it can be used by the processor. This book has referred to components in the former category as *main memory*, or simply *memory*, and components in the latter category as *mass storage*. This chapter will be concerned with main memory and how it is interfaced to the system bus.

As we have seen, the main memory is comprised of groups of bits called bytes and words that are addressed as units. Until now the processor-oriented meaning of "word" has been used which refers to the numbers of bits that can be simultaneously transferred over the data bus and manipulated as a unit by the processor. Most memory designers, however, use the term "word" to mean the smallest group of bits that is associated with an address. In an 8086/8088 system these groups are the 8-bit bytes. Therefore, when considering memory design in general, "word" will often be used in contexts where "byte" appeared in previous discussions. When a chance of confusion exists, the term "word" will be modified by the word "memory" or the word "processor." The number of bits in a memory word is called its *length* (e.g., the length of an 8086/8088 memory word is 8).

One of the principal attributes of a memory is whether or not it can retain its contents while its power is turned off. A memory that can do so is said to be *nonvolatile*; otherwise, it is said to be *volatile*. Whether or not a memory is volatile is particularly important if it contains instructions, because if it is volatile, these

instructions will be lost during even momentary power failures and will have to be reloaded each time the computer is brought up. For most microcomputer applications, the system must contain at least some nonvolatile memory, but most of the memory can be volatile.

Another important attribute depends on the ways in which the memory can be accessed. If a memory unit can only be read from, it is called *read-only memory (ROM)*, but if it can be both read from and written into, it is called *read/write memory*. *Random access* refers to storage for which all locations can be accessed in the same amount of time, as opposed to *sequential* (or *serial*) *access*, in which the locations must be referenced in order (e.g., as on a tape). The main memory of all modern computers is random access; however, when referring to main memory the phrase "random access" has historically been used as a synonym for read/write and read/write memory has been referred to as *random access memory (RAM)*. Thus, the two principal types of main memory are ROM and RAM.

The most prominent kind of nonvolatile read/write memory is *magnetic core*, or simply *core, memory*, which is constructed of small, doughnut-shaped magnets. Volatile read/write memory is made from semiconductor ICs. Although core memory is nonvolatile, it requires a considerable amount of support electronics, which causes it to be relatively expensive. As a result, most microcomputer systems use semiconductor memory and rely on mass storage devices or memory with backup power to store information during shutdown. ROM memory, which is obviously nonvolatile, is also made from semiconductor ICs. Because it does not need to include write circuitry, ROM has the least complicated design. ROM is commonly used to store bootstrap loaders so that the system can be brought up without manually loading initialization instructions. It can, however, be used only to store data and programs that never change. Because core memory is seldom used in microcomputer systems it will not be considered in this book.

This chapter begins with a discussion of the overall design of memory. Sections 10-2 and 10-3 consider the two types of semiconductor read/write memory: static RAM and dynamic RAM. Section 10-4 considers the design of backup power for volatile memory so that it becomes, in effect, nonvolatile memory. The last section examines the various types of ROM.

10-1 GENERAL MEMORY ORGANIZATION

The memory of a computer system normally consists of one or more PC boards that are connected to the system bus. On each board is a module that is addressed by the high-order bits on the address bus. As shown in Fig. 10-1, most systems include both ROM and RAM modules. It should be pointed out, however, that some small systems which are used as simple controllers may contain only ROM and/or may include the memory on the same board, or even the same IC chip, as the processor.

The general design of a memory module is shown in Fig. 10-2. It consists primarily of an interface and an array of memory IC devices, each of which contains

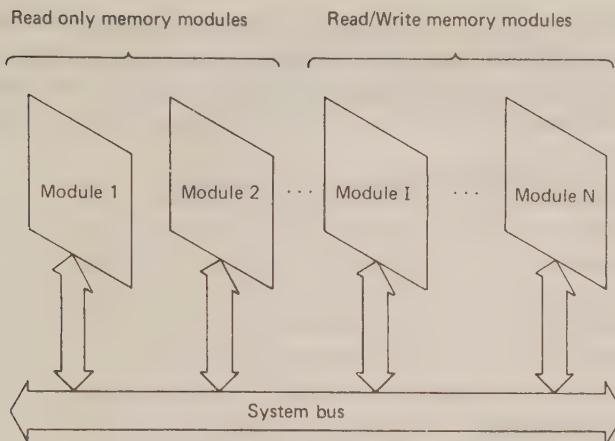


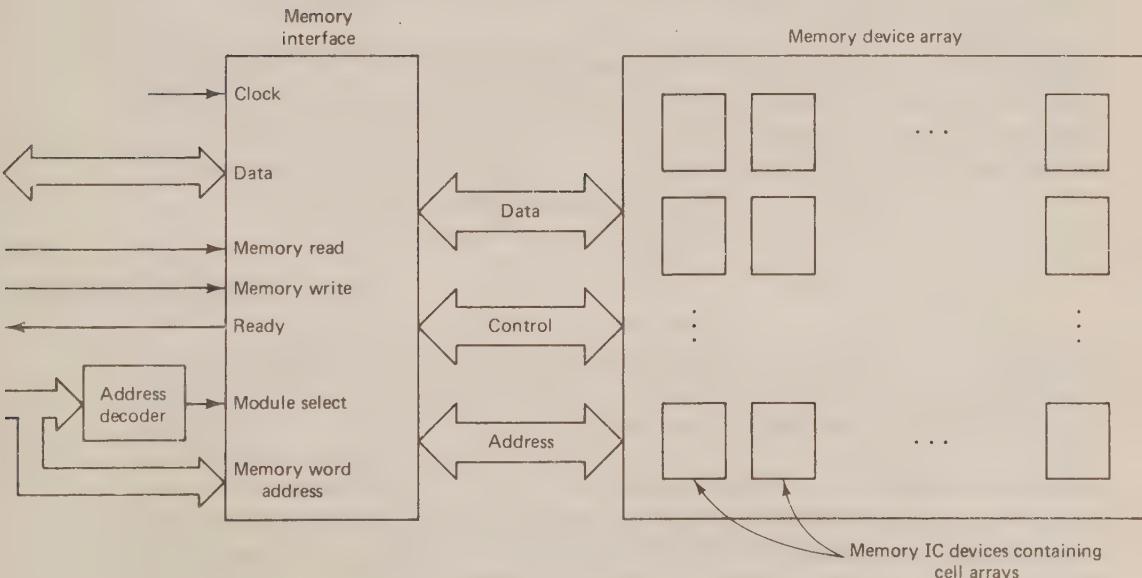
Figure 10-1 Overall organization of memory. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

an array of memory cells with each cell being the circuitry needed to store 1 bit. The cells in a device may be accessed separately or in groups, but either way the following relations must hold:

$$\frac{\text{No. of devices in a row}}{\text{No. of devices in a column}} = \frac{\text{No. of bits in a word}}{\text{No. of cells in a group}}$$

$$\frac{\text{No. of devices in a column}}{\text{No. of devices in a row}} = \frac{\text{No. of words in module}}{\text{No. of groups in a device}}$$

Figure 10-2 Typical memory module design.



A memory device is said to be an $M \times N$ device if it contains M groups with each group consisting of N cells. Similarly, a $K \times L$ module is one that consists of K words, each having L bits. To demonstrate the application of the above relations, several memory module designs which utilize typical memory devices are summarized in Fig. 10-3.

The principal criteria involved in designing a memory are:

1. Cost.
2. Capacity.
3. Speed.
4. Power consumption.
5. Reliability.
6. Volatility and access capability.

The cost of a memory module normally consists of two components, one which is independent of the size of the module and is called *overhead*, and one which is proportional to the size and is called the *incremental cost*. The overhead cost is due primarily to the support and interface electronics in the module, and the incremental cost is most closely related to the cost of the memory devices. Both overhead and incremental costs are dependent on the number of pin connections and the complexity of the board design. Therefore, memory devices with large bit capacities and that require few supporting chips ordinarily have a cost advantage. Because the overhead tends to be the same regardless of the capacity of the module but must be included for each module, it is generally better to fulfill the overall

Figure 10-3 Representative memory device arrays. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

| Memory Size | Type of RAM | No. of chips in each column | No. of chips in each row | No. of chips in the module |
|-------------|-------------|-----------------------------|--------------------------|----------------------------|
| 4K x 8 | 1K x 1 | 4 | 8 | 32 |
| | 4K x 1 | 1 | 8 | 8 |
| | 256 x 4 | 16 | 2 | 32 |
| | 1K x 4 | 4 | 2 | 8 |
| 4K x 16 | 1K x 1 | 4 | 16 | 64 |
| | 4K x 1 | 1 | 16 | 16 |
| | 256 x 4 | 16 | 4 | 64 |
| | 1K x 4 | 4 | 4 | 16 |
| 16K x 8 | 1K x 4 | 16 | 2 | 32 |
| | 4K x 1 | 4 | 8 | 32 |
| | 8K x 1 | 2 | 8 | 16 |
| | 16K x 1 | 1 | 8 | 8 |
| 64K x 8 | 16K x 1 | 4 | 8 | 32 |
| | 64K x 1 | 1 | 8 | 8 |

memory capacity requirement using as few modules as possible. Another factor which enters into the overhead is the cost of the power supply. The fewer the number of supply voltages that are required, the less complicated the power supply and board design.

The speed of a memory is gauged by its *access time*, which is defined as the time delay between the receipt of a stable address input and the data output. This time depends on many factors and is even related to the bit capacity of the memory devices. High-speed transistors tend to require more chip space so that fewer cells can be included on each chip. Also, high-speed devices, which are normally made using a bipolar technology, are more expensive and consume more power.

Power consumption can be very important in systems that must sometimes be powered by batteries or solar cells (e.g., space vehicles). The technology used in constructing the IC determines the power required for each cell. The technology most often used in low-power applications is the complementary metal-oxide semiconductor (CMOS) technology. The main disadvantage associated with CMOS is that a fair amount of chip space is needed for each cell; thus the capacity of each device is reduced. Unfortunately, power consumption and speed tend to be proportional and it is difficult and expensive to achieve both low power and high speed. The technology that has been found to be a good compromise of speed, power consumption, capacity per device, and cost is the high-density MOS (HMOS) technology.

Because the reliability of an IC is high once it has been thoroughly tested, the reliability of a module is strongly related to the number of solder connections and board complexity. Therefore, reliability increases as the total pin count decreases, thus supplying an additional incentive to minimize the number of memory and supporting devices in a module.

Nonvolatility and access capability are pretty much dictated by the application. Unless an application requires nonvolatility there is no reason to provide it. If nonvolatile RAM is needed, then either core memory must be used or backup power must be available so that the volatile semiconductor memory can be maintained in a powered-up state. ROM is generally used whenever possible because it is less expensive, nonvolatile, more reliable, and impervious to noise, and its simple construction lends itself to high cell density.

10-2 STATIC RAM DEVICES

For static memory devices, a cell is commonly implemented using six MOS transistors, as shown in Fig. 10-4. Information is stored according to the states of transistors Q_1 and Q_2 . This cross-coupled transistor pair is such that when one transistor is on, the other is off, and vice versa. A 1 is assigned to the state that exists when Q_2 is on and Q_1 is off, and a 0 is assigned to the opposite state. The transistors Q_3 and Q_4 serve as resistors and Q_5 and Q_6 act as enable gates. During a write operation, first the cell is selected by raising the voltage level on the select line. When this is done, transistors Q_5 and Q_6 act as short circuits, so that the

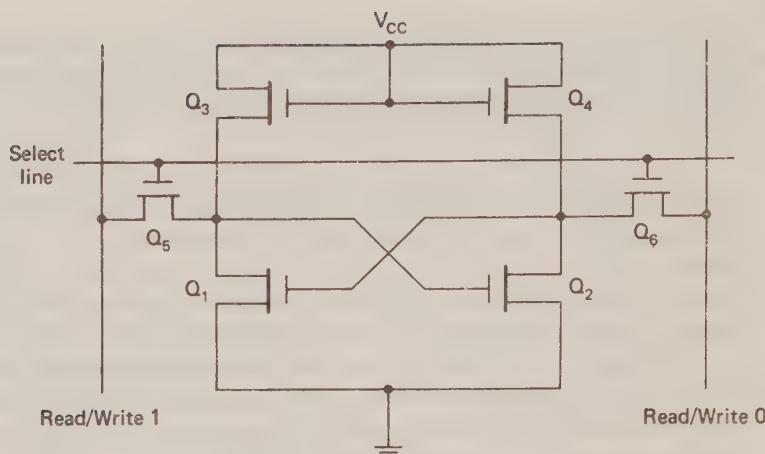


Figure 10-4 Schematic diagram of a six-transistor static RAM cell. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

Read/Write 1 line is applied to the gate of Q_2 and the Read/Write 0 line is applied to the gate of Q_1 . To write a 1 into the cell, a 1 is placed on Read/Write 1 and a 0 is placed on Read/Write 0; this causes Q_2 to be turned on and Q_1 to be turned off. On the other hand, if a 0 is to be written into the cell, a 1 is placed on Read/Write 0 and a 0 is placed on Read/Write 1. In either case, once they are set the states of Q_1 and Q_2 will remain unchanged until the next write operation. The cell can be read simply by applying a voltage to the select line. When this is done, the state of Q_1 is applied to Read/Write 0 and the state of Q_2 is applied to Read/Write 1.

The number of memory cells and their arrangement in a static memory device varies considerably. Common sizes range from 256×4 to $16K \times 1$. A 256×4 RAM contains 256 locations, each having 4 bits, while a $16K \times 1$ configuration provides 16K locations, each containing only 1 bit. The general organization of a static RAM is described by the block diagram of Fig. 10-5, which illustrates a $1K \times 1$ RAM. The memory cells are organized in a matrix of 32 rows and 32 columns. Address bits A_9 through A_0 are divided into row and column addresses and together specify one of the 1024 cells. Row address inputs (A_4 – A_0) are decoded to select one of the 32 rows of storage cells. Column address inputs (A_9 – A_5) not only select the column, but also enable the corresponding I/O circuits consisting of drivers and sense amplifiers. These circuits permit a stored bit to be output during a read operation and to be changed during a write operation. The read/write (R/W) control input specifies the type of operation, high for read and low for write. The chip enable input (\overline{CE}) is used to select the appropriate row of memory devices in the memory module.

A $4K \times 8$ memory device array which is constructed from $1K \times 1$ devices is shown in Fig. 10-6. If a chip is enabled, a read or write operation will proceed as specified by the R/W control input. Otherwise, the read/write signal will not be recognized and the output is forced into a high-impedance state. This allows the

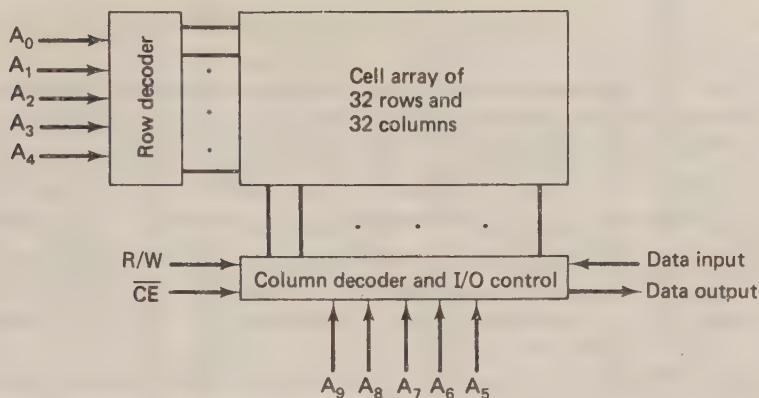


Figure 10-5 Block diagram of a $1K \times 1$ memory device. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

data outputs of several memory chips to be directly tied together so that $2K \times 8$, $4K \times 8$, and so on, memories can be constructed from columns of $1K \times 1$ chips with each column contributing one bit of the data byte. When this is done, the bit being output not only depends on the signals on the address lines, but also depends on which chips receive the chip enable signal. Each row in the array is connected to a row enable line and the row enable lines are activated by higher-order address bits, which for this example are bits A₁₁ and A₁₀. When a row is selected each device in the row will input or output a bit according to the signals on the A₉-A₀ lines. In summary, if the address contains 16 bits, A₁₅-A₁₂ would select the module, A₁₁ and A₁₀ would select the row, and A₉-A₀ would select the bits in the devices which constitute the addressed byte.

Because each cell in a static RAM device must contain several transistors, each device generally contains fewer cells than a dynamic RAM chip of comparable design. Also, they tend to use more power since one of the two cross-coupled transistors is always on, thus consuming power continuously. The major advantage of static RAM is that it does not need to be refreshed.

As with most other digital devices, it is easy to connect semiconductor memory devices together because they have built-in supporting electronics. However, the timing constraints of the input signals are critical and the timing requirements vary from one device to another. To ensure proper operation, the control logic on a memory board must provide address inputs and control signals that satisfy the timing parameters as specified by the manufacturer of the memory devices being used. The input timing during a memory read operation is different from the input timing during a memory write operation.

The most important timing parameter to be considered in choosing a memory device is the *access time*. The maximum time delay from an address input to a data output is longer than the delay between a chip enable and a data output, and consequently the former timing figure is normally considered to be the access time. The access time for commonly used MOS RAMs varies from 50 to 500 ns.

For a read operation, once the output data are valid, the address input cannot

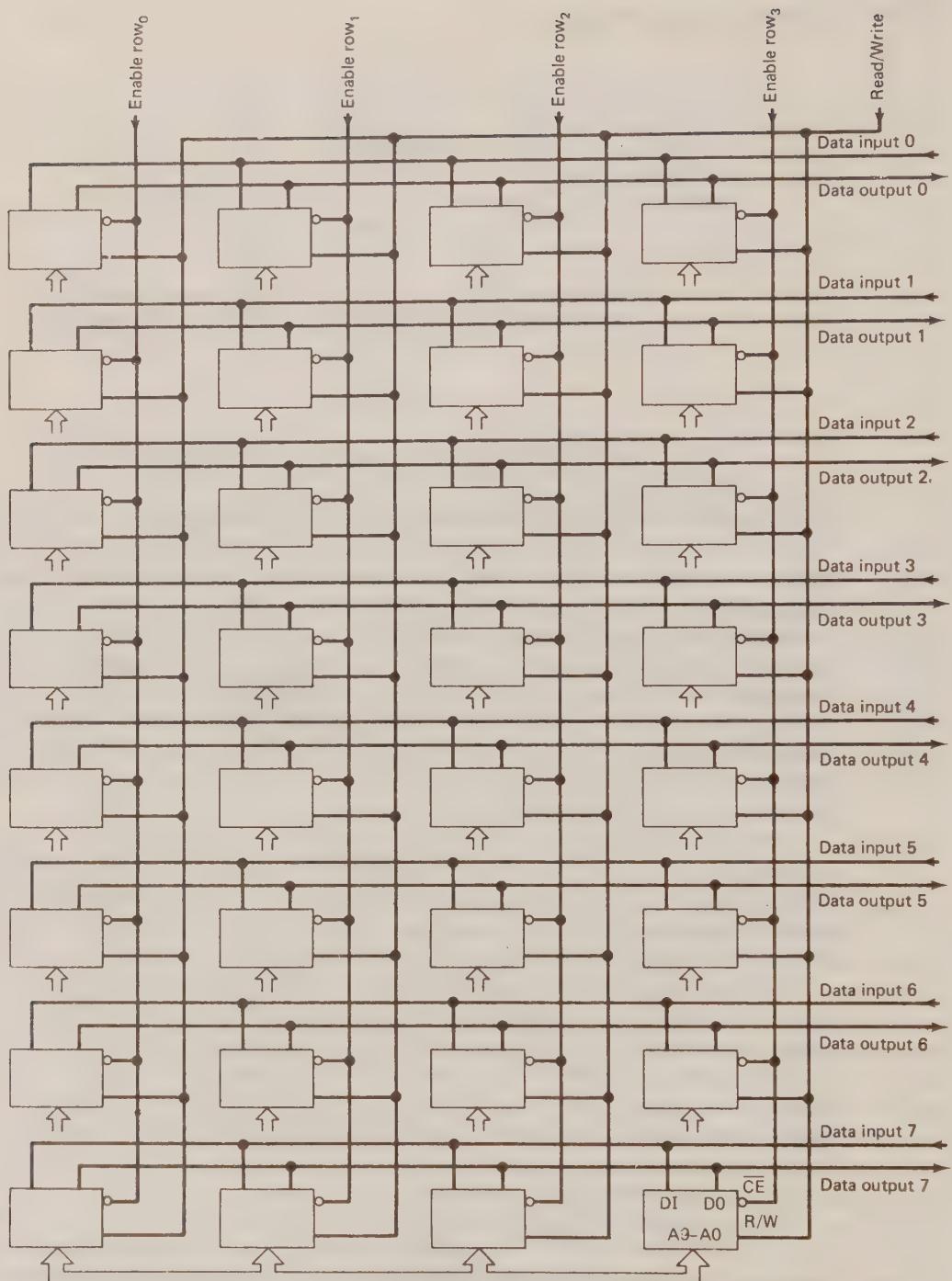
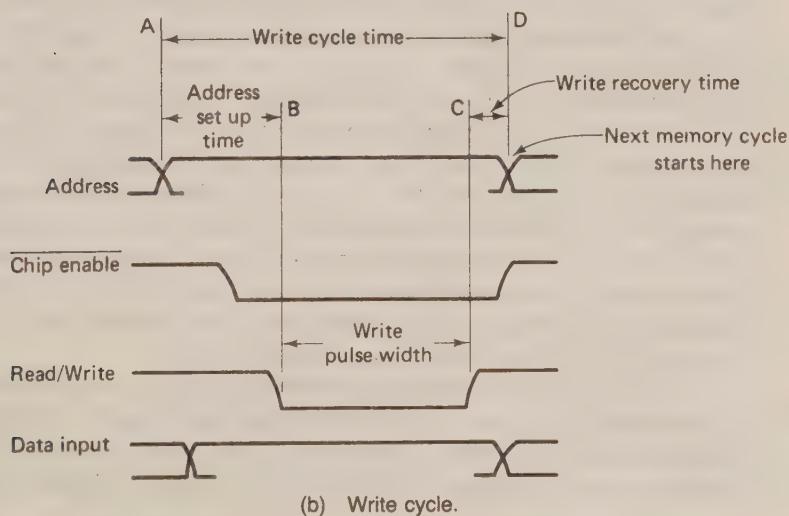
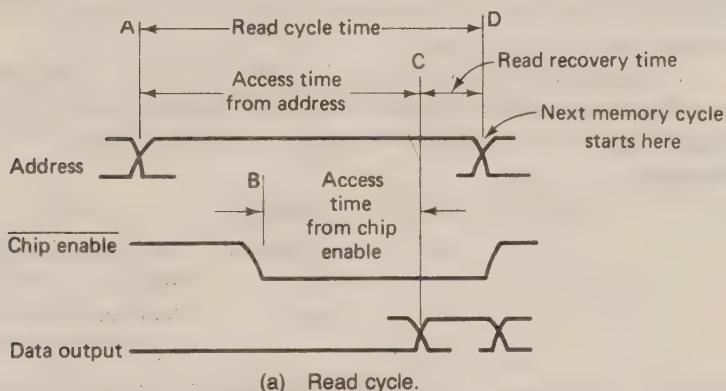


Figure 10-6 4K × 8 memory device array constructed from 1K × 1 devices. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

be changed immediately to start another read operation. This is because the device needs a certain amount of time, called *read recovery time*, to complete its internal operations before the next memory operation. The sum of the access time and read recovery time is the *memory read cycle time*. This is the time needed between the start of a read operation and the start of the next memory cycle. The *memory write cycle time* can be similarly defined and may be different from the read cycle time. Figure 10-7(a) illustrates the timing of a memory read cycle. The address is applied at point A, which is the beginning of the read cycle, and must be held stable during the entire cycle. In order to reduce the access time, the chip enable input should be applied before point B. The data output becomes valid after point C and remains valid as long as the address and chip enable inputs hold. The R/W control input is not shown in the timing diagram for the read cycle, but should remain high throughout the entire cycle.

Figure 10-7 Timing for memory reference cycles. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.



A typical write cycle is shown in Fig. 10-7(b). In addition to the address and chip enable inputs, an active low write pulse on the R/W line and the data to be stored must be applied during the write cycle. The timing of data input is less restrictive and can be satisfied simply by holding the data input stable during the entire cycle. However, the application of the write pulse has two critical timing parameters: the address setup time and the write pulse width. The *address setup time* is the time required for the address to stabilize and is the time that must elapse before the write pulse can be applied. In Fig. 10-7(b) the address setup time is the time interval between points A and B. The *write pulse width* defines the amount of time that the write input must remain active low. The *write cycle time* is the time interval between points A and D and is the sum of address setup time, write pulse width, and write recovery time. Both the read and write recovery times may be zero for some memory devices.

It is important to note that the access time and cycle time discussed in this section are the minimum timing requirements for the memory devices themselves. The access time and cycle time for the memory system as a whole are considerably longer because of the delays resulting from the I/O control logic, system bus logic, and memory interface logic.

An example of the design of a $16K \times 8$ static RAM memory module for a maximum mode 8088 is shown in Fig. 10-8. It is assumed that the \overline{CE} and \overline{WE} inputs, and the D7-D0 pins of the $4K \times 8$ static RAM, have the following relationships:

| \overline{CE} | \overline{WE} | Function | D7-D0 |
|-----------------|-----------------|--------------|--|
| 1 | X | Not selected | High-impedance state |
| 0 | 1 | Read | Data put on lines |
| 0 | 0 | Write | High impedance and data are taken from lines |

where 1 is high and 0 is low. When a memory device is not selected (i.e., \overline{CE} is high) it will go into a standby state which permits it to operate at reduced power.

The incoming address bus splits into two parts, with lines A19-A14 being used to select the module. A13 and A12 are input to the chip enable logic, which is detailed in Fig. 10-9(a). The chip enable logic has four outputs, \overline{CE}_0 through \overline{CE}_3 , only one of which may be active at any one time. \overline{CE}_0 is connected to the Chip Enable (\overline{CE}) input of the memory device associated with the lowest 4K addresses and is active when $A_{13} = A_{12} = 0$. Similarly, \overline{CE}_1 is active when $A_{13} = 0$ and $A_{12} = 1$, \overline{CE}_2 is active when $A_{13} = 1$ and $A_{12} = 0$, and \overline{CE}_3 is active when $A_{13} = A_{12} = 1$. In all cases the module select line must be 1 before a \overline{CE} output can be active. The module select line is active only when the module is selected and a memory read or write is recognized. The address lines A11-A0 are applied to the A11-A0 pins of all of the memory devices.

MWTC and the module select line are input to a write pulse generator which is constructed from two monostable multivibrators and is detailed in Fig. 10-9(b). The output of this circuit is connected to the Write Enable (\overline{WE}) pins of all the memory devices and causes the data on D7-D0 to be put into the addressed byte.

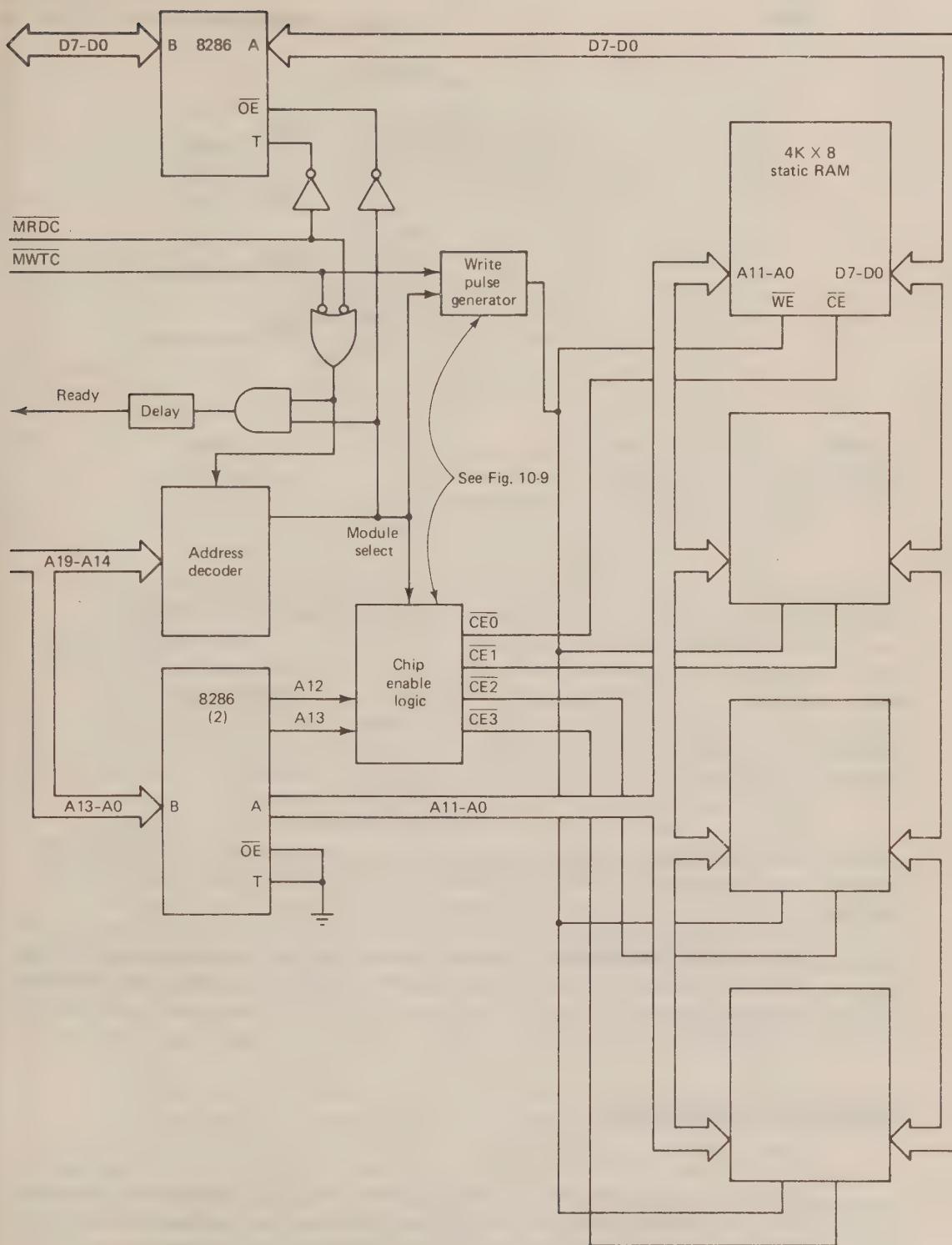
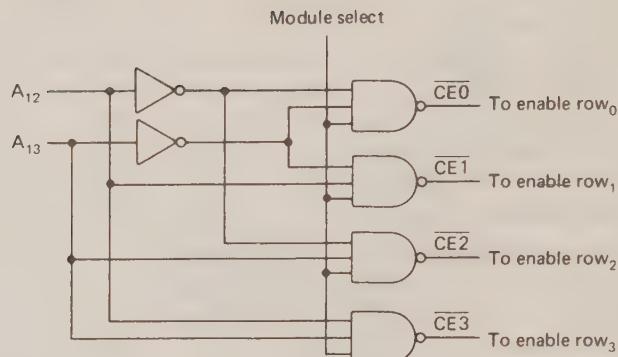
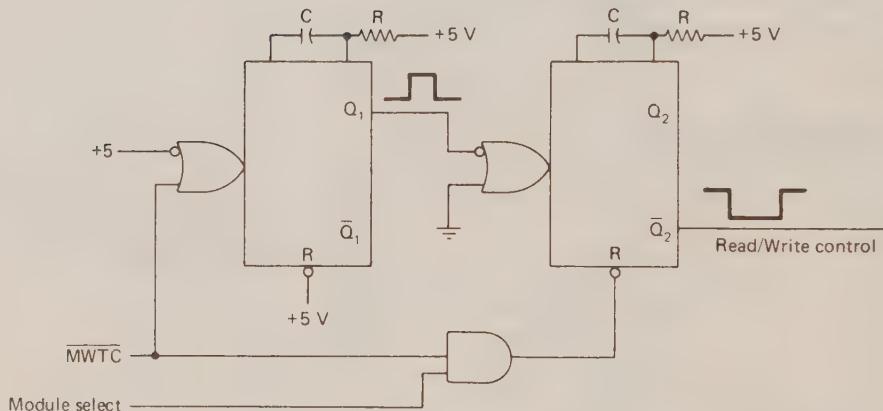


Figure 10-8 16K × 8 memory module for a maximum mode 8088.



(a) Chip enable logic



(b) Write pulse generator

Figure 10-9 Support logic for the memory module design in Fig. 10-8. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

Note that in the device array shown in Fig. 10-6 there are separate data-in and data-out lines emerging from each device, but in Fig. 10-8 the data lines are bidirectional. Internally, there must be separate lines for reading from and writing to the cells, but it is possible to separate the bidirectional signals into read and write signals inside the device:

10-3 DYNAMIC RAM DEVICES

Just like static RAMs, the memory on a dynamic memory chip is organized in a matrix formed by rows and columns of memory cells. The simplest type of dynamic RAM cell contains only one transistor and one capacitor, as shown in Fig. 10-10. Whether a 1 or 0 is contained in a cell is determined by whether or not there is a

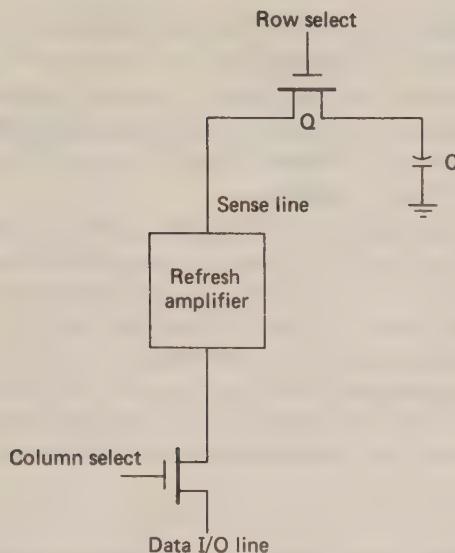


Figure 10-10 Typical one-transistor dynamic RAM cell. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

charge on the capacitor. During a read operation, one of the row select lines is brought high by decoding the row address (low-order address bits). The activated row select line turns on the switch transistors Q for all cells in the selected row. This causes the refresh amplifier associated with each column to sense the voltage level on the corresponding capacitor and interpret it as a 0 or a 1. The column address (high-order address bits) enables one cell in the selected row for the output. During this process, the capacitors in an entire row are disturbed. In order to retain the stored information, the same row of cells is rewritten by the refresh amplifiers. A write operation is done similarly except that the data input is stored in the selected cell while the other cells in the same row are simply refreshed.

As a result of the storage discharge through pn-junction leakage current, dynamic memory cells must be repeatedly read and restored; this process is called *memory refresh*. The storage discharge rate increases as the operating temperature rises, and the necessary time interval between refreshes ranges from 1 to 100 ms. When operating at 70°C, the typical refresh time interval is 2 ms. Although a row of cells is refreshed during a read or write, the randomness of memory references cannot guarantee that every word in a memory module is refreshed within the 2-ms time limit. A systematic way of accomplishing a memory refresh is through memory refresh cycles.

In a *memory refresh cycle*, a row address is sent to the memory chips, and a read operation is performed to refresh the selected row of cells. However, a refresh cycle differs from a regular memory read cycle in the following respects:

1. The address input to the memory chips does not come from the address bus. Instead, the row address is supplied by a binary counter called the *refresh address counter*. This counter is incremented by one for each memory refresh

- cycle so that it sequences through all the row addresses. The column address is not involved because all elements in a row are refreshed simultaneously.
2. During a memory refresh cycle, all memory chips are enabled so that memory refresh is performed on every chip in the memory module simultaneously. This reduces the number of refresh cycles. In a regular read cycle, at most one row of memory chips is enabled.
 3. In addition to the chip enable control input, normally a dynamic RAM has a data output enable control. These two control inputs are combined internally so that the data output is forced to its high-impedance mode unless both control inputs are activated. During a memory refresh cycle, the data output enable control is deactivated. This is necessary because all the chips in the same column are selected and their data outputs are tied together. On the other hand, during a regular memory read cycle, only one row of chips is selected; consequently, the data output enable signal to each row is activated.

Consider a memory module of 16K bytes implemented by $4\text{K} \times 1$ dynamic RAMs. The memory device array consists of four rows and eight columns. Each chip has 64 rows and 64 columns of memory cells and has separate row address (6 bits) and column address (6 bits) pins. It is assumed that the chip enable and output enable pins are CE and CS, respectively. The block diagram in Fig. 10-11 shows the logic needed to generate the chip enable and the refresh address signals during a memory refresh. The memory refresh cycle signal, which indicates that a refresh is in progress, can be generated by a refresh cycle timing generator in the memory module. If the current cycle is a refresh cycle, the 2-to-1 multiplexer selects the row address from the refresh address counter; otherwise, the row address comes from the address bus. Assuming that every cell in the chip array must be refreshed within a 2-ms time interval, a refresh cycle is required for every $2 \times 10^{-3}/64 = 31.25 \mu\text{s}$. At the end of each refresh cycle, the binary counter is incremented by 1 so that it points to the next row to be refreshed. During a refresh cycle, all chips on the module are enabled for performing a read operation by activating the chip enable signals. Their data outputs are forced into the high-impedance state by deactivating the output enable signals.

In addition to requiring refresh logic, a chief disadvantage in using dynamic RAMs is that during a refresh cycle the memory module cannot initiate a read or write cycle until the refresh cycle has been completed. As a result, a read or write request will take up to twice as long to complete if a refresh is in progress. Assuming a cycle time of 400 ns (for all memory cycles—refresh, read, write),

$$\frac{64 \times 400 \times 10^{-9}}{2 \times 10^{-3}} \times 100\% = 1.28\%$$

of the memory time is spent on refresh.

There are several reasons why dynamic RAMs are attractive to memory designers, especially when the memory is large. Three of the main reasons are:

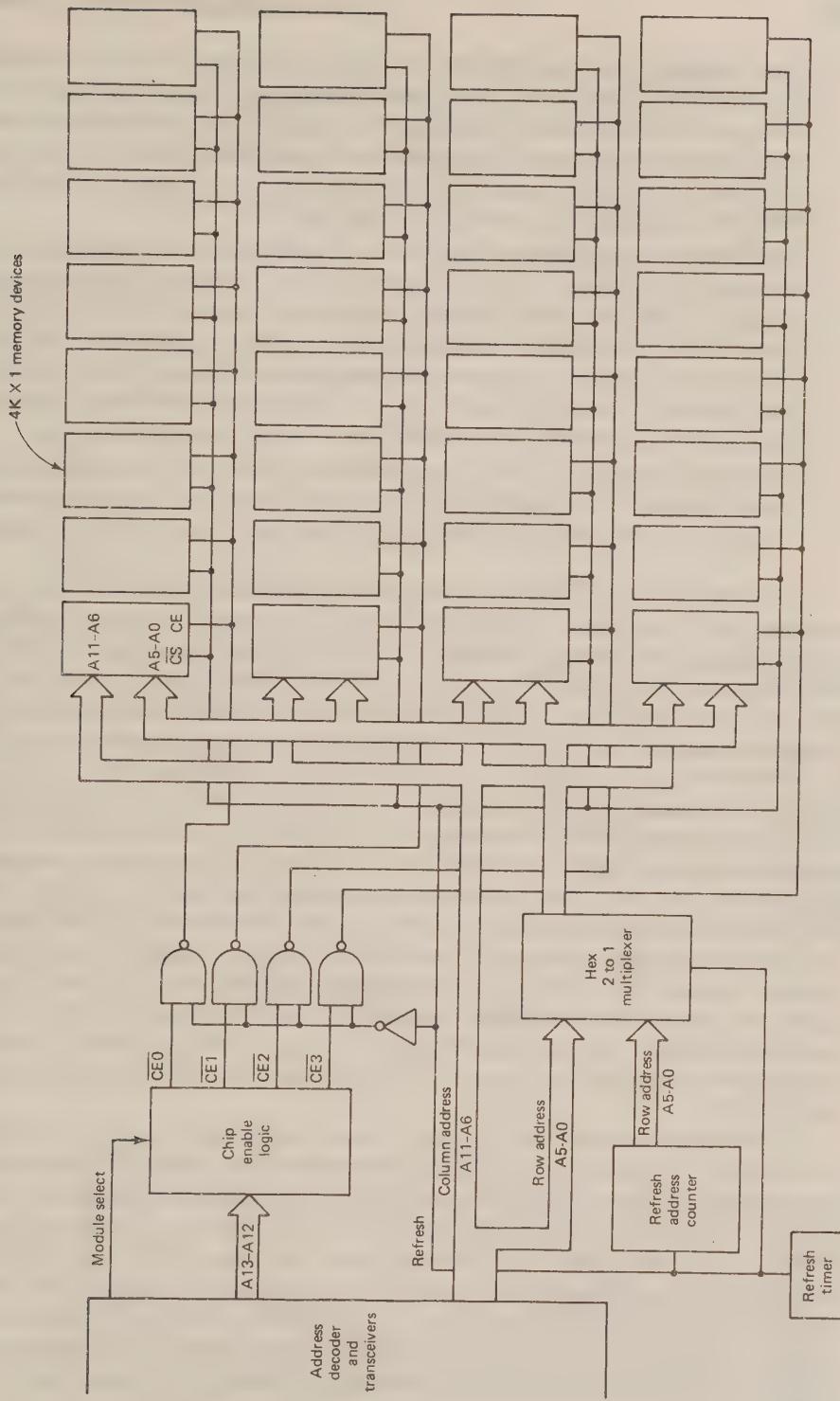
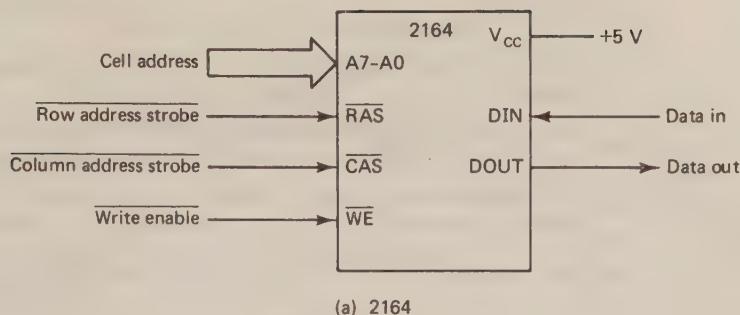


Figure 10-11 Refresh address logic for a 16K-byte RAM implemented using $4K \times 1$ dynamic RAM devices.

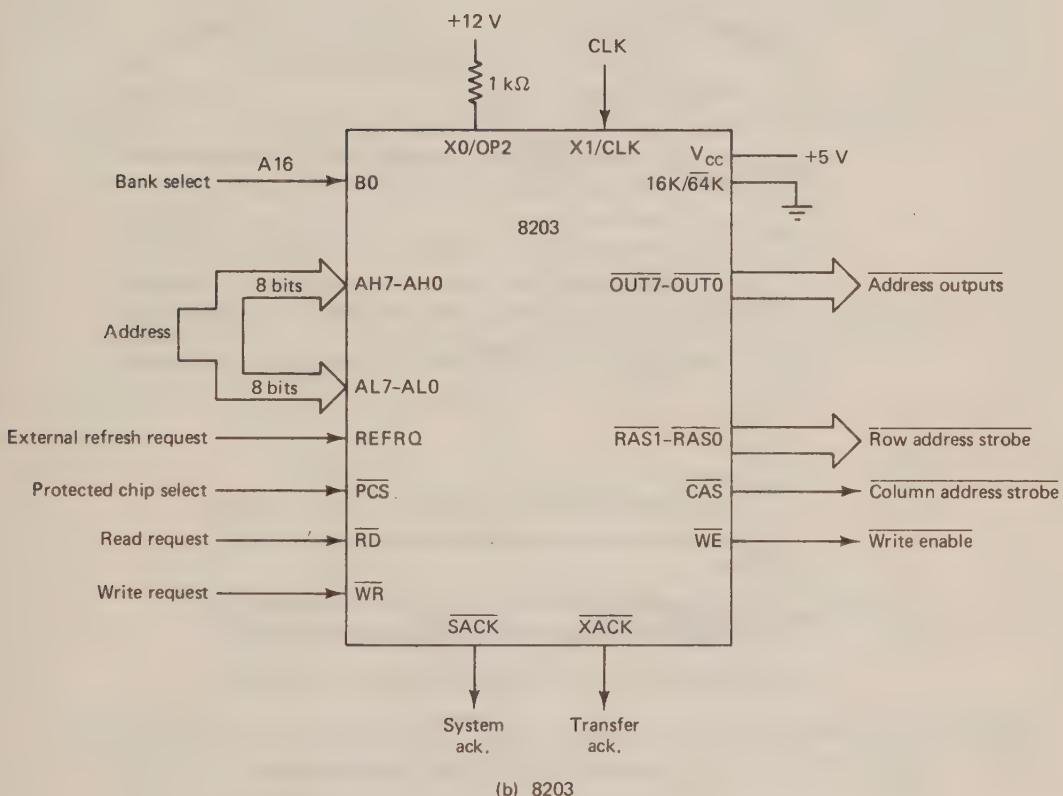
1. **High Density**—For static RAM, a typical cell requires six MOS transistors. The structure of a dynamic cell is much simpler and can be implemented with three, or even one MOS transistor. As a result, more memory cells can be put into a single chip and the number of memory chips needed to implement a memory module is reduced. A common size for a dynamic RAM chip is $16K \times 1$, and $64K \times 1$ devices are also available.
2. **Low Power Consumption**—The power consumption per bit of dynamic RAM is considerably lower than that of static RAM. The power dissipation is less than 0.05 mW per bit for dynamic RAM and typically 0.2 mW per bit for static RAM. This feature reduces the system power requirements and lowers the cost. In addition, the power consumption of dynamic RAM is extremely low in standby mode; this makes it very attractive in the design of memory that is made nonvolatile through the use of a backup power source.
3. **Economy**—Dynamic RAM is less expensive per bit than static RAM. However, dynamic RAM requires more supporting circuitry, and, therefore, there is little or no economic advantage when building a small memory system.

For higher-density RAM, the row address and column address normally share the same set of pins, thus reducing the device's pin count. For these RAMs some manufacturers have produced single ICs that include the refresh support logic and other logic needed in controlling the row/column address pins. Toward this end, Intel has made available its 8203 dynamic RAM controller, which is specifically designed to support its 2117, 2118, and 2164 dynamic RAM memory devices. Here we will concentrate on the 8203's use with the 2164, which is a $64K \times 1$ device. The block diagrams for the 2164 and 8203 are shown in Fig. 10-12. (The 8203 can be put in one of two modes according to its $16K/64K$ pin; only the assignments for the $64K$ mode are shown.) The 2164 has four 128×128 cell arrays but only eight address pins, A7–A0. This means that the row address and the column address must share the same pins and be received one after the other. The row address is strobed by a negative-going pulse on the $\overline{\text{RAS}}$ pin and, with $\overline{\text{RAS}}$ still low, the column address is strobed by a negative-going pulse on the $\overline{\text{CAS}}$ pin. The most significant row and column address bits specify one of the four cell arrays. During a memory refresh cycle, the address input A7 is not used and all four cell arrays perform their refreshes simultaneously. This allows the entire device to be refreshed in 128 cycles.

The timing diagrams for the read, write, and refresh-only cycles are shown in Fig. 10-13. For a read cycle, $\overline{\text{WE}}$ must be inactive before the $\overline{\text{CAS}}$ pulse is applied and remain inactive until the $\overline{\text{CAS}}$ pulse is over. After the column address is strobed, $\overline{\text{RAS}}$ is raised and with $\overline{\text{RAS}}$ high and $\overline{\text{CAS}}$ low the data bit is made available on DOUT. For a write cycle the DIN signal should be applied by the time $\overline{\text{CAS}}$ goes low, but after the $\overline{\text{WE}}$ pin goes low. The write is performed through the DIN pin while $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and $\overline{\text{WE}}$ are all low. The DOUT pin is held at its high-impedance state throughout the write cycle. For the refresh-only cycle only the row address is strobed and the $\overline{\text{CAS}}$ pin is held inactive. The DOUT pin is kept in its high-impedance state.



(a) 2164



(b) 8203

Figure 10-12 Pin assignments for the 2164 dynamic RAM and its associated 8203 controller.

The 8203 is designed to output signals whose timing meets the 2164 requirements. The OUT7-OUT0 lines provide the properly sequenced row and column addresses, RAS1-RAS0 provide the row address strobes for up to two banks of 2164s, and CAS and WE supply the column address strobe and write enable signals for all of the 2164s in the module. (Note that the addresses output by the 8203 are

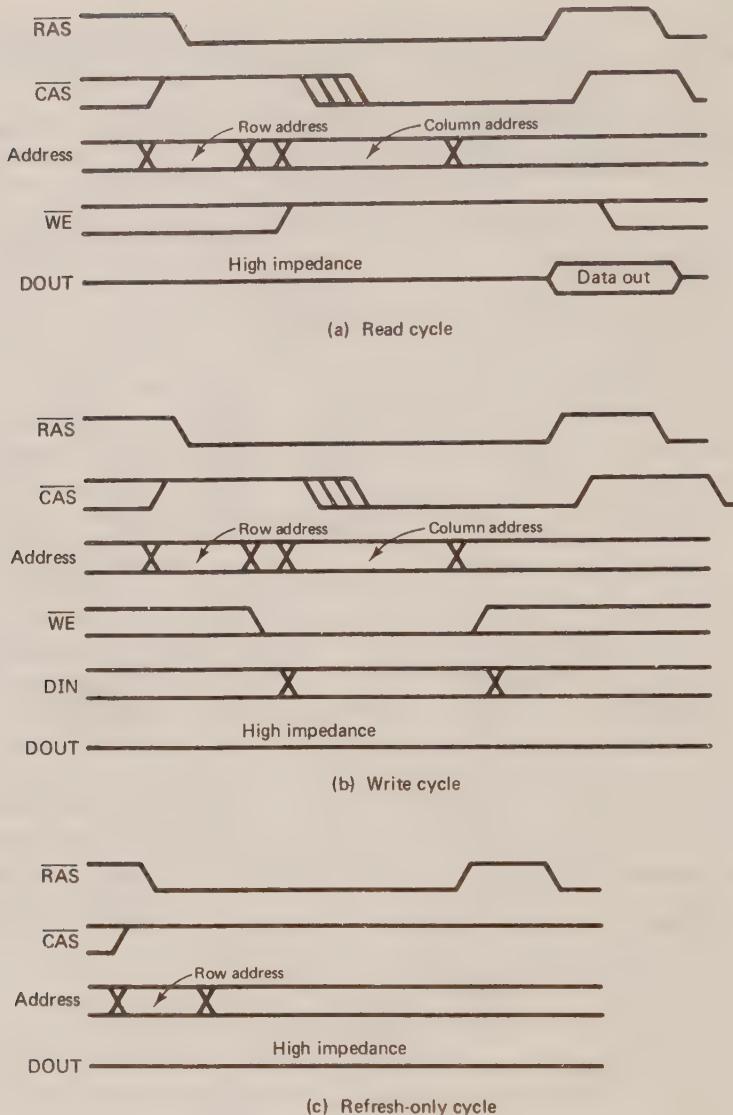


Figure 10-13 Timing diagrams for the 2164.

inverted. This does not cause a problem, but it does mean that all 0s on the address lines will access the cells having row and column addresses that are all 1s.)

The bank select input B0 determines the $\overline{\text{RAS}}$ pin to be activated. AL7–AL0 are used to generate the row address and AH7–AH0 the column address. Normally, the refresh cycle timing is generated inside the 8203, but the REFRO pin allows the refresh cycles to be initiated by an external source. The module selection is made through the $\overline{\text{PCS}}$ pin. It is called the protected chip select pin because once

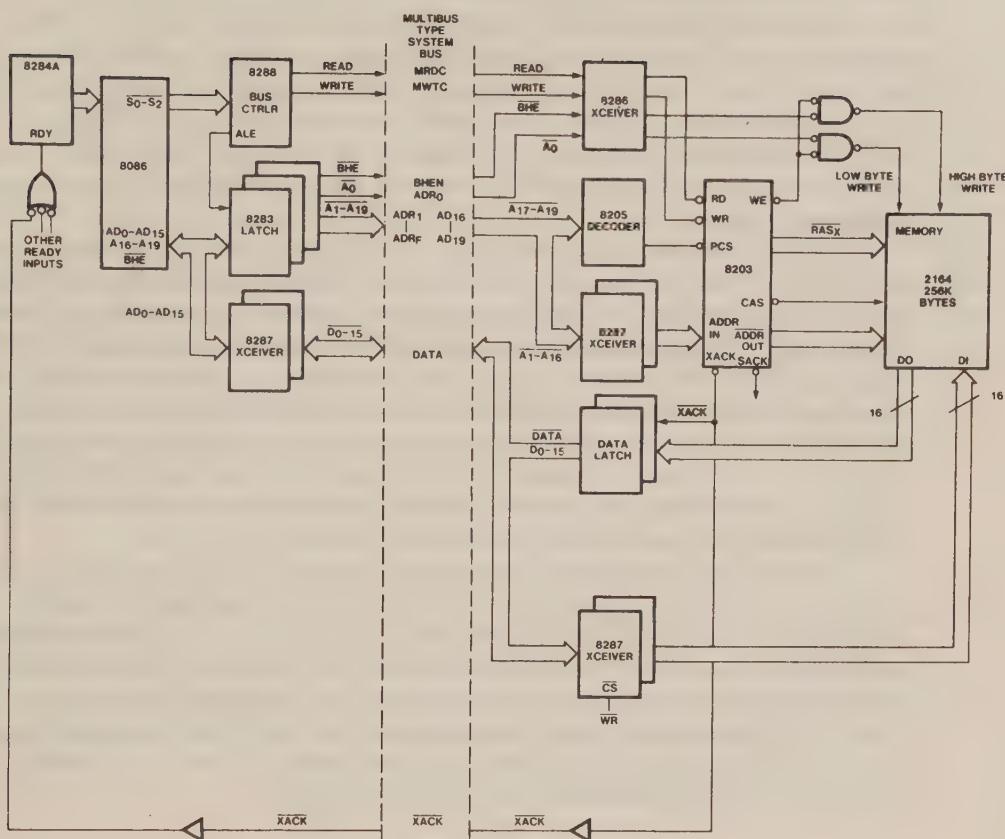


Figure 10-14 Maximum mode 8086, MULTIBUS system with a 256K-byte memory module.
(Reprinted by permission of Intel Corporation. Copyright 1982.)

it becomes active the memory cycle cannot be aborted, even if it immediately returns to its inactive state. The RD and WR inputs specify whether a memory read or write is to be conducted.

The XACK output is a strobe indicating that data are available during a read cycle or that the data have been written during a write cycle. It can be used to strobe data into data output latches and to send the ready signal to the processor. The SACK output signals the beginning of a memory access cycle and if a refresh cycle is taking place when a memory request is made, the SACK signal is delayed until the read or write cycle begins. If the memory device access time is known to be sufficiently low to guarantee that a read will be complete by the end of the T₃ clock cycle or a write will be complete by the end of the T₄ cycle, the SACK output

may be used as the ready signal instead of the \overline{XACK} output, thus saving wait states that might occur if \overline{XACK} is used.

Either an oscillator must be connected across X0 and X1 or, if OP2 is connected to +12 V, an external clock signal must be applied to CLK. This signal may come from a bus clock line or a clock included in the memory module. Only +5 V is needed for the main power supply, but if the OP2 input is used, a +12-V voltage is required. (REFRQ is actually a dual-purpose pin that can be used for advanced reads, but this feature as well as the read-modify-write cycle will not be considered here.)

Figure 10-14 illustrates how an 8203 and thirty-two 2164s could be used to construct a 256K-byte memory module for a maximum mode 8086 MULTIBUS system. This is an Intel design which assumes that the data and address buses are inverted and therefore uses 8283s and 8287s to interface to these buses instead of 8282s and 8286s. The memory device array has 16 columns so that words can be accessed. This requires that the \overline{BHE} and A0 bus lines be used in conjunction with the \overline{WE} signal to determine whether only the low-order byte, only the high-order byte, or an entire word is being written. (Figure 10-14 was taken from Reference 1, which also includes, on page 9-88, the design of a minimum mode 8088 system with an 8203/2118-based 64K-byte memory module.)

Another way of reducing the number of chips needed in the support circuitry for dynamic RAM is to put a set of refresh logic on each memory device, thus permitting the device to refresh itself. Such a device is called an integrated RAM and, except for memory accesses sometimes being held up by refresh cycles, the device appears to the user to be a static RAM. An example of this approach is the Intel 2186/7, which is an $8K \times 8$ integrated RAM. The 2186/7 has pin assignments that are essentially the same as for Intel's static RAM devices. In particular, it has \overline{OE} , \overline{WE} , and \overline{CE} pins which serve the same purposes.

10-4 BACKUP POWER FOR SEMICONDUCTOR MEMORIES

One major disadvantage of using MOS RAMs to construct main memory is that the stored information may be lost as a result of even very short power failures. The solution is to provide a backup power supply which will support the system if the main supply fails. Because of the cost, normally only part of the memory system is protected from power failure. During a power failure, the status and vital data within the program being executed can be stored in the nonvolatile memory modules; then by restoring this information after the power has returned, the program is able to continue.

Some MOS RAMs consume much less power while they are just maintaining information than when they are executing read/write operations. To reduce the drain on the backup power supply during a power failure, the memory module can be forced into a standby mode in which all memory chips are disabled and the

stored data are simply retained. Because of this feature, batteries become a practical way to provide backup power for MOS memory over reasonable lengths of time.

Figure 10-15 shows a power supply with a battery backup. During normal operation, the power is provided by a power supply which converts an ac line voltage into a regulated dc voltage that is maintained at V_{CC} . The output of the battery is lower than the normal V_{CC} , and consequently diode D1 is forward-biased and diode D2 is cut off. If the main power fails, the capacitor discharges until its voltage is lower than the output of the battery. At this time, D2 is forward-biased and the battery supplies the power to the memory. Once the main supply is restored, D2 is cut off and the battery is recharged. Another circuit that is commonly used to switch power between the main supply and the battery is one that includes a relay. The relay is such that power comes from the main supply during normal operation and from the battery during power failure. The relay is controlled by a power-loss-detection circuit.

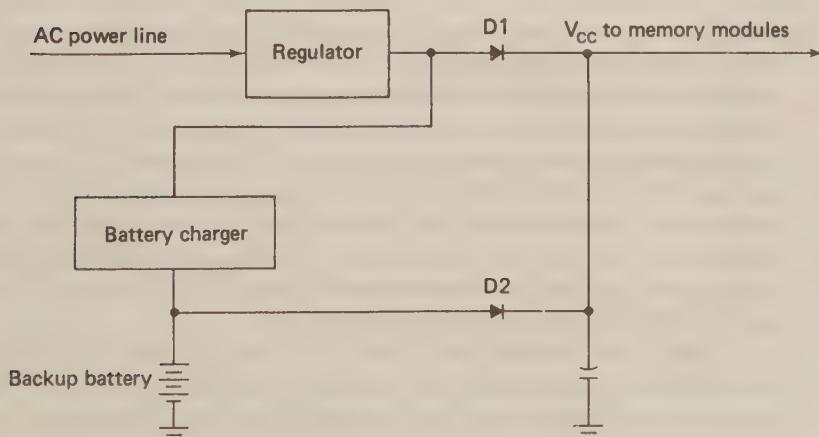
The type and numbers of batteries required in the backup supply are determined by the following factors:

1. The supply current required by the memory modules.
2. The battery discharge characteristics.
3. The size, weight, and cost of the batteries.
4. The maximum length of time the memory must be supplied by backup batteries.

Because a memory module consists of a memory chip array and supporting logic, the total discharge current requirement can be calculated by

$$\text{Discharge current} = \frac{N_m \times P_m + P_s}{V}$$

Figure 10-15 Battery backup power system. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.



where

N_m = number of memory chips

P_m = power dissipated by each memory chip

P_s = total power dissipated by the supporting logic

V = supply voltage

The required supply current is significantly less if the memory is forced into standby mode during a power failure. Another way to reduce standby power is by using low-power-dissipation CMOS devices for the memory interface and control logic.

The capacity of a battery is rated in terms of ampere-hours at a specific discharge current. However, the ampere-hour rating tends to decrease as the discharge current rating increases. The protection time that can be provided by a backup power source is the ratio of the ampere-hour rating to the discharge current, provided that the discharge current is less than or equal to its rating. For example, assume that a battery has a capacity of 3.2 A·h at a 1-A discharge current and that a memory module draws 0.8 A. Then the battery can supply the power to the memory module for at least 4 h. Three batteries in parallel will be able to provide protection for at least 12 h.

A desirable feature for a backup battery is a nearly constant output voltage during discharge. Batteries that satisfy this criterion are commercially available in both the rechargeable and nonrechargeable types. Examples of the nonrechargeable type are mercury and silver oxide batteries, which offer large capacity, yet small size. Nickel-cadmium and lead-calcium batteries are widely used rechargeable batteries. Although they are considerably larger and heavier than most batteries, the fact that they can be recharged whenever main power is restored may be required in some applications.

10-5 ROM DEVICES

ROM is such that once its contents are set they cannot be changed, at least not without the aid of special equipment. Its cell density can be higher than that of read/write memory because its construction is simpler (there is no need for write circuitry). It is also nonvolatile and reliable. However, it is restricted to uses where writing is not necessary. As noted in the introduction to this chapter, most memory systems contain both ROM and RAM modules, with the ROM modules being used to store such things as bootstrap loaders and permanent data tables. ROM is also sometimes used to hold the resident monitor and other suitable parts of the operating system (thus obviating the need for a bootstrap loader). It may even contain permanently installed language interpreters.

There are basically four types of ROMs, the four types being distinguished by the way in which their contents are set. Setting the contents of a ROM is sometimes called programming, but should not be confused with the generation of

instruction sequences discussed in the previous chapters. In one type of ROM the contents are determined by a masking operation that is performed while the chip is being manufactured. Such chips cannot be altered by the user and are referred to simply as ROMs. The contents of the second type can, if the proper equipment is available, be set by the user. They are called *programmable-read-only memories* (PROMs). As with the masked ROMs, once a memory of this type is programmed, its contents can never be changed. The third and fourth types can not only be programmed by the user, but also by using special equipment can be erased and reprogrammed many times. They are called *erasable-programmable-read-only memories* (EPROMs) and *electrically-erasable-read-only memories* (E^2ROMs).

Because a masked ROM requires the manufacturer to first produce a mask according to a given set of contents, or *program*, it is expensive to produce the first ROM, but additional copies are inexpensive. Therefore, ROMs are used only after the development stage of a computer system is complete, when it is not likely that the contents of the ROMs will need to be changed. The only pins required on a ROM device are a set of address input pins, a set of data output pins, and a chip select pin. Figure 10-16 shows how two $2K \times 8$ ROMs could be used to build a $4K \times 8$ byte storage module. In addition to the logic shown, the usual module select address decoder, transceivers, and latches would be required.

Normally, PROMs are constructed of diode matrices. They are programmed by using external pins to select the diode links that are to be “burned” or “blown,” thus causing the diode matrix to be permanently programmed. The 1s are represented by the diode links left intact and the 0s by the diode links that are blown. Because the PROMs for all customers are identical when they are shipped, their price is not as dependent on quantity, but their circuitry is more complex and expensive due to their programmability. Therefore, for a few devices PROMs are cheaper than masked ROMs, but for large-quantity orders masked ROMs are cheaper. As a result, PROMs (or EPROMs) are used during development, but masked ROMs are used in the final mass-produced products.

PROMs are programmed bit by bit by placing the address of the location containing the bit on the address pins and forcing a current *into* the bit's data output pin while the voltage supply and appropriate control pins are pulsed. The magnitude of the current, the magnitudes and width of the pulses, and the manner in which the pulses are applied varies from one PROM to the next. Generally, the programming cycles are interspersed with verification cycles and the cycles are alternated for approximately twice the time needed to program the bit.

As opposed to a PROM, in which the contents are permanently programmed by “burning” out diode links, the contents of an EPROM are determined by charge distribution. EPROMs are programmed by charge injection, and once programmed the charge distribution is maintained until it is disturbed by some external energy source such as an ultraviolet light. Instead of completely encasing an EPROM chip in a protective package as is done with other ICs, an EPROM is constructed by putting a quartz window over its circuitry, which will allow the external energy to pass. By exposing the memory to the external energy source for several minutes (10 to 50 minutes, depending on the specific device), the charge is redistributed

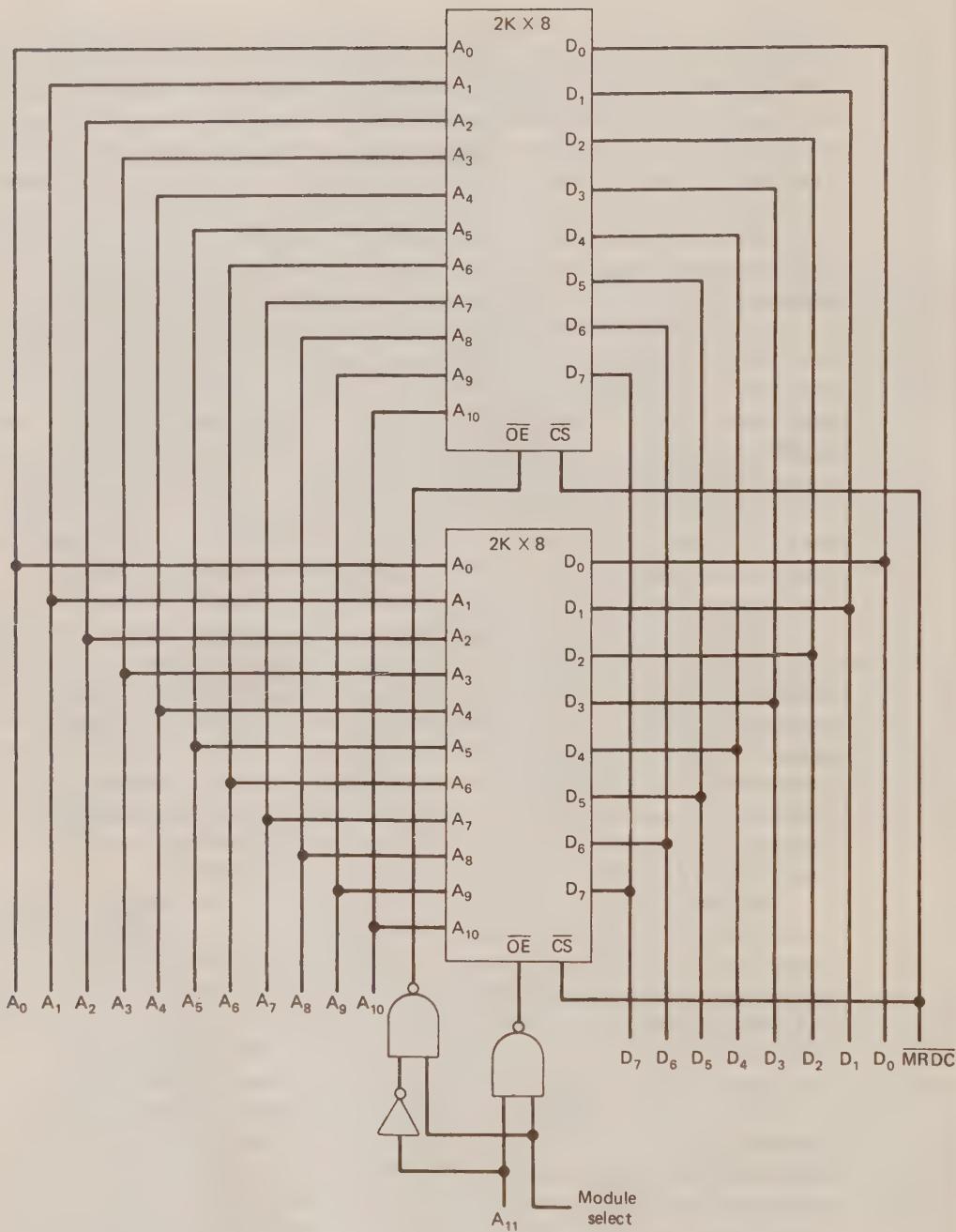


Figure 10-16 4K × 8 ROM memory. Glenn A. Gibson, Yu-Cheng Liu, MICROCOMPUTERS FOR ENGINEERS AND SCIENTISTS © 1980, Prentice-Hall, Inc.

to its natural state, thus destroying the old memory contents. The EPROM can then be reprogrammed.

As with PROMs, EPROMs are normally used during the development of a product and are replaced by masked ROMs once the design is complete. Because they can be erased, they have the advantage of being reusable. However, unlike PROMs, EPROMs may lose their contents over extended periods of time and should not be put in final products that are expected to last for years. How long one can depend on an EPROM retaining its contents depends on the environmental conditions it is subjected to, and may vary from a few months to several years.

An EPROM is programmed by applying an address to the address pins and high or low voltages to *all* of the data output pins, and then applying the proper voltages and pulses to the supply and control pins. As an example, to program an Intel 2764 PROM (8K × 8), 21 V is applied to the V_{pp} pin while the \overline{CE} pin is held low. The address of the byte to be programmed is put on pins A12–A0 and the data byte is simultaneously applied to O7–O0. The data byte is then written into the addressed byte by pulsing \overline{PGM} with +5 V. The contents of each byte should be verified after it has been written.

Even PROM and EPROM memory devices made by one manufacturer may have programming signal specifications that vary considerably. Particularly in the case of PROM programming, the programming specifications may be complex and must be followed precisely. Therefore, many manufacturers have made available units, called *PROM programmers*, especially designed to program PROMs and EPROMs. To accommodate the varying electrical requirements, some programmers are built to accept circuit modules which provide customized electrical outputs. Each module is called a *personality card* and provides all the electrical signals needed for programming a given type of memory chip. In addition to personality cards for the various PROMs and EPROMs, a PROM programmer may have a control module which includes a microprocessor. The control module permits a block of data from a microcomputer development system (MDS) to be written into a PROM (or EPROM) with a single command. A programmer may also include a display for verifying the contents of a PROM. In conjunction with the Intel Universal PROM programmer, commands are available under the ISIS-II operating system for performing the following operations:

1. Loading the data to be programmed from a selected input device (disk file, paper tape, or system console) into the MDS memory.
2. Displaying or changing data in the MDS memory.
3. Programming a segment of a PROM with the data which are stored beginning at a specified address in the MDS memory.
4. Transferring a block of data in a PROM into memory so that the contents of the PROM may be examined through the system console or used to produce a duplicate PROM.
5. Transferring a block of data from a PROM into a disk file.

6. Comparing a block of data in a PROM with the contents of a segment of memory (program verification).

An E²ROM is also reprogrammable, but offers the added advantage that each individual byte can be easily erased and reprogrammed. The principal disadvantage of an E²ROM is that it tends to be more expensive.

BIBLIOGRAPHY

1. 1982 *Data Components Catalog* (Santa Clara, Calif.: Intel Corporation, 1982).
2. Gibson, Glenn A., and Yu-cheng Liu, *Microcomputers for Engineers and Scientists* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980).
3. Hall, Douglas V., *Microprocessors and Digital Systems* (New York: McGraw-Hill Book Company, 1980).
4. Givone, Donald D., and Robert P. Roesser, *Microprocessors/Microcomputers: An Introduction* (New York: McGraw-Hill Book Company, 1980).
5. Intel *Memory Design Handbook* (Santa Clara, Calif.: Intel Corporation, 1977).
6. Luecke, Gerald, Jack P. Mize, and William N. Carr, *Semiconductor Memory Design and Application* (New York: McGraw-Hill Book Company, 1973).

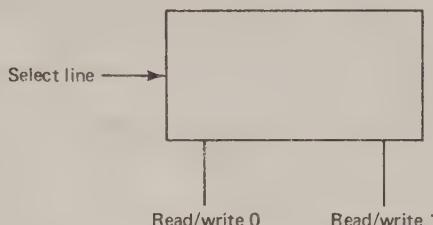
EXERCISES

1. Complete the following table:

| Memory size | Type of device | Device array | |
|-------------|----------------|----------------|-------------------|
| | | Number of rows | Number of columns |
| 4K × 8 | 2K × 4 | | |
| 64K × 8 | 32K × 1 | | |
| 8K × 16 | 4K × 4 | | |
| 1M × 16 | 64K × 1 | | |

2. For a 32K-byte memory with a single error detection (parity) bit for each byte, determine the configuration of the chip array in terms of the numbers of memory chips in each row, in each column, and in the entire module using 8K × 1 RAMs. Repeat for 16K × 1 RAMs.
3. Construct a table that summarizes the discussion of the major design considerations given in Sec. 10-1.
4. Draw a logic diagram that shows the details of how the block diagram in Fig. 10-5 could

be implemented for a two-row by two-column cell array. The following symbol is to be used to represent a cell:



5. Show how the design in Fig. 10-8 could be simplified if the total memory capacity were only $8K \times 8$. Show how the design could be further simplified if a single-board minimum mode 8088 were assumed.
6. Consider a $256K \times 8$ memory module constructed of $32K \times 1$ dynamic RAM devices with 128 rows of cells. If the entire memory is to be refreshed once each millisecond, what would be the period of the row refresh cycles? Plot the percent of time required for memory refresh versus the memory cycle time as the memory cycle time increases from 100 ns to 800 ns.
7. Redesign the logic shown in Fig. 10-11 assuming that the module is to be a 64K-byte module constructed of $32K \times 1$ devices having 128 rows of cells.
8. Summarize the advantages and disadvantages of dynamic RAMs as opposed to the static RAMs.
9. Suppose that $64K \times 1$ dynamic RAM devices supplied with a +5-V supply are to be used to construct a $256K$ -byte memory module. If each device dissipates 1 W and the support logic requires 2 W, what is the total discharge current needed by the module? How many batteries would be needed to provide backup power for 8 h if the batteries are rated at 4 A-h at a discharge current of 2 A?
10. Repeat Exercise 9 assuming that backup power is needed for only 15 minutes but the batteries' rated discharge current is only 0.7 A.
11. Draw the necessary interfacing logic for the 4K-byte ROM module in Fig. 10-16.
12. One of the major applications of ROM is to replace SSI packages in implementing complex logic functions. Consider the following Boolean function:

$$\begin{aligned} f = & X_0 X_1 \bar{X}_2 X_3 X_4 X_5 X_6 X_7 \bar{X}_8 \\ & + \bar{X}_0 \bar{X}_1 X_2 \bar{X}_3 X_4 X_5 X_6 X_7 \\ & + X_2 \bar{X}_3 \bar{X}_4 \bar{X}_5 X_6 X_7 X_8 \\ & + X_0 \bar{X}_2 X_3 X_4 X_5 X_6 \bar{X}_7 \bar{X}_8 \end{aligned}$$

How many basic logic gates (i.e., inverters, two-input AND gates, and two-input OR gates) would be required to implement this function? If a 512×8 ROM is used to implement the same function and the least significant data output bit is used as the output f , what should be the contents of the ROM? (Hint: Only the least significant data bit of each word is important.) Also, discuss the propagation delay associated with each approach.

13. How many Boolean functions can be implemented by a single $2K \times 8$ ROM, and what are the restrictions on the input variables of these functions?

1

Multiprocessor Configurations

In Chap. 9 it was seen that a DMA controller could improve the system throughput by concurrently performing I/O as the CPU continued its processing. This is possible because the CPU does not utilize all of the bus cycles. An 8086 typically uses only 50 to 80 percent of the available bus time, depending on the application. A DMA controller can steal bus cycles to transfer data while only minimally affecting the processing. Also, it releases the CPU from performing the relatively slow I/O functions. Such a system is a simple case of having more than one processor in the system, with both processors operating in parallel to improve the system performance.

Although the capability of a DMA controller is rather limited, the concept of concurrent operations, which has been proved to be an effective way of improving a system's operation, can be extended to components that are somewhat more complex. If a system includes two or more components that can execute instructions simultaneously, it is called a *multiprocessing system*. The added processors could be special purpose processors which are specifically designed to perform certain tasks efficiently, or other general purpose processors. For example, due to the 8086's limited data width and its lack of floating point arithmetic instructions, it requires many instructions to perform a single floating point operation. For a system requiring a lot of computations, it would be desirable to perform such calculations with a supporting numeric data processor that is specifically designed to quickly operate on floating point numbers and numbers having larger than usual data widths. Also, it is sometimes advantageous to include in a system an I/O processor with greater capabilities than a DMA controller, one that can perform string manipulations, code conversion, character searching, and bit testing as well as the

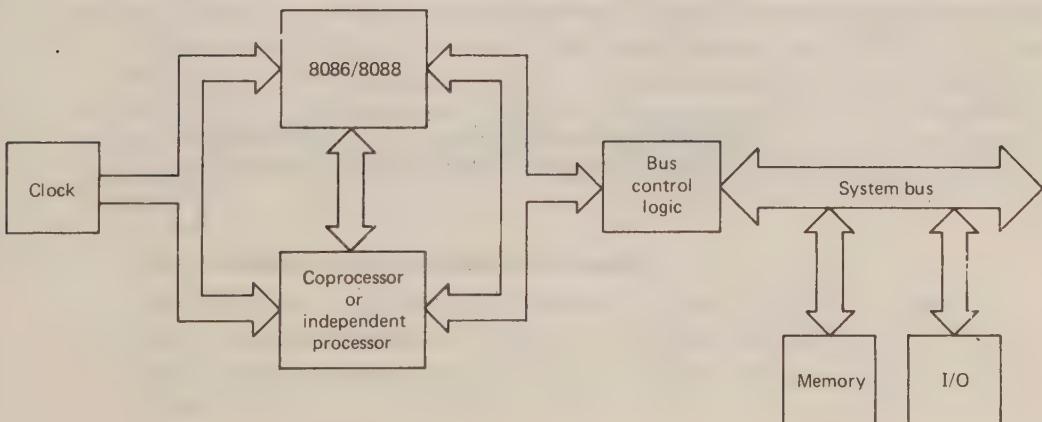
normal DMA operations. This would permit the CPU to concentrate on higher-level functions.

As the cost/performance ratio of single-chip microprocessors declines, it becomes more cost effective to use multiple processors than to use a single complex multiple-chip processor in what has come to be known as the centralized approach. In addition to improving the overall cost/performance ratio of a system, a multiprocessor configuration offers several desirable features not found in a one-processor design. First, several processors may be combined to fit the needs of an application while avoiding the expense of the unneeded capabilities of a centralized system. Yet the modularity of a multiprocessor system provides room for expansion because it is easy to add more processors as the need arises. Second, in a multiprocessor system tasks are divided among the modules. Should a failure occur, it is easier and cheaper to find and replace the malfunctioning processor than it is to find and replace the failing part in a complex processor.

Two problems must be considered in designing a multiprocessor system, bus contention and interprocessor communication. Because more than one processor shares the system memory and I/O devices through a common system bus, extra logic must be included to ensure that only one processor has access to the system bus at a time. In order for one processor to dispatch a task or return a result to another processor, an unambiguous way for the two processors to interact must be provided. How the bus contention and processor communication problems are resolved dictates the connections between the processors.

The maximum mode of the 8086 and 8088 is specifically designed to implement multiprocessor systems. Multiprocessing features are provided in maximum mode to accommodate three basic configurations. They are the coprocessor, closely coupled, and loosely coupled configurations. The first two of these configurations are very similar in that both the CPU and the supporting, or external, processor share not only the entire memory and I/O subsystem, but as shown in Fig. 11-1, they also share the same bus control logic and clock generator. In both of these con-

Figure 11-1 Closely coupled configuration.



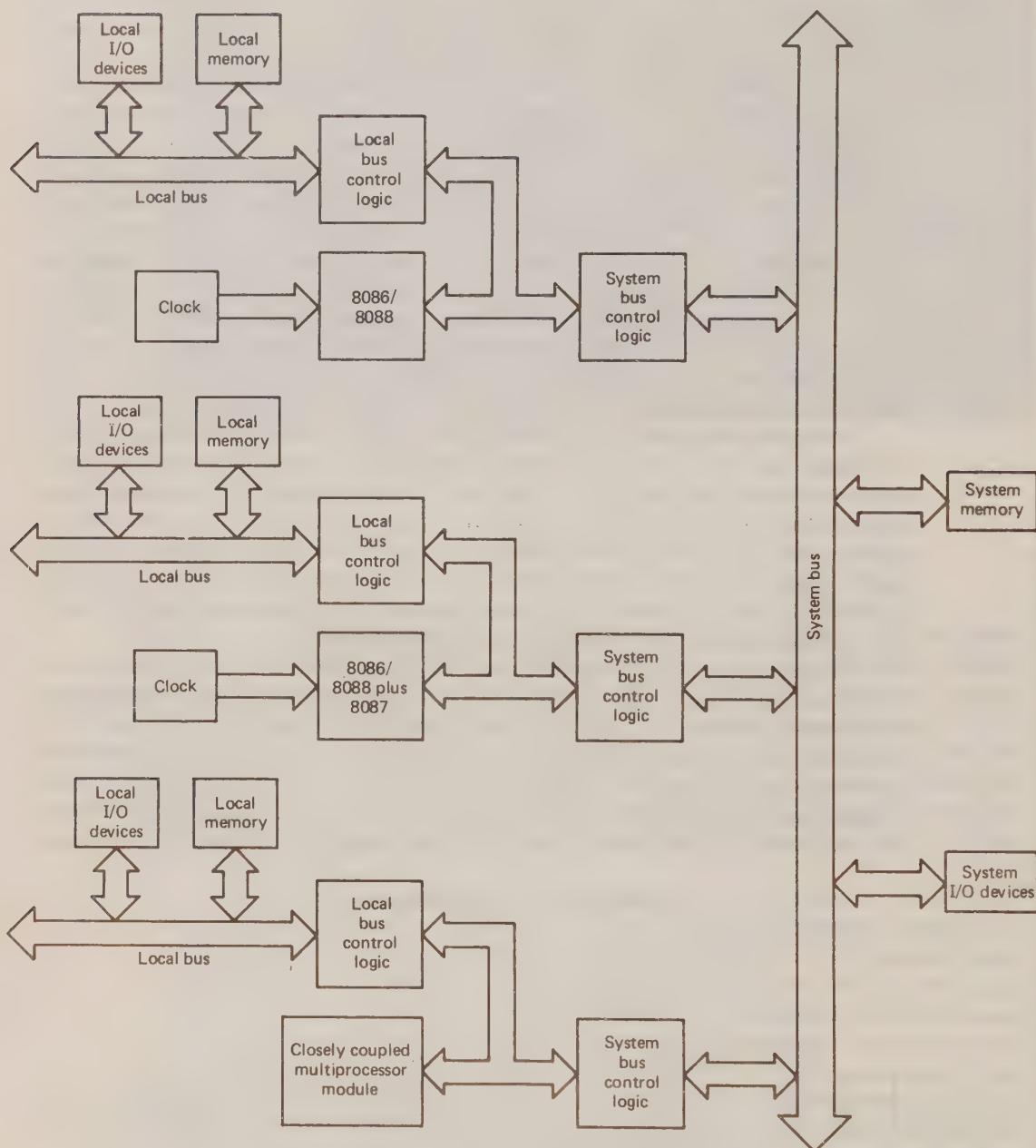


Figure 11-2 Loosely coupled configuration.

figurations, the 8086/8088 is the master, or host, and the supporting processor is the slave. The bus access control is provided by the CPU; therefore, the bus request signal from the supporting processor is connected to the CPU. In a closely coupled configuration the supporting processor may act independently, but in a coprocessor design it is dependent and must interact directly with the CPU. In a coprocessor arrangement there are more direct lines between the processing elements. Since the 8086/8088 always acts as the host in a coprocessor or closely coupled design, two 8086/8088s cannot appear in these configurations.

Loosely coupled configurations are used for medium-size to large systems. Each module in the system may be the system bus master and may consist of an 8086, an 8088, another processor capable of being a bus master, or a coprocessor or closely coupled configuration. Several modules may share the system resources, and system bus control logic must resolve the bus contention problem. As shown in Fig. 11-2, each potential bus master runs independently and there are no direct connections between them. Interprocessor communication is made possible through the shared resources. In addition to the shared resources, each module may include its own memory and I/O devices. The processors in the separate modules can simultaneously access their private subsystems through their local buses and perform their local data references and instruction fetches independently, thus improving the degree of concurrent processing.

This chapter first discusses the multiprocessing features of 8086/8088 processors, including their special instructions and control signals. Implementation of the various 8086/8088 multiprocessor configurations are examined in Sec. 11-2. The last two sections describe Intel's 8087 numeric data processor and 8089 integrated I/O processor, respectively.

1-1 QUEUE STATUS AND THE LOCK FACILITY

Although the maximum mode and the 8288 bus controller were introduced in Chap. 8, their multiprocessing features were not considered at that point. It is the purpose of this section to extend that discussion to multiprocessing situations, even though detailed discussions of their applications is left to later sections.

Because the 8086 has a 6-byte instruction queue and the 8088 has a 4-byte queue, the instruction that has just been fetched may not be executed immediately. In order to allow external logic to monitor the execution sequence, a maximum mode 8086/8088 outputs the queue status through its QS1 and QS0 pins. During each clock cycle the queue status is examined and the QS1 and QS0 bits are encoded as follows:

00—No instruction was taken from the queue.

01—The first byte of the current instruction was taken from the queue.

10—The queue was flushed because of a transfer instruction.

11—A byte other than the first byte of an instruction was taken from the queue.

By monitoring the bus and the queue status, external logic can simulate the CPU's instruction execution sequence and determine which instruction is currently being executed. This facility is necessary so that the 8086/8088 can indicate when an instruction is to be executed by a coprocessor.

As pointed out in Chap. 7, it is necessary to control the accesses to the shared resources in a multiprogramming environment. Normally, semaphores are used to ensure that at any given time only one process may enter its critical section of code in which a shared resource is accessed. Let us now reconsider the semaphore implementation:

| | | | |
|-----------|---|---|---|
| TRYAGAIN: | MOV AL,0 XCHG SEMAPHORE,AL TEST AL,AL JZ TRYAGAIN . . . | } | Critical section in which a process accesses a shared resource |
| | MOV SEMAPHORE,1 | | |

This implementation works fine for a system in which all of the processes are executed by the same processor, because the processor cannot switch from one process to another in the middle of an instruction. However, if competing processes are running on different processors, the situation is more complex.

Suppose that processor A is concurrently ready to update a record in memory while processor B is ready to sort the same record. Since both processors are running independently, they might test the semaphore at the same time. Note that the XCHG instruction requires two bus cycles, one which inputs the old semaphore and one which outputs the new semaphore. It is possible that after processor A fetches the semaphore, processor B gains control of the next bus cycle and fetches the same semaphore.

Suppose that the location SEMAPHORE contains a 1 and both processors A and B are executing

TRYAGAIN: XCHG SEMAPHORE,AL

and

1. Processor A uses the first available bus cycle to get the contents of SEMAPHORE.
2. Processor B uses the next bus cycle to get the contents of SEMAPHORE.
3. Processor A clears SEMAPHORE during the next bus cycle, thus completing its XCHG instruction.
4. Processor B clears SEMAPHORE during the next bus cycle, thus completing its XCHG instruction.

After this sequence is through, the AL registers in both processors will contain 1 and the

TEST AL,AL

instructions will cause the JZ instructions to fail. Therefore, both processors will enter their critical sections of code.

To avoid this problem, the processor that starts executing its XCHG instruction first (which in this example is processor A) must have exclusive use of the bus until the XCHG instruction is completed. On the 8086/8088 this exclusive use is guaranteed by a LOCK prefix:

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|

which for a maximum mode CPU, activates the LOCK output pin during the execution of the instruction that follows the prefix. The LOCK signal indicates to the bus control logic that no other processors may gain control of the system bus until the locked instruction is completed. To get around the problem encountered in the above example the XCHG instructions could be replaced with:

TRYAGAIN: LOCK XCHG SEMAPHORE,AL

This would ensure that each exchange will be completed in two *consecutive* bus cycles.

Physically, in a loosely coupled system each processing module includes a bus arbiter and the bus arbiters are connected together by special control lines in the system bus. One of these lines is a busy line which is active whenever the bus is in use. When a processing module's arbiter is given control of the bus it will activate the busy line, which will prevent other arbiters from seizing the bus until after the next bus cycle. If a LOCK signal is sent to the arbiter controlling the bus, then that arbiter will retain control of the system by holding the busy line active until the LOCK signal is dropped. Thus, if a processor applies a LOCK signal throughout the execution of an entire instruction, its arbiter will not relinquish the system bus until the instruction is complete. Bus arbiters are discussed further in Sec. 11-2-3, where loosely coupled configurations are considered in detail.

In a module including a coprocessor or closely coupled configuration, if a bus request is made through one of the 8086/8088's RQ/GT pins while the LOCK pin is active, then the request will be held until the LOCK signal is dropped. Then the 8086/8088 will respond to the request by returning a grant. In addition to being activated by a LOCK prefix, an interrupt request on a CPU's INTR pin will cause the LOCK pin to be held low from the beginning of the first INTA pulse until after the second INTA pulse. This guarantees availability of the bus until the completion of an interrupt cycle.

Another possible application of the bus lock capability is to allow fast execution of an instruction which requires several bus cycles. For example, in a multiprocessor system a block of data can be transferred at a higher speed by using the LOCK prefix as follows:

LOCK REP MOVS DEST,SRC

During the execution of this instruction the system bus will be reserved for the sole use of the processor executing the instruction.

A prefix is considered an extension of the instruction following the prefix; therefore, interrupts that occur during the execution of an instruction having a LOCK prefix are not honored until the entire instruction is completed. However, if LOCK is combined with a REP prefix as in the above example, interrupts may be acknowledged at the end of each move operation. Furthermore, upon return from the interrupt, only the prefix immediately preceding the MOVS instruction is restored. Therefore, it is usually best to disable interrupts before executing a LOCK REP combination such as the one indicated above.

In addition to the multiprocessing capabilities of the 8086/8088, the 8288 bus controller also provides control functions for supporting loosely coupled systems in which an 8288 is used in conjunction with a bus arbiter. These control functions are made possible by the presence of the address enable (AEN), I/O bus mode (IOB), command enable (CEN), and master cascade enable/peripheral data enable (MCE/PDEN) pins. There applications are examined in the loosely coupled configurations discussion in Sec. 11-2-3. In the previous discussions the AEN, IOB, CEN, and MCE/PDEN pins were connected for using the 8288 in a single CPU design.

11-2 8086/8088-BASED MULTIPROCESSING SYSTEMS

Let us now consider the three fundamental multiprocessor configurations that the 8086 and 8088 are designed to support. This section gives a general description of how the 8086 and 8088 general purpose microprocessors are used as the dominant processors in these configurations, and the succeeding sections provide specific examples based on other Intel devices with processing capability.

11-2-1 Coprocessor Configurations

Although the 8086 and 8088 are powerful single-chip microprocessors, their instruction set is not sufficient to effectively satisfy some complex applications. For such applications, the 8086/8088 must be supplemented with coprocessors that extend the instruction set in directions that will allow the necessary special computations to be accomplished more efficiently. For example, the 8086/8088 has no instructions for performing floating point arithmetic, but by using an Intel 8087 numeric data processor as a coprocessor, an application that heavily involves floating point calculations can be readily satisfied.

It will be seen that, except for the coprocessor itself, a coprocessor design does not require any extra logic other than that normally needed for a maximum mode system. Both the CPU and coprocessor execute their instructions from the same program, which is written in a superset of the 8086/8088 instruction set. Other than possibly fetching an operand for the coprocessor, the CPU does not need to take any further action if the instruction is executed by the coprocessor.

The interaction between the CPU and the coprocessor when an instruction is executed by the coprocessor is depicted in Fig. 11-3. An instruction to be executed

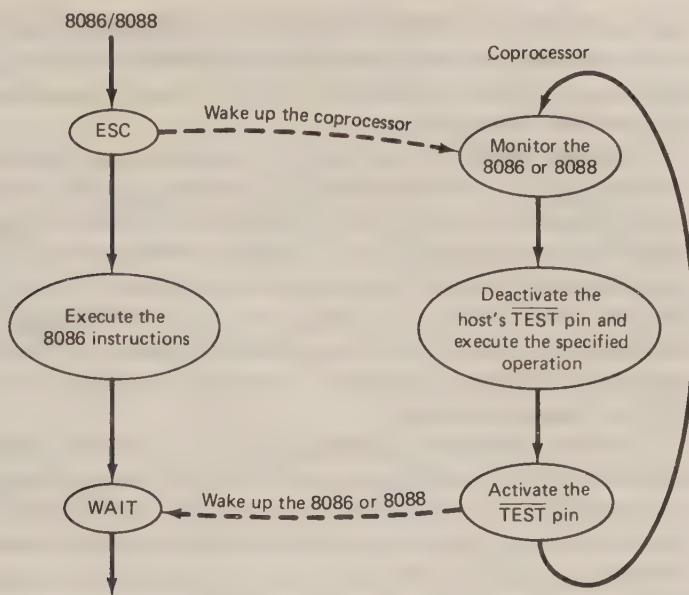


Figure 11-3 Synchronization between the 8086 and its coprocessor.

by the coprocessor is indicated when an escape (ESC) instruction appears in the program sequence. Only the host CPU can fetch instructions, but the coprocessor also receives all instructions and monitors the instruction sequencing of the host. An ESC instruction contains an external operation code, indicating what the coprocessor is to do and is simultaneously decoded by both the coprocessor and the host. At this point the host may simply go on to the next instruction or it may fetch the first word of a memory operand for the coprocessor and then go on to the next instruction. If the CPU fetches the first word of an operand, the coprocessor will capture the data word and its 20-bit physical address. For a source operand that is longer than one word, the coprocessor can obtain the remaining words by stealing bus cycles. If the memory operand specified in the ESC instruction is a destination, the coprocessor ignores the data word fetched by the host and later the coprocessor will store the result into the captured address. In either case, the coprocessor will send a busy (high) signal to the host's TEST pin and, as the host continues processing the instruction stream, the coprocessor will perform the operation indicated by the code in the ESC instruction. This parallel operation continues until the host needs the coprocessor to perform another operation or must have the results from the current operation. At that time the host should execute a WAIT instruction and wait until its TEST pin is activated by the coprocessor. The WAIT instruction repeatedly checks the TEST pin until it becomes activated and then executes the next instruction in sequence.

An ESC assembler language instruction has two operands. The first of two operands indicates the external op code, which determines the action to be taken by the coprocessor. If the second operand specifies a memory location, then as

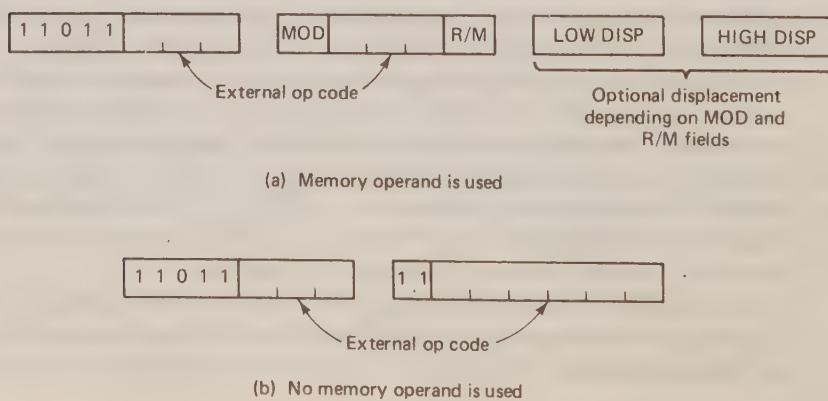
explained above, the 8086/8088 will fetch a word from this location for the coprocessor and may pass the coprocessor an address for storing a result. If the second operand is a register, the register address is treated as part of the external op code and the CPU does nothing.

The machine code for the ESC instruction may have either of the two forms given in Fig. 11-4. In both cases the first byte consists of 11011 followed by 3 bits of the external op code. For the form in Fig. 11-4(a) the second byte specifies a memory addressing mode and 3 more bits of the external op code, thus permitting up to 64 distinct external op codes. If the addressing mode calls for a displacement, this form is extended to include 1 or 2 bytes for holding the displacement. The second byte of the form in Fig. 11-4(b) consists of 11 followed by 6 additional external op code bits. This gives a total of 9 external op code bits and permits up to 512 of these op codes.

The interfacing of a coprocessor to a host CPU is shown in Fig. 11-5. Both the host and the coprocessor share the same clock generator and bus control logic. It is possible to have two coprocessors connected to the same host CPU. When this is done, the coprocessors must be assigned distinct subsets of the set of external op codes and each coprocessor must be able to recognize and execute the members of its subset. For the most part parallel lines can be used to connect the host to its coprocessors. For two coprocessors, one could use the $\overline{RQ/GT0}$ pin on the 8086/8088 and the other could use the $\overline{RQ/GT1}$ pin. The two coprocessors would be connected to separate 8259A interrupt request pins.

In order for a coprocessor to determine when the host is executing an ESC instruction, it must monitor the host CPU's status on the $\overline{S2-S0}$ lines and the AD15–AD0 lines for fetched instructions. Because instructions are prefetched by the CPU, an ESC instruction might not be executed immediately or, if it is preceded by a branch instruction, it might not be executed at all. The coprocessor must track the instruction stream by monitoring the queue status bits QS1 and QS0 and maintaining an instruction queue identical to that of the host. If the queue's status is 00, the coprocessor does nothing, but if it is 01, it will compare the five MSBs of the first

Figure 11-4 Machine code formats for the ESC instruction.



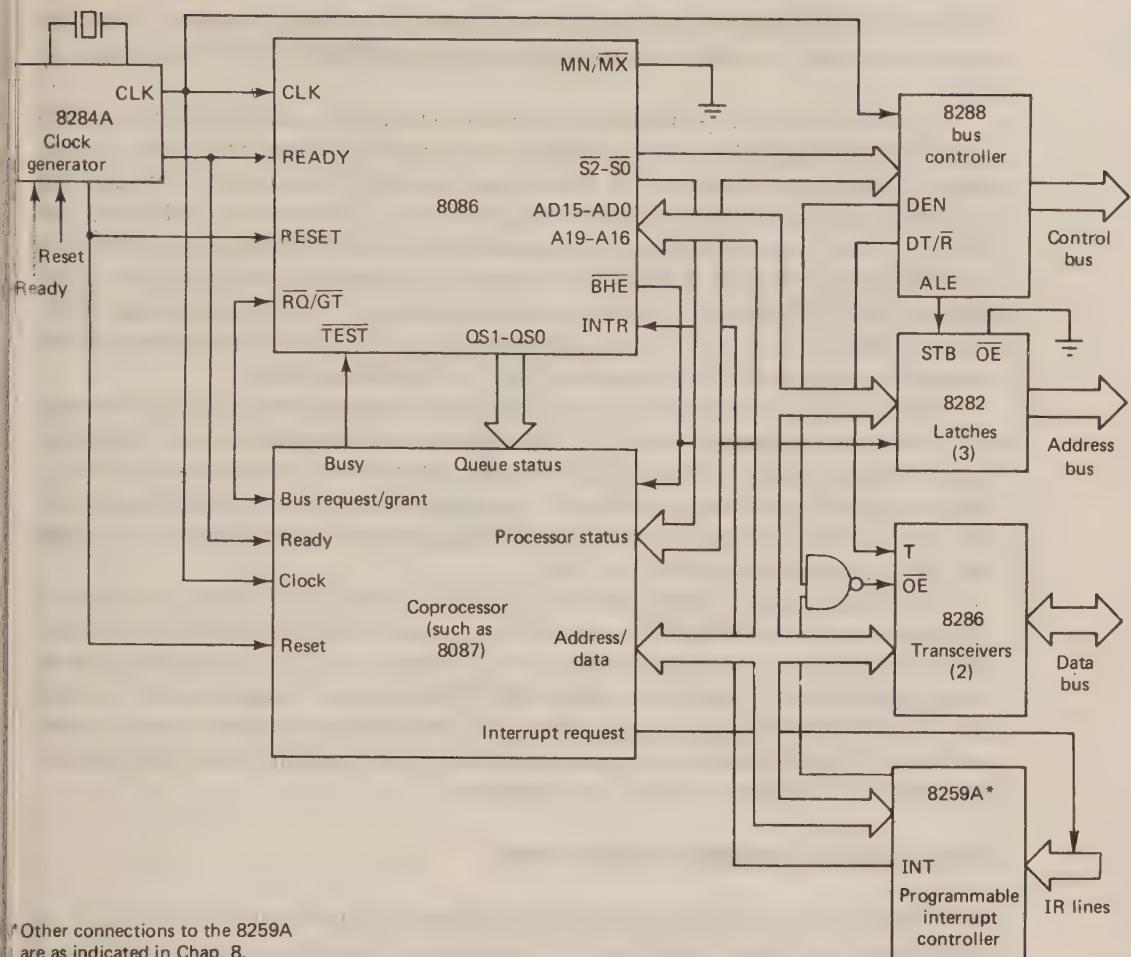


Figure 11-5 Coprocessor configuration.

byte in the queue to 11011. If there is a match, then an ESC instruction is ready to be executed and, assuming that the coprocessor recognizes the external op code, it will perform the indicated operation; otherwise, this byte is ignored and is deleted from the queue. The queue status 10 indicates that the queue in the host is being flushed and, therefore, causes the queue in the coprocessor to also be emptied. The 11 status combination indicates that the first byte in the queue is not the first byte of an instruction and this byte is looked at by the coprocessor only if it is known to be part of an ESC instruction. If not part of an ESC instruction, this byte is ignored.

The coprocessor should be designed so that when an error occurs during the decoding or execution of an ESC instruction, it will send out an interrupt request (which is normally sent to an 8259A). The coprocessor should also be designed so that it can steal bus cycles by making bus requests through one of the host's **RQ/GT** TEST pins.

* Other connections to the 8259A are as indicated in Chap. 8.

GT pins when additional data must be read from or stored in memory. Last, the coprocessor must be able to apply a high signal to the host's TEST pin while it is busy.

It is necessary for a coprocessor to know whether it is working with an 8086 or an 8088 because they have different data bus widths and instruction queue lengths. The type of host can be determined each time the system is brought up by having the coprocessor examine the host's pin 34 immediately following the RESET. This pin is always high on a maximum mode 8088 and on an 8086 it is the BHE/S7 pin. Recall that the first instruction following a reset is taken from address FFFF0. Therefore, the first address on the bus is always even and BHE is always initially low. In order for the coprocessor to know when to check pin 34 it must be connected to the same reset line as the 8086 or 8088.

When both a coprocessor and an independent processor which fetches its own instructions are connected to a CPU, the coprocessor must be able to determine whether an instruction is being fetched by the independent processor or the CPU; otherwise, the instruction queue in the coprocessor may be erroneously updated. This can be done by monitoring the S6 status bit, which is always low for the 8086 and 8088 and always high for the 8089.

Although at the present time the only Intel device which can be used as a coprocessor with the 8086/8088 is the 8087, one could design a customized coprocessor to facilitate any particular application. However, any coprocessor design must be compatible with a maximum mode 8086/8088 system. It must be able to read the CPU status and queue status, make bus and interrupt requests, receive reset and ready signals, receive bus grants, maintain an instruction queue, and decode the external op codes in the ESC instructions.

11-2-2 Closely Coupled Configurations

The 8086/8088 also supports another type of external processor, referred to as an independent processor, which, unlike a coprocessor, executes its own instruction stream. To minimize costs, the independent processor can be tied to the CPU to form a closely coupled multiprocessor system in which both share the same clock and bus control logic. For instruction fetches and data references, the independent processor accesses the bus through the CPU's RQ/GT lines.

Instead of using special instructions such as ESC and WAIT, communication between the host and the independent processor is accomplished through shared memory space. As illustrated in Fig. 11-6, the host sets up a message in memory and then wakes up the independent processor by sending a command to one of the independent processor's ports. The independent processor then accesses the shared memory to get the assigned task and executes the task in parallel with the host. After the task is completed, the external processor notifies its host of the completion by using either a status bit or an interrupt request. The message format varies depending on the design of the independent processor and the application. Typically, a message should specify which operation is to be performed, the input parameters, and the addresses of the locations in which to store the results. An

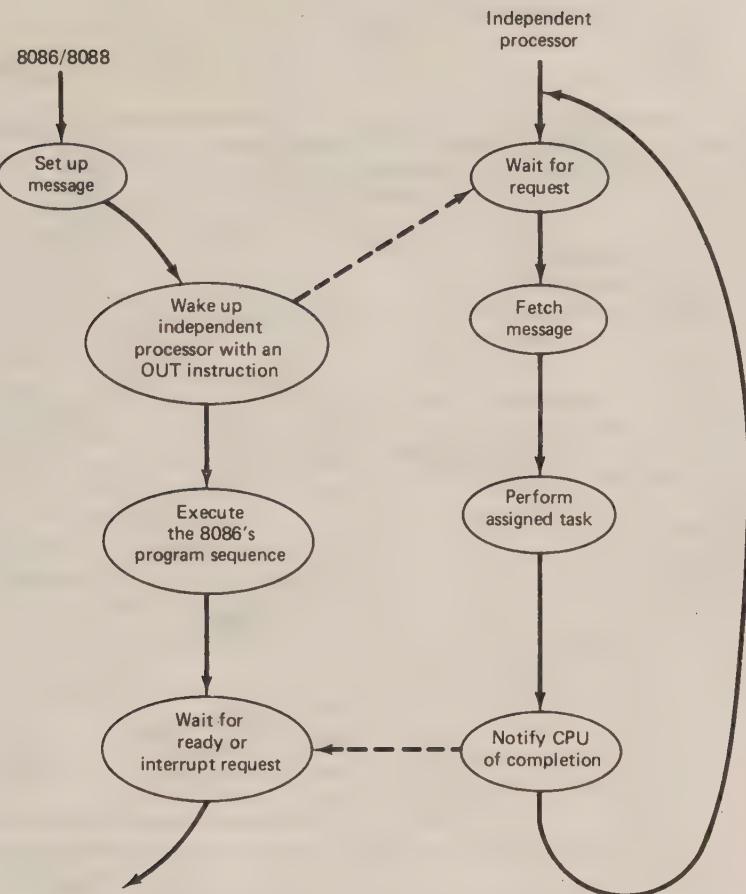


Figure 11-6 Interprocessor communication through shared memory.

example of an independent processor is the Intel 8089 I/O processor and the message layout for the 8089 is described in Sec. 11-4.

A typical connection between the 8086 and an independent processor in a closely coupled configuration is given in Fig. 11-7. Since an independent processor executes its own program, the host's queue status bits are not monitored. The independent processor requests bus accesses via a request/grant line. When one processor is using the bus, the other processor forces its status and address/data outputs to their high-impedance state so that they are logically disconnected from the bus. To wake up the independent processor, the host executes an OUT instruction to output to a port that is assigned to the independent processor. Upon completion of a given task, the independent processor sets a status bit in the shared memory space and may also generate an interrupt. Because the 8086 and 8088 have two request/grant pins, more than one independent processor and/or coprocessor can be connected to the host, thus providing multiprocessing capabilities at a

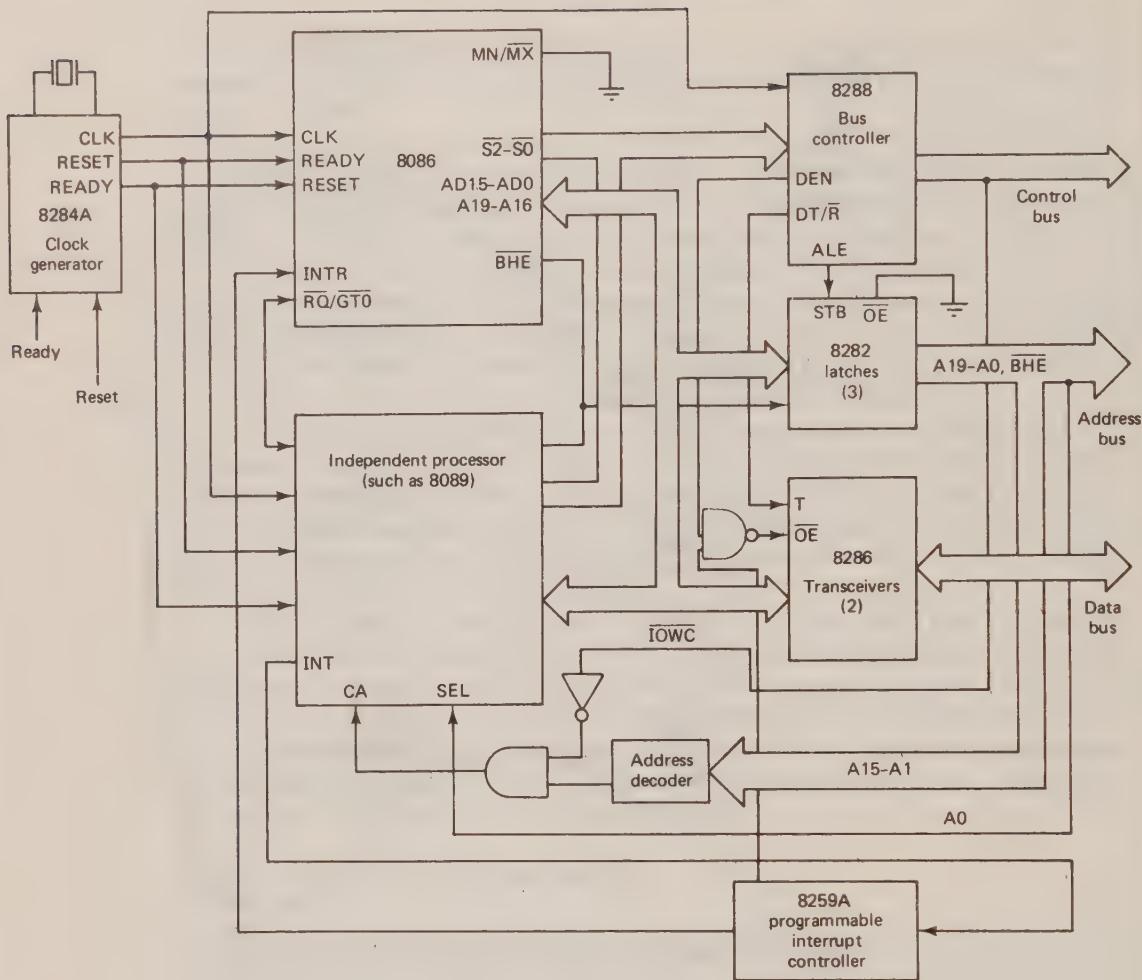
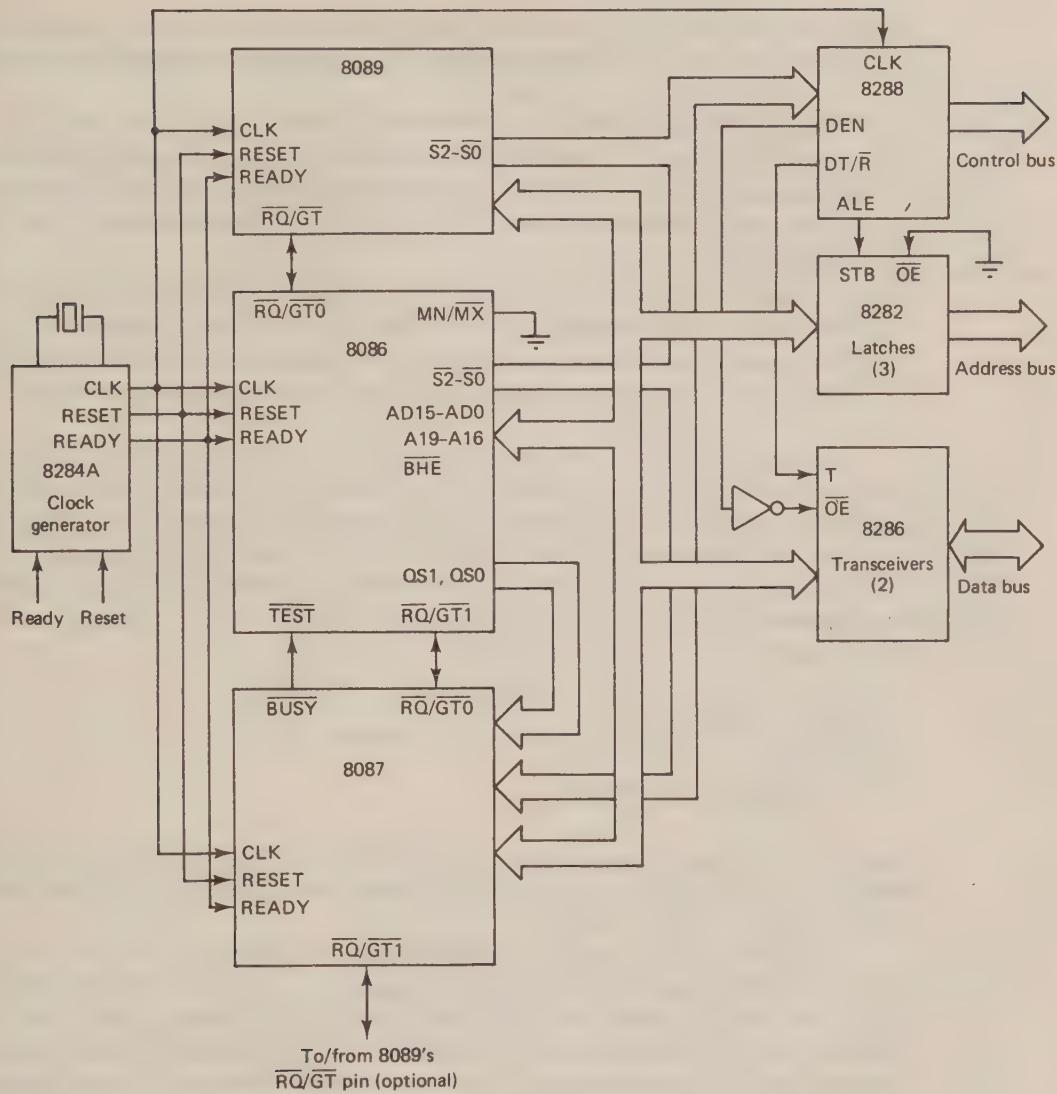


Figure 11-7 8086 connections in a closely coupled configuration.

minimal cost. Closely coupled configurations typically comprise small to medium-size multiprocessor systems. Figure 11-8 illustrates how the coprocessor and closely coupled configurations may be combined. In the example, the 8087 and 8089 share the bus control logic with the host 8086, which provides the bus arbitration function. Because $\overline{RQ}/GT0$ is connected to the 8089, for simultaneous requests the 8086 always gives higher priority to the 8089. However, when the 8087 is using the bus, an 8089's request will not be honored until the 8087 releases the bus. To reduce the maximum wait time, the 8089's \overline{RQ}/GT line could be connected to the $\overline{RQ}/GT1$ pin of the 8087. When this is done, a bus request from the 8089 will force the 8087 to release the bus after the current bus cycle even though it may need several more bus cycles to complete its current instruction.



NOTE: Interrupt system and other details are not shown.

Figure 11-8 Configuration involving both a coprocessor and an independent processor.

11-2-3 Loosely Coupled Configurations

In a multiprocessor system, two 8086s or 8088s cannot be tied directly together. In a loosely coupled multiprocessor system, each CPU has its own bus control logic and bus arbitration is resolved by extending this logic and adding external logic that is common to all master modules. Therefore, several CPUs can form a very

large system and each CPU may have independent processors and/or a coprocessor attached to it. A loosely coupled configuration provides the following advantages:

1. High system throughput can be achieved by having more than one CPU.
2. The system can be expanded in a modular form. Each bus master module is an independent unit and normally resides on a separate PC board. Therefore, a bus master module can be added or removed without affecting the other modules in the system.
3. A failure in one module normally does not cause a breakdown of the entire system and the faulty module can be easily detected and replaced.
4. Each bus master may have a local bus to access dedicated memory or I/O devices so that a greater degree of parallel processing can be achieved.

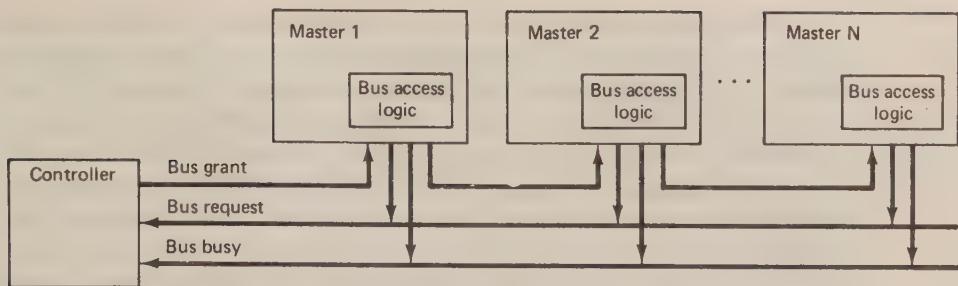
In a loosely coupled multiprocessor system, more than one bus master module may have access to the shared system bus. Since each master is running independently, extra bus control logic must be provided to resolve the bus arbitration problem. This extra logic is called *bus access logic* and it is its responsibility to make sure that only one bus master at a time has control of the bus. Simultaneous bus requests are resolved on a priority basis. There are three schemes for establishing priority:

1. Daisy chaining.
2. Polling.
3. Independent requesting.

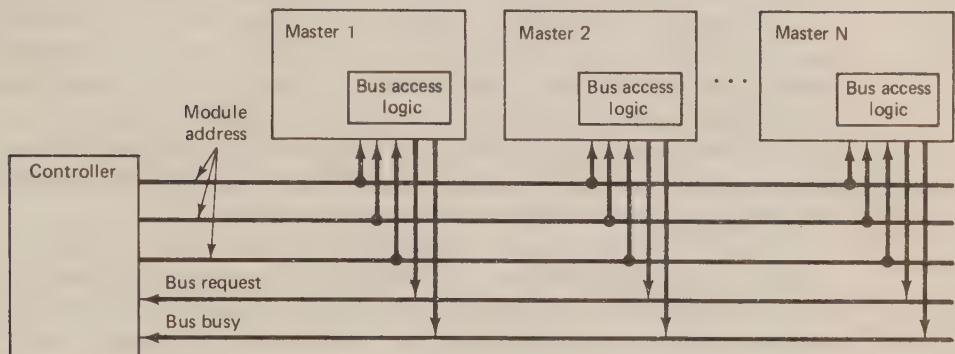
Figure 11-9 illustrates the general concepts underlying these schemes. Implementation of the control logic may vary depending on the complexity of the bus access logic included in each module.

The daisy chain method is characterized by its simplicity and low cost. All masters use the same line for making bus requests. To respond to a bus request signal, the controller sends out a bus grant signal if the bus busy signal is inactive. The grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus. This module blocks the propagation of the bus grant signal, activates the busy line, and gains control of the bus. Therefore, any other requesting module will not receive the grant signal and the priority is determined by the physical locations of the modules. The one located closest to the controller has the highest priority.

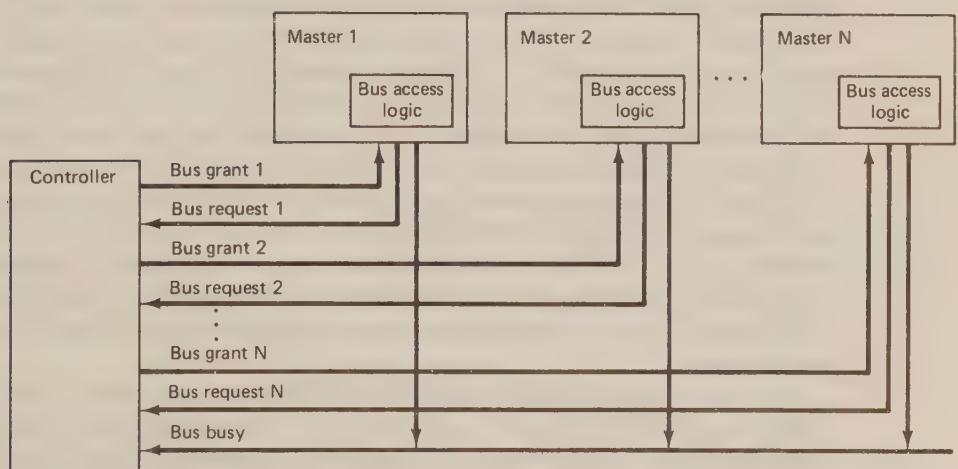
Compared to the other two methods, the daisy chain scheme requires the least number of control lines and this number is independent of the number of modules in the system. However, the arbitration time is slow due to the propagation delay of the bus grant signal. This delay is proportional to the number of modules and, therefore, a daisy chain-based system is limited to only a few modules. Furthermore, the priority of each module is fixed by its physical location and the failure of a module causes the whole system to fail.



(a) Daisy chain method



(b) Polling method



(c) Independent requests method

Figure 11-9 Bus allocation schemes.

The polling scheme uses a set of lines sufficient to address each module. In response to a bus request, the controller generates a sequence of module addresses. When a requesting module recognizes its address, it activates the busy line and begins to use the bus. The major advantage of polling is that the priority can be dynamically changed by altering the polling sequence stored in the controller.

The independent requests scheme resolves priority in a parallel fashion. Each module has a separate pair of bus request and bus grant lines and each pair has a priority assigned to it. The controller includes a priority decoder, which selects the request with the highest priority and returns the corresponding bus grant signal. Arbitration is fast and is independent of the number of modules in the system. Compared to the other two methods, the independent requests design is the fastest; however, it requires more bus request and bus grant lines ($2n$ lines for n modules).

A module's host 8086 or 8088 lacks the capability of requesting bus access and recognizing bus grants. Therefore, as noted above, it is necessary for each module containing a bus master to have extra logic for sending and receiving bus access signals. The Intel 8289 bus arbiter is specifically designed to provide the necessary bus access handshaking. Operating in conjunction with the bus controller, the bus arbiter controls the access of its associated master to the bus by using either the daisy chain or the independent requests scheme.

Figure 11-10 shows the connections between the bus master and the bus arbiter, assuming that all memory and I/O devices are global and are shared by the bus masters through the system bus. Other situations in which a bus master may have access to its private memory or I/O devices through a local bus will be considered shortly.

As shown in the figure, the 8289 inputs and monitors the CPU's status bits S₂-S₀ to determine when to request or release the bus. The 8288 also monitors the status of the CPU to detect the beginning of a bus cycle. After the CPU initiates a bus cycle, a signal on the ALE line causes the address to be latched into the 8283s. If the bus arbiter is currently in control of the bus, it enables the outputs of the 8288 bus controller and the 8283 address latches, and the DEN output of the 8288 enables the data transceivers. Thereafter, the bus cycle proceeds in the normal way. If the bus arbiter does not presently control the bus, then it raises its AEN pin, forcing the command outputs of the 8288 and the address outputs of the 8283s into their high-impedance state. Since the DEN output of the 8288 is also controlled by the AEN line, the data transceivers are disabled. In addition, a high on the AEN line prevents the clock generator from sending a ready signal to the CPU, so that the CPU enters a wait state immediately after the T3 state of the current cycle.

Once the bus arbiter is granted the bus access, it activates the BUSY and AEN pins. The AEN signal causes the address followed by the command signals (after a delay for setup time) to be placed on the system bus, and enables the data transceivers by way of the DEN line. After the data transfer is complete, the addressed location returns an acknowledge signal via the ready line, which causes the 8284A to activate the READY pin. The CPU then exits the wait state and completes its current bus cycle.

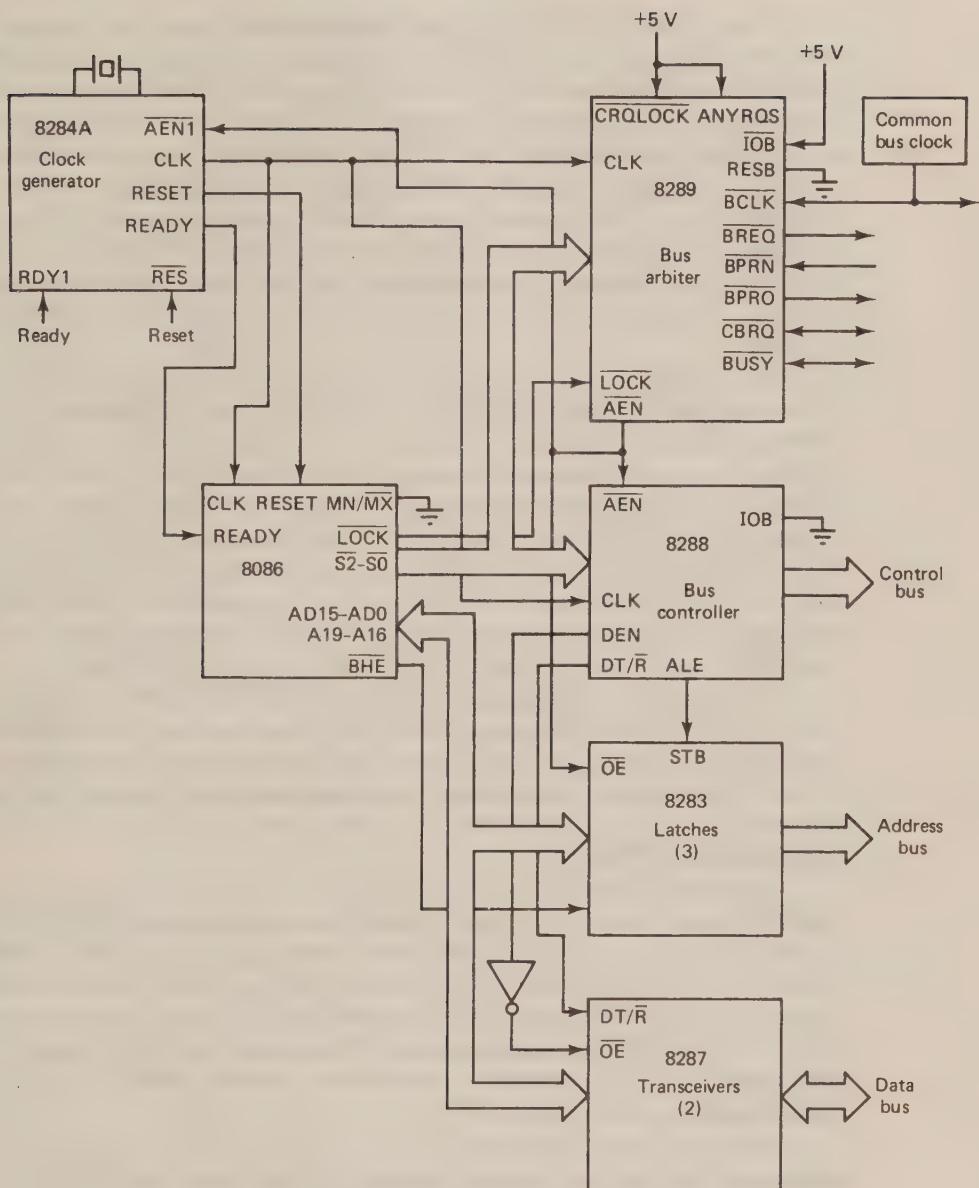


Figure 11-10 Single-processor module.

The LOCK output of the CPU is directly connected to the bus arbiter. A low on the LOCK line prevents the 8289 from relinquishing the bus. The RESB (resident bus mode) and IOB (I/O bus mode) pins are strapping options which permit the processor to access local memory and I/O devices. Because the design in Fig. 11-10 assumes that all resources are shared, both of these modes are disabled.

Four of the other 8289 pins are for bus request, grant, and release lines. The 8289 provides these pins to support the independent requests (parallel) and the daisy chain (serial) accessing schemes. Figure 11-11 shows how bus arbiters are connected to resolve bus priority using the independent requests method. For a system using this design the BREQ (bus request) line from each arbiter is fed into a priority resolving controller. The priority encoder generates a binary number corresponding to the bus request input with the highest priority. This number is decoded to return a bus grant signal to the selected arbiter through its BPRN (bus priority in) pin. Because the timing of each module is controlled by an independent clock, it is necessary to have a separate common clock to synchronize the bus requests.

The CBRQ (common bus request) and BUSY (busy) pins provide a means for the current bus master to release the bus. Both pins are bidirectional with open collector outputs. Their connections are also shown in Fig. 11-11. When a module needs to frequently access the system bus, it is better for its arbiter to retain the bus (except when its processor is inactive) until it is forced off by another module. Otherwise, each bus cycle would incur an overhead due to the request and release steps. A bus master of higher priority can acquire the bus via its BREQ line and the priority selection logic. After the current bus master completes its bus cycle, it releases the bus and deactivates BUSY. The requesting master which accepts the BPRN signal then activates BUSY and begins to use the bus.

A bus arbiter of lower priority may use the CBRQ line to acquire the bus. This, of course, would require that the CBRQ pins be connected together as shown in Fig. 11-11. When the CBRQ is low, the current master will surrender the bus if it is in a TI state. The priority resolving logic then allocates the bus to the requesting arbiter which has the next higher priority by putting a signal on the corresponding BPRN line.

In connection with allowing lower-priority devices to take control of the bus, the 8289 provides two inputs, ANYRQST (any request) and CRQLCK (common bus request lock). The ANYRQST pin is a strapping option. When ANYRQST is tied to +5 V, an assertion of CBRQ forces the bus arbiter to release the bus at the end of the current bus cycle regardless of the priority of the requesting bus master. The CRQLCK input allows an 8289 to ignore a CBRQ signal. Therefore, activating CRQLCK prevents an 8289 from releasing the bus to any other bus master having a lower priority.

The need for having an external priority controller is eliminated by using the daisy chain bus allocation scheme illustrated in Fig. 11-12. The BPRO (bus priority out) of each module is connected to the BPRN (bus priority in) of the next lower priority bus master in the chain. For any processing module in the chain, a low input on BPRN indicates that it has the highest priority and may acquire the bus. If its BPRN pin is high, it will not have received the grant signal and will not be able to pass on a grant signal to its BPRO. The BPRN signal of the highest-priority arbiter is grounded; therefore, the BPRN pin of the requesting arbiter with the highest priority will be low and, for arbiters farther down the chain, it will be high.

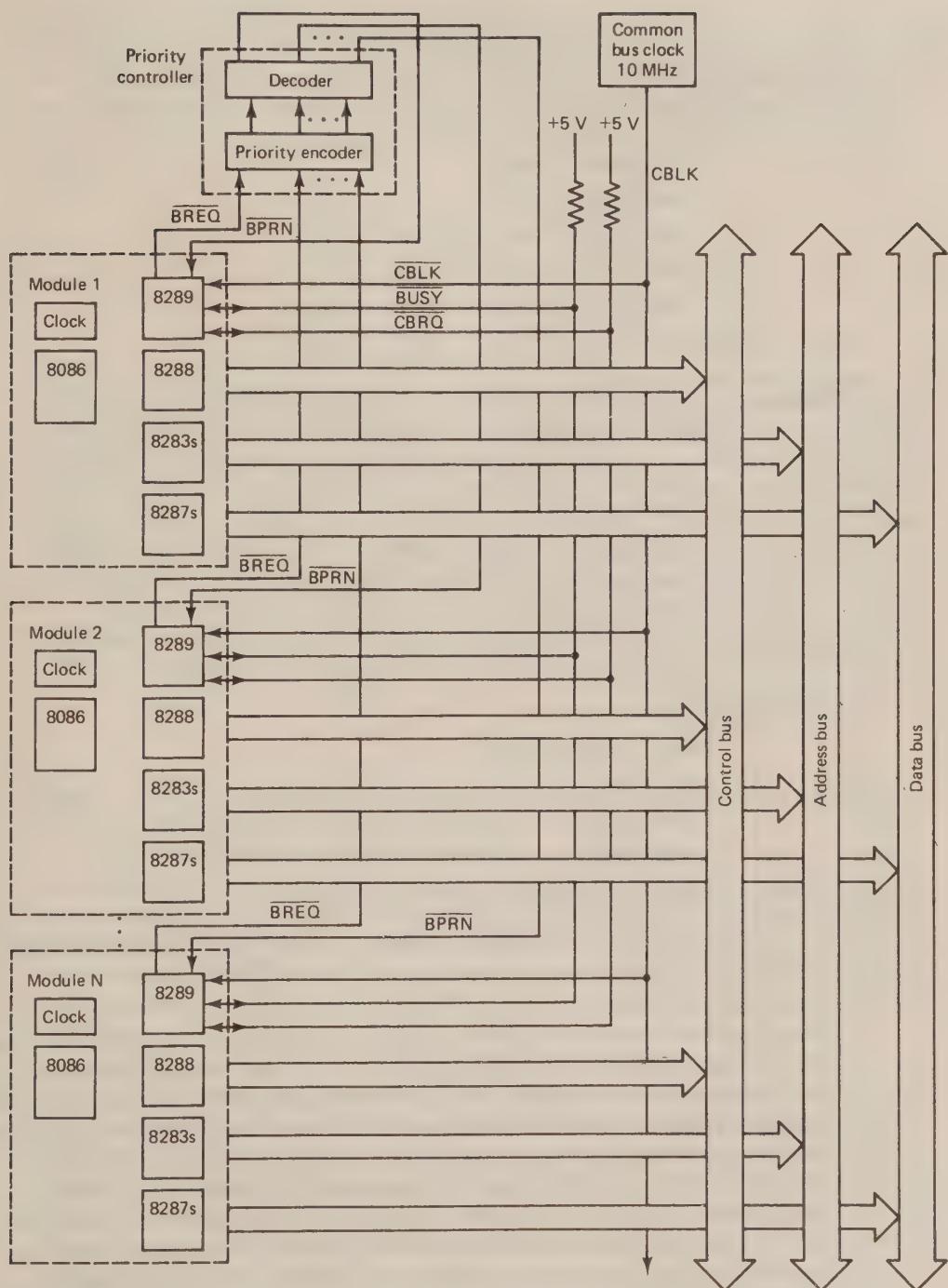


Figure 11-11 8289 connections to a priority controller.

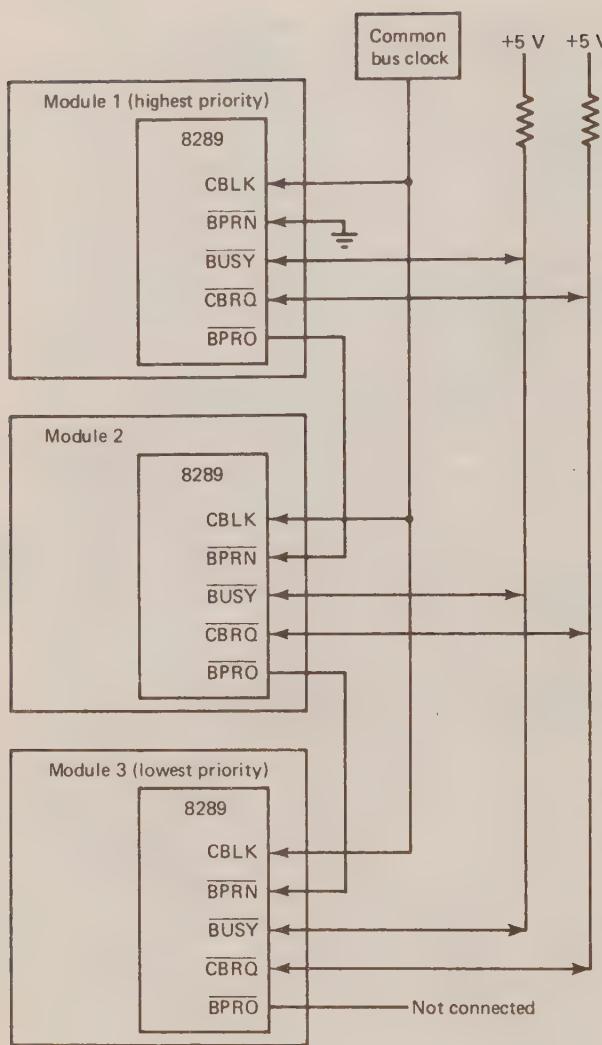


Figure 11-12 Daisy chain bus allocation using 8289s.

Since the bus allocation must be resolved in a BCLK clock period, the timing is normally such that a chain cannot include more than three processing modules.

As more processing modules are tied to the system bus, the bus utilization will soon become saturated. Bus congestion seriously degrades the performance of a system and tends to negate the speed advantage of multiprocessing. A rough analysis illustrating the influence of the number of processors on system performance can be obtained by considering k identical modules. It is reasonable to say that the instruction execution rate of each processor is proportional to the rate at which that processor accesses the bus. Assume that the bus bandwidth is N cycles per second and that without interference each module in the system uses M bus cycles per second. The bus utilization by a module is defined as the average fraction

of bus cycles used by the module assuming no interference and is $B_P = M/N$. This ratio must be less than 1 and will depend on the instructions being executed. For the 8086, it will typically range from 0.5 to 0.8. Let I_S be the system instruction execution rate (i.e., the number of instructions executed by the system per second) and I_P be the instruction execution rate of each module assuming no interference. Then

$$I_S \approx kI_P \quad \text{if } k \leq \frac{1}{B_P}$$

and

$$I_S \approx \frac{I_P}{B_P} \quad \text{if } k > \frac{1}{B_P}$$

A representative graph of I_S versus the number of processing modules for $B_P = 1/2$ is shown in Fig. 11-13(a). Although I_S is approximately equal to kI_P for an underutilized system, I_S is actually less than kI_P because a module will enter a wait state if it needs to access the system bus and the bus is being used by another processor. It is important to note from the figure that once the bus is saturated, adding more modules will no longer improve the system performance. The bus becomes the bottleneck of the system and the system is said to be *bus bound*.

The utilization of each module, which is defined as the ratio of the instruction execution rate (in the presence of other modules) to I_P , is somewhat degraded when bus congestion becomes serious, i.e., kM approaches N . This utilization is

$$U_m \leq \frac{N/k}{M} = \frac{N}{kM} = \frac{1}{kB_P}$$

Figure 11-13(b) illustrates the influence of the number of modules on the performance of each module for the case $B_P = 1/2$.

The most effective solution to the bus congestion problem is for each processing module to include a local memory dedicated to its CPU. Each module would use its local memory to store its program and as a working area, and the shared memory would be primarily used for intermodule communications. This would allow several CPUs to fetch instructions or reference operands simultaneously, thus reducing the traffic on the system bus and increasing the degree of concurrent processing.

As illustrated in Fig. 11-14, if a processing module has a local bus, then it must include two sets of bus control logic, one for accessing the system bus and the other for the local bus. Each set would need to contain a bus controller, address latches, and data transceivers. Since the local bus is dedicated to only one CPU, it does not require a bus arbiter and its address latches are always enabled (assuming there is no local DMA controller). During a bus cycle, the address is decoded to determine if it is in the local space or shared space. If the address is in the local space, the local bus controller is enabled and a bus cycle proceeds immediately. Meanwhile, the output of the decoder disables the system bus controller (and causes the bus arbiter to release the system bus if the arbiter has control of it). If the

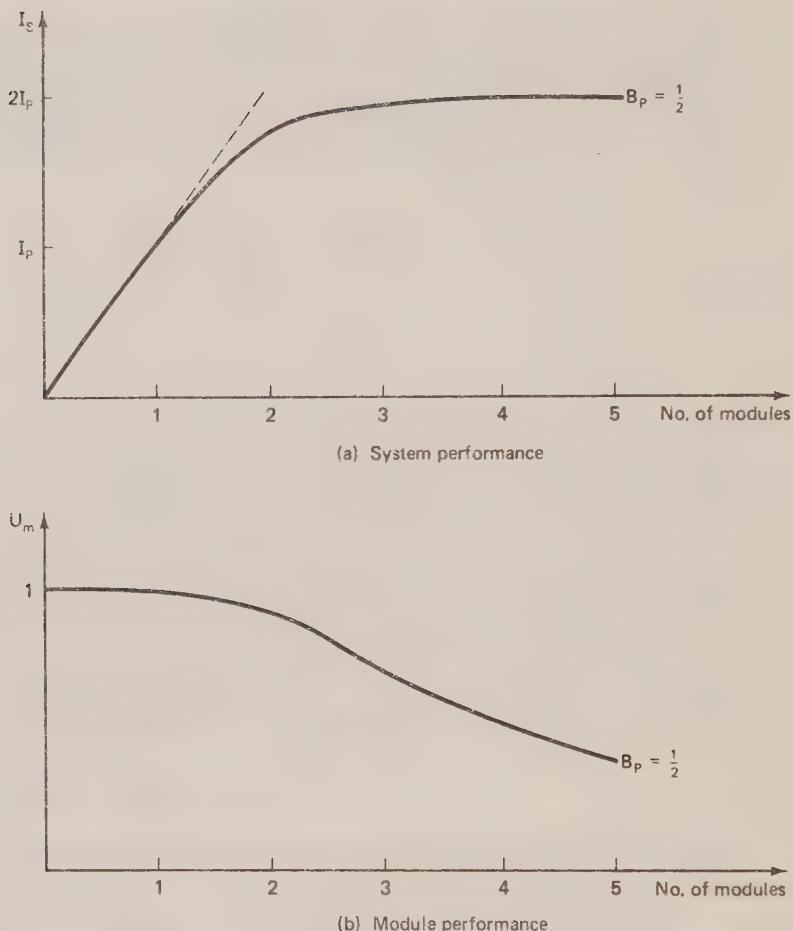


Figure 11-13 Multiprocessing system performance criteria.

address is in the address space of the system bus, the address decoder activates the SYSB/RESB pin on the 8289 and a system bus access is attempted. In this event, the local bus controller is disabled, preventing it from sending out a bus command, and the system bus interface is enabled. The bus arbiter then requests the use of the system bus and initiates a bus cycle when it becomes the arbiter having the highest priority.

As shown in Fig. 11-14, the RESB and \overline{IOB} lines are tied to +5 V. With local resources, each processing module would normally access the system bus only occasionally. Therefore, it is desirable to enable ANYRQST so that the 8289 may release the bus to lower-priority processing modules as soon as the current bus cycle is complete.

Figure 11-15 gives another configuration that reduces the traffic on the system bus, but results in a simpler design than the one shown in Fig. 11-14. This config-

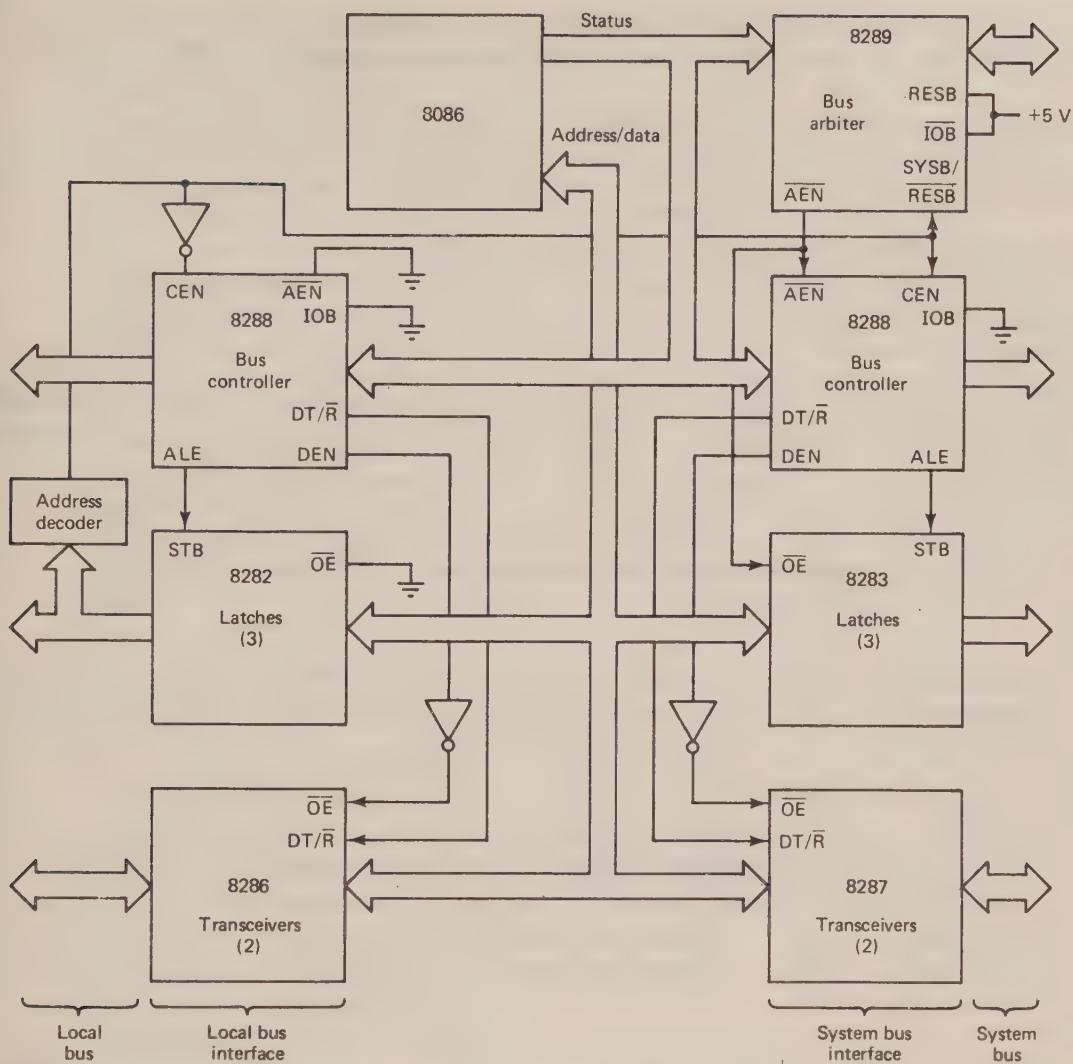


Figure 11-14 Configuration with both a local bus and a system bus.

uration locates all the I/O on a local bus and all of the memory on the system bus. The separation of the I/O and memory address spaces permits special facilities of the 8289 to direct the address and control signals flow and eliminates the need for the address decoder and local 8288 bus controller. By strapping the RESB pin on the 8289 low the SYSB/RESB is ignored, and by strapping the IOB pin low on the 8289 and high on the 8288 these devices are put into their I/O bus mode. In this mode the system bus is requested and surrendered according to the S2 line, which indicates whether an I/O (low) or memory (high) transfer is to be conducted. The AEN output of the 8289 is still used to control the address/data and memory

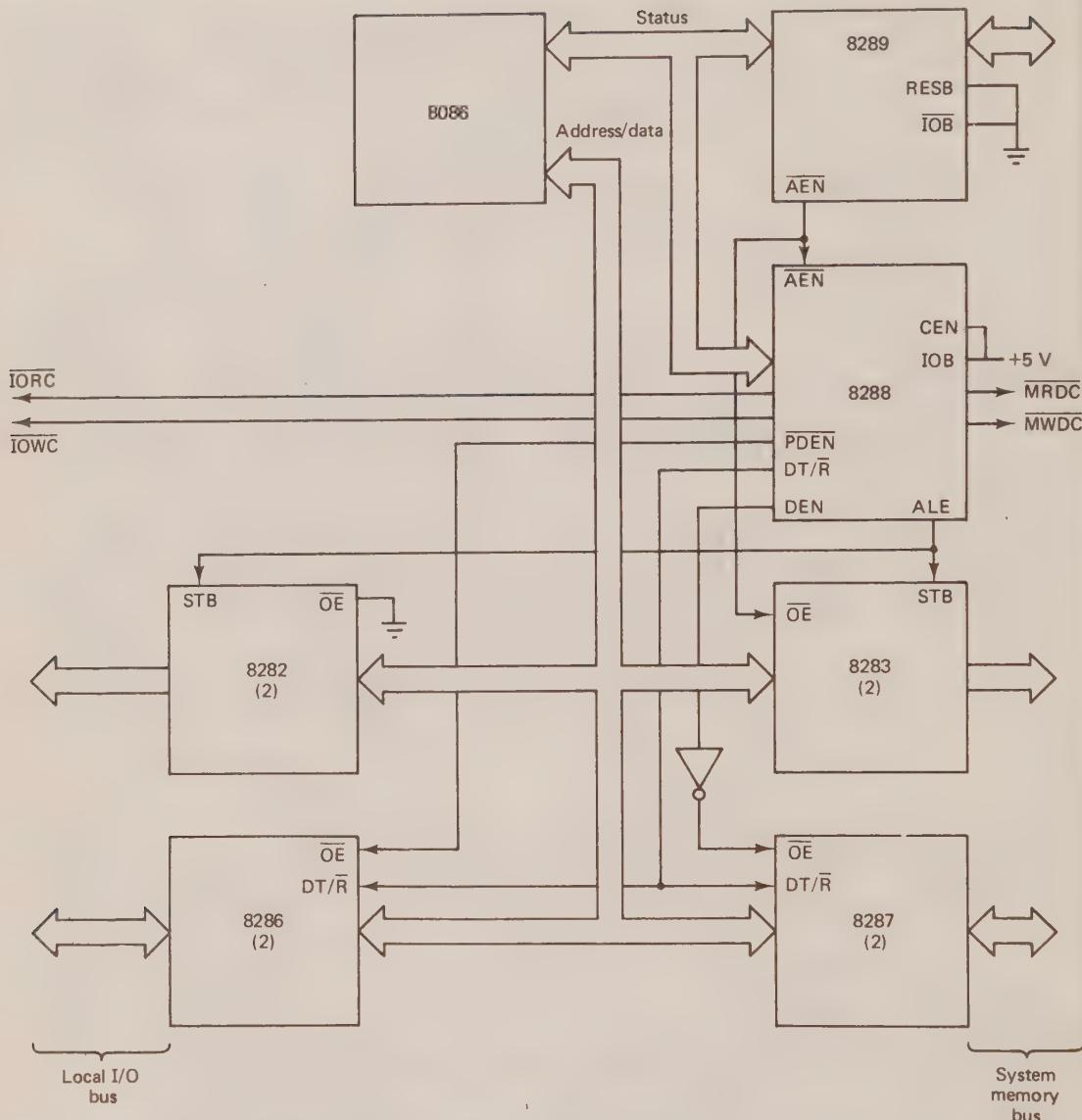


Figure 11-15 Separate local I/O bus configuration.

command lines but has no effect on the I/O control lines \overline{IORC} , \overline{IOWC} , \overline{AIOWC} , and \overline{INTA} (whose outputs are always enabled). With IOB on the 8288 strapped high the MCE/PDEN pin becomes the peripheral data enable pin, which serves to enable the I/O bus data transceivers. When an I/O transfer is being made \overline{PDEN} is active and DEN is inactive. For a memory transfer DEN is active and \overline{PDEN} is inactive. This configuration is particularly useful when implemented in conjunction

with an 8089 I/O processor. It is also possible to connect I/O interfaces to the system bus, but their port addresses must be addressed as memory (i.e., memory-mapped I/O devices).

The local bus mode and the I/O bus mode may be combined by strapping the RESB and IOB pins of the 8289 to high and low, respectively. The resulting configuration allows part of the memory to be placed on the local bus along with the I/O devices.

It has been pointed out that local memory reduces the use of the system bus in a multiprocessor system. However, a processing module may still need to initially load the control program from the shared system memory to its local memory, and move results from the local memory to the system memory if the results are needed by other modules. A dual-port memory can be used to further reduce the traffic on the system bus by handling the interprocessor communications. This is a hardware concept that is similar to the software concept of using a common area to provide the communication between a calling program and a subprogram. Such a memory has two ports, to allow references from both the local bus and the system bus. As shown in Fig. 11-16, the processor of the master module in which the dual-port memory resides accesses the memory through the local bus. But other proc-

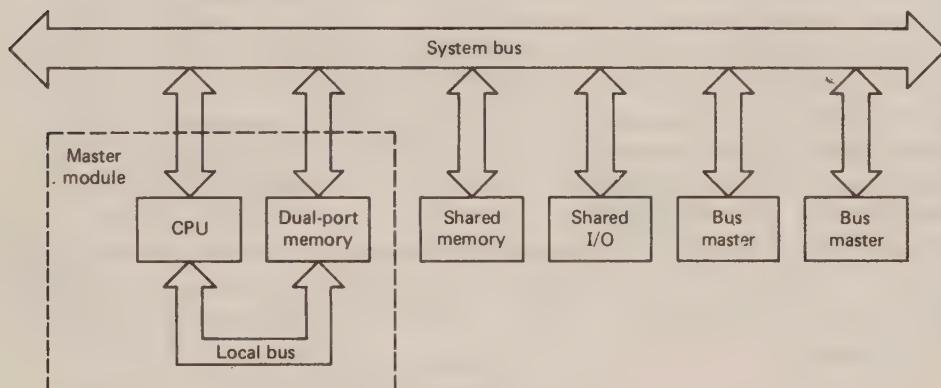


Figure 11-16 Dual-port memory.

essing modules may also access the same memory, although they can communicate with it only through the system bus. This configuration is particularly useful in inputting or outputting large blocks of data through a processing module which is dedicated to I/O processing. In this design, locking the system bus does not prevent the master module from accessing the dual-port memory, and vice versa. In order to use the scheme for avoiding the critical section problems described in Sec. 11-1, the semaphore should be placed in the shared system memory; otherwise, the implementation becomes more complicated.

In summary, processing modules of different configurations may be combined

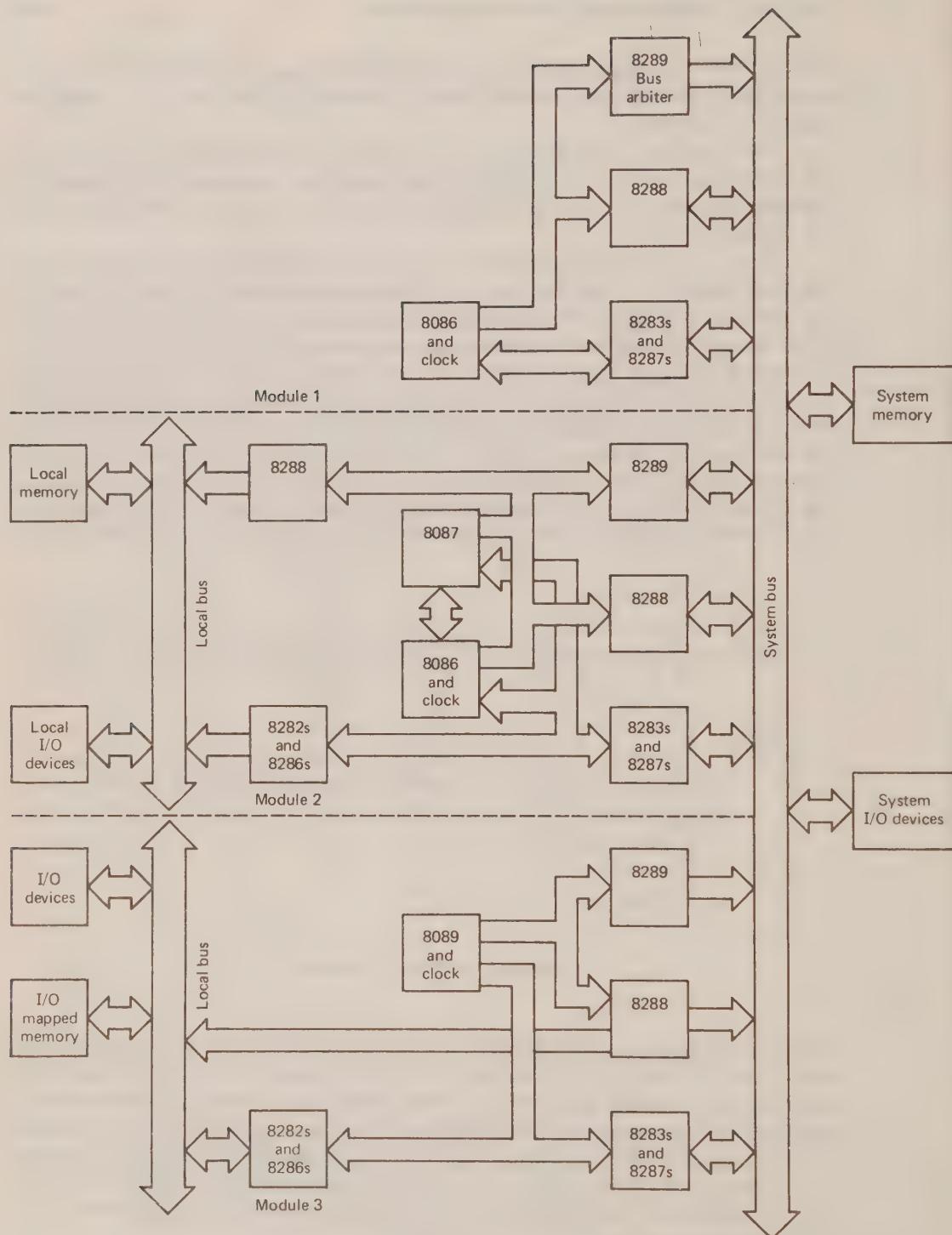


Figure 11-17 Complex multiprocessing system.

to form a complex, loosely coupled multiprocessor system. Each module in such a system may be:

1. A single 8086 or 8088 or an independent processor such as an 8089.
2. A cluster of processors consisting of an 8086 or 8088 and a coprocessor (such as an 8087) and/or independent processors.
3. A cluster of independent processors (such as two 8089s).

In addition, each module may include a local bus or a dedicated I/O bus. Figure 11-17 illustrates one of the many possible multiprocessing designs—designs that may include several complex processing modules

11-2-4 Microcomputer Networks

The previous multiprocessor configurations have a common characteristic and that is that all processors share the same system bus. Thus, the interprocessor communications are through shared memory and processors must be physically located close to each other.

By using serial links, many microcomputer systems can communicate with each other and share some of the same hardware and software resources. Large systems of this type are called *computer networks*. Normally, synchronous serial transfer is employed to send messages between the systems. Commands and data are transmitted as message packets. Each packet normally includes several fields containing such things as synchronization characters, the sender's address, the receiver's address, the text, error detection characters, and termination characters. The controlling information and its arrangement is referred to as the system's *communication protocol*. Each microcomputer (or multiprocessor element) in the network is relatively independent of the others, and the microcomputers could be miles, or even thousands of miles, apart. A failure in one microcomputer element of the network could be easily isolated without affecting the remainder of the system.

11-3 THE 8087 NUMERIC DATA PROCESSOR

The 8087 numeric data processor (NDP) is specially designed to perform arithmetic operations efficiently. It can operate on data of the integer, decimal, and real types, with lengths ranging from 2 to 10 bytes. The instruction set not only includes various forms of addition, subtraction, multiplication, and division, but also provides many useful functions, such as taking the square root, exponentiation, taking the tangent, and so on. As an example of its computing power, the 8087 can multiply two 64-bit real numbers in about 27 μ s and calculate a square root in about 36 μ s. If performed by the 8086 through emulation, the same operations would require approximately 2 ms and 20 ms, respectively. The 8087 provides a simple and

effective way to enhance the performance of an 8086/8088 based system, particularly when an application is primarily computational in nature.

The NDP cannot fetch its own instructions and, therefore, must operate with either an 8086 or 8088, which acts as the host in a coprocessor configuration. The interfacing between the host and the NDP simply requires connections between the corresponding pins of the two processors and, as indicated in Fig. 11-5, no additional components are required.

A pin diagram of the 8087 is shown in Fig. 11-18. The address/data, status, ready, reset, clock, ground, and power pins of the NDP have the same pin positions as those assigned to the 8086/8088. Among the remaining eight pins, four of them are not used. The other pins are connected as follows: the BUSY pin to the host's TEST pin; the RQ/GT0 pin to the host's RQ/GT0 or RQ/GT1 pin; the INT pin to the interrupt management logic (assuming the 8087 is enabled for interrupts); and the RQ/GT1 could be connected to the bus request/grant pin of an independent processor such as an 8089. This simple interface allows an existing maximum mode 8086/8088-based system to be easily upgraded by replacing the original CPU with a piggyback module, which consists of an 8086/8088 and an 8087.

11-3-1 NDP's Data Types

The 8087 can operate on memory operands of seven different data types: word integer, short integer, long integer, packed BCD, short real, long real, and temporary real. The number of bytes, format, and approximate range for each of these data types are shown in Fig. 11-19. In memory, the least significant byte of a number is always stored in the lowest address.

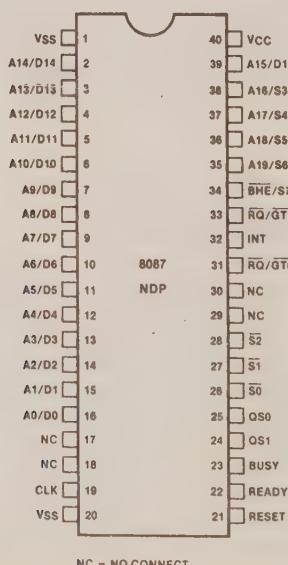


Figure 11-18 Pin diagram for the 8087.
(Reprinted by permission of Intel Corporation. Copyright 1982.)

Figure 11-19 Data formats for the NDP.

The 8086/8088 can operate on integers of only 16 bits, with a range from -2^{15} to $2^{15} - 1$. In addition to the word integer format, the 8087 allows integer operands to be represented using 32 bits (short integer) and 64 bits (long integer). Negative integers are coded in 2's complement form. The greater number of bits significantly extends the ranges of integers to -2^{31} through $2^{31} - 1$ for the short integer format and -2^{63} through $2^{63} - 1$ for the long integer format. This is roughly $\pm 2 \times 10^9$ and $\pm 9 \times 10^{18}$, respectively.

In the packed BCD format, a decimal number is stored in 10 bytes. The entire most significant byte is dedicated to the sign. The most significant bit of this byte indicates whether a decimal number is positive (0) or negative (1). The remaining 9 bytes represent the magnitude with two BCD digits packed into each byte. Therefore, the valid range of values in the packed BCD format is $-10^{18} + 1$ to $10^{18} - 1$.

The real data types, also called the floating point types, can represent operands which may vary from extremely small to extremely large values and retain a constant number of significant digits during calculations. A real data format is divided into three fields: sign, exponent, and mantissa, i.e.,

$$X = \pm 2^{\text{exp}} \times \text{mantissa}$$

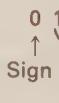
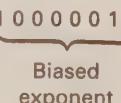
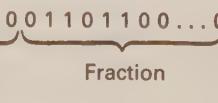
The exponent adjusts the position of the binary point in the mantissa. Decreasing the exponent by 1 moves the binary point to the right by one position. Therefore, a very small value can be represented by using a negative exponent without losing any precision. However, except for numbers whose mantissa parts fall within the range of the format, a number may not be exactly representable, thus causing a roundoff error. If leading 0s are allowed, a given number may have more than one representation in a real number format. But since there are a fixed number of bits in the mantissa, leading 0s increase the roundoff error. Therefore, in order to minimize the roundoff errors, after each calculation the 8087 deletes the leading 0s by properly adjusting the exponent. A nonzero real number is said to be *normalized* when its mantissa is in the form of 1.F, where F represents a fraction.

Normally, a bias value is added to the true exponent so that the true exponent is actually the number in the exponent field minus the bias value. Using biased exponents allows two normalized real numbers of the same sign to be compared by simply comparing the bytes from left to right as if they were integers.

The 8087 recognizes three real data types: short real, long real, and temporary real. In the short real data format, the biased exponent E and the fraction F have 8 and 23 bits, respectively, and the number is represented by the form:

$$(-1)^S \times 2^{E-127} \times 1.F$$

where S is the sign bit. Because the 1 appearing before the fraction is always present, it is implied and is not physically stored. For example, suppose that a short real number is stored as follows:

| | | | | |
|---|---|---|---|-------------------|
| 0 |  |  |  | $= 41360000_{16}$ |
|---|---|---|---|-------------------|

Then the true exponent is $130 - 127 = 3$ and the floating point number being represented is

$$\begin{aligned} +1.011011 \times 2^3 &= +1011.011 = 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 \\ &\quad + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 11.375_{10} \end{aligned}$$

To illustrate the conversion of a real number to its short real form, consider the number 20.59375_{10} . First, one should convert the integral and fractional parts to binary as follows:

$$10100 + .10011 = 10100.10011$$

After normalizing this number by moving the binary point until it is between the first and second bits, it is written:

$$1.010010011 \times 2^4$$

From this form it is seen that

$$S = 0, \quad E = 127 + 4 = 131 = 10000011, \quad \text{and} \quad F = 010010011,$$

and the short real format of the number is:

$$\underbrace{0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1}_{\text{Biased exponent}} \underbrace{0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \dots 0}_{\text{Fraction}} = 41A4C000_{16}$$

For the short real data type, the valid range of the biased exponent is $0 < E < 255$. Consequently, the numbers that can be represented are from $\pm 2^{-126}$ to $\pm 2^{128}$, approximately $\pm 1 \times 10^{-38}$ to $\pm 3 \times 10^{38}$.

A biased exponent of all 1s is reserved to represent infinity or “not-a-number” (NAN). At the other extreme, a biased exponent of all 0s is used to represent +0 (all 0s with a + sign), -0 (all 0s with a - sign), or a denormal. A denormal is a result that causes an underflow and has leading 0s in the mantissa even after the exponent is adjusted to its smallest possible value. NaNs and denormals are normally used to indicate overflows and underflows, respectively, although they may be used for other purposes.

The long real format has 11 exponent bits and 52 fraction bits. As with the short real format, the first nonzero bit in the mantissa is implied and not stored. The range of representable nonzero quantities is extended to approximately $\pm 10^{-308}$ to $\pm 10^{308}$. As a comparison the range of nonzero real numbers for the IBM 370 real format is $\pm 10^{-78}$ to $\pm 10^{76}$.

The 8087 internally stores all numbers in the temporary real format which uses 15 bits for the exponent and 64 bits for the mantissa. Unlike the short and long real formats, the most significant bit in the mantissa is actually stored. Because of the extended precision (19 to 20 decimal digits), integer and packed BCD operands can be operated on internally using floating point arithmetic and still yield exact results. A primary reason for using the temporary real format for internal

data storage is to reduce the chances for overflows and underflows during a series of calculations which produce a final result that is within the required range. For example, consider the calculation:

$$D \leftarrow (A * B) / C$$

where A, B, C, and D are in the short real format. The multiply operation may yield a result which is too large to be represented in the short real format, but yet the final result, after the division by C, may still be within the valid range. The 15-bit exponent of the temporary real format extends the range of valid numbers to approximately $\pm 10^{\pm 4932}$; therefore, for most applications the user need not worry about overflows and underflows in the intermediate calculations.

In order to support the 8087's data formats, in addition to the DB, DW, and DD directives, the ASM-86 assembler provides the data declaration directives DQ (define quadword) and DT (define tenbyte) to allocate storage or define data. The directive DQ is used to define an 8-byte storage block for long integer and long real values. For a DT directive, 10 bytes are allocated and storage can be initialized with values in the packed decimal or temporary real format. Some examples are given in Fig. 11-20. The PTR operator may be used with the DQ or DT type just as they are with the DB, DW, and DD types.

| | | | |
|------------|----|---------------|--------------------------|
| SHORT_REAL | DD | 1.125 | ;STORE AS SHORT REAL |
| SHORT_INT | DD | 12345678H | ;STORE AS SHORT INTEGER |
| LONG_REAL | DQ | -1.9 | ;STORE AS LONG REAL |
| LONG_INT | DQ | 12345678ABCDH | ;STORE AS LONG INTEGER |
| BCD | DT | -91567 | ;STORE AS PACKED DECIMAL |
| REAL_CONST | DT | -1.3 | ;STORE AS TEMPORARY REAL |

Figure 11-20 Examples of DD, DQ, and DT directives.

11-3-2 Processor Architecture

A general block diagram of the 8087 is given in Fig. 11-21. The monitor and control logic maintains a 6-byte instruction queue (only 4 bytes are used if operated in conjunction with the 8088) and tracks the instruction execution sequence of the host. If the instruction currently being executed by the host is an ESC instruction, the 8087 decodes the external op code to perform the specified operation, and also captures the operand and operand address. Instructions other than ESC instructions are simply ignored by the 8087.

There are eight data registers which can be accessed either as a stack or randomly relative to the top of the stack. An operand may be popped from this stack or pushed onto it. The top stack element is pointed to by the ST bits, which are bits 13, 12, and 11 of the status register. A push operation first decrements ST by 1 and then loads the operand into the new top of the stack element, and a pop operation retrieves the top of the stack and then increments ST by 1.

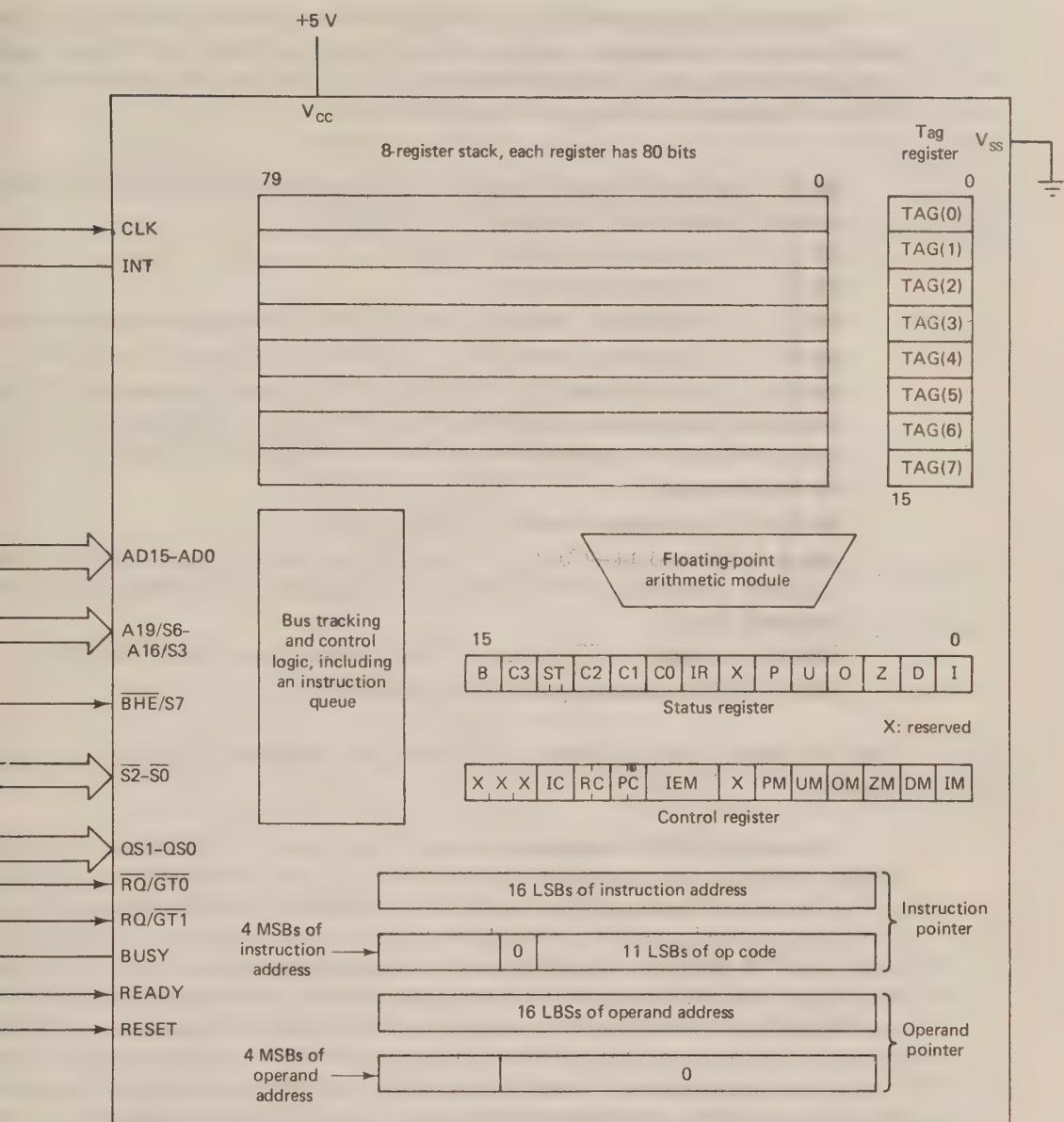


Figure 11-21 Block diagram of the 8087.

As a conventional register file, each register may be referenced by using an index to the stack pointer. This is called *relative stack addressing*. For relative stack addressing the registers are considered to be in a circle with register 7 being next to register 0. For instance, if ST contains a 3, then ST(2) and ST(6) represent register 5 and register 1, respectively. Because all numbers are internally stored in the temporary real format, each register consists of 80 bits.

The status register is 16 bits wide. It reports various errors, stores the condition code for certain instructions, specifies which register is the top of the stack, and indicates the busy status. The bit definitions are given below. A 1 in bit 0 through 5, 7, or 15 indicates that the given condition exists.

Bit 0—I, an invalid operation such as stack overflow, stack underflow, invalid operand, square root of a negative number, etc.

Bit 1—D; the operand is not normalized.

Bit 2—Z, a divide-by-zero error.

Bit 3—O, an exponent overflow error, i.e., the biased exponent is too large.

Bit 4—U, an exponent underflow, i.e., the biased exponent is too small.

Bit 5—P, a precision error, i.e., the result is not exactly representable in the destination format and roundoff has been performed. This indication is normally used only for applications where exact results are required.

Bit 6—Reserved.

Bit 7—IR, an interrupt request is being made.

Bits 8, 9, 10, and 14—C0, C1, C2, and C3 indicate the condition code. The condition code is set by the compare and examine instructions, which are discussed later.

Bits 13, 12, and 11—ST, indicates which element is the top of the stack.

Bit 15—B, the current operation is not complete.

After the 8087 is reset or initialized, all status bits except the condition code are cleared.

Although the 8087 recognizes six error types, each error type may be individually masked from causing an interrupt by setting the corresponding mask bits to 1 in the control register. These mask bits are denoted IM (invalid operation), DM (denormalized operand), ZM (divide by zero), OM (overflow), UM (underflow), and PM (precision error). If masked, the error will not cause an interrupt but, instead, the 8087 will perform a standard response and then proceed with the next instruction in sequence. (For descriptions of the standard responses, see Reference 1.) In particular, the precision error, for which the standard response is to "return the rounded result," should be masked for floating point arithmetic because for most applications precision errors will occur most of the time. Precision errors are of consequence only in special situations.

Whether an interrupt request, including error-related requests, will be generated also depends on the interrupt enable mask bit (IEM) in the control register. When this bit is set to 1, all interrupts are disabled except when the CPU is executing a WAIT instruction. If IEM is 0, an unmasked error can cause an interrupt to be sent to the CPU so that the error can be handled by an error-handling-interrupt routine. In the error-handling routine, the current instruction pointer and operand

pointer, which are stored in the 8087, can be examined by the 8086/8088 by first putting them into memory. This can be done by using the appropriate 8087 instructions (see Sec. 11-3-3). The contents of these pointers identify the instruction and operand addresses when the error occurred. Note that only the least significant 11 bits of the instruction code are kept in the instruction pointer register because the most significant 5 bits are always 11011, the op code of an ESC instruction.

The remaining bits in the control register provide flexibility in controlling precision (PC), rounding (RC), and infinity representations (IC). These bits are defined as follows:

PC bits—00 means 24-bit precision.

01 is reserved.

10 means 53-bit precision.

11 means 64-bit precision.

RC bits—00 means round to nearest.

01 means round toward $-\infty$.

10 means round toward $+\infty$.

11 means truncation.

IC bit—0 indicates that $+\infty$ and $-\infty$ are treated as a single unsigned infinitive.

1 indicates that $+\infty$ and $-\infty$ are treated as two signed infinitives.

During a reset or initialization of the 8087, it sets the PC bits to 11, RC bits to 00, IC bit to 0, IEM bit to 0, and all error mask bits to 1.

The tag register holds the status of the contents of the data registers. Each data register is associated with two tag bits, which indicate whether its contents are valid (00), zero (01), a special value—i.e., NAN, infinity, or denormal—(10), or empty (11).

11-3-3 Instruction Set

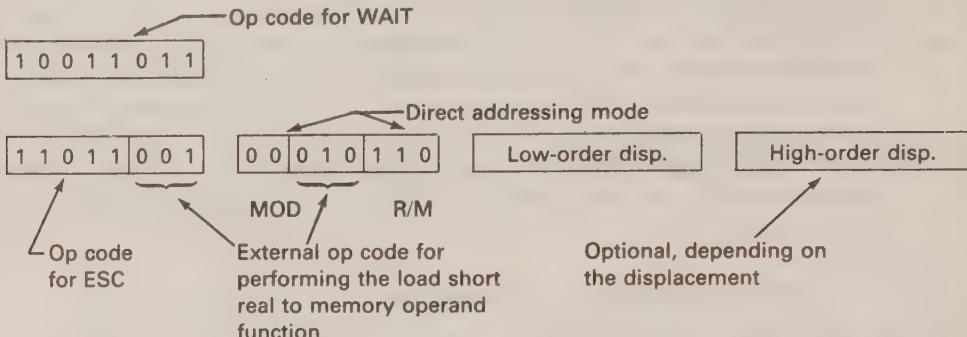
The 8087 has 68 instructions, which can be divided into six groups according to their functions. These six groups are referred to as the data transfer, arithmetic, comparison, transcendental, constant, and processor control groups.

Because the 8087 and the host CPU share the same instruction stream, the ASM-86 assembler allows the user to write programs in a super instruction set which consists of the 8086's and the 8087's instructions. For each 8087 instruction, the assembler generates two machine instructions, a wait instruction followed by an ESC instruction whose external code indicates the 8087's instruction. For example, assuming that SHORT_REAL is defined by a DD directive, the instruction

FST SHORT_REAL

which converts the contents of the top of the ST stack to the short real format and

stores the result into SHORT_REAL, will be assembled as



The WAIT instruction preceding the ESC instruction causes the 8086/8088 to enter a wait state until its TEST pin becomes active, and is necessary to prevent the ESC instruction from being decoded before the 8087 completes its current operation. If the TEST pin is already active or when it becomes active, the ESC instruction will be decoded by the 8087 and then both the 8086 and 8087 will proceed in parallel.

However, a WAIT must be explicitly included whenever the CPU wants to access a memory operand involved in the previous 8087's instruction. For example, in the following sequence the CPU must wait until the 8087 has returned its result to SHORT_REAL before it can execute the CMP instruction:

| | |
|------|----------------------------------|
| FST | SHORT_REAL |
| . | |
| . | |
| . | |
| WAIT | 8086's instructions which do not |
| CMP | use SHORT_REAL |
| JG | |
| . | |
| . | |
| | |
| | |

Note that Intel has a software package that emulates all 8087 instructions. For a program to be executed by emulation, the FWAIT instruction, which is an alternate mnemonic for the WAIT instruction, must be used for synchronization. Because there is no 8087 to activate the TEST pin for emulated execution, the package will change any FWAIT or inserted WAIT to a NOP to avoid an endless wait. However, explicit WAIT instructions are not eliminated from the user's object code.

Let us now consider the definitions of the 8087's assembler language instructions. Many of the assembler instructions allow the user to specify operands in more than one way, either explicitly or implicitly. For such instructions, slashes will be employed to separate the alternate operand forms. For instance, the operand field denoted

//SRC/DST,SRC

means that the instruction may be coded in any one of the following three forms:

1. Both the source and destination operands are implicit (this is indicated by having nothing between the first two slashes).
2. The source operand is specified and the destination operand is implicit.
3. Both the source and destination operands are specified by the user.

An operand may be a register in the register stack or a memory operand. For a register operand ST represents the top stack element, and ST(i) means the ith register below the top stack element (i.e., the register corresponding to ST + i). A memory operand may be addressed by any one of the 8086's data addressing modes. The data type of a memory operand may be word integer, short integer, long integer, packed BCD, short real, long real, or temporary real.

The data transfer group includes the nine instructions which are summarized in Fig. 11-22. A memory operand whose type is word integer, short integer, long integer, short real, long real, temporary real, or packed BCD can be converted to temporary real and then pushed onto the register stack, or a register can be pushed onto the stack. Conversely, the contents of the top stack element can be converted from temporary real to the destination's data format, and then stored in memory. In addition, instructions are available for popping the contents of a register from the stack and exchanging the contents of a register with the top of the stack. The tag register is updated following each data transfer instruction.

The 8087 has a variety of instructions for performing the arithmetic operations addition, subtraction, multiplication, and division. Results are always stored on the top of the register stack or in a specified register, and one of the source operands must be the top of the stack. For real arithmetic instructions, the other operand may be located in a specified register or in memory. Special forms are provided to pop the stack after the arithmetic operation is completed. Because data are internally represented in the temporary real format, arithmetic involving operands of other types can always be accomplished by first loading the operands into the appropriate registers using the data transfer instructions, and then performing the real arithmetic operation. However, arithmetic instructions are provided that will accept memory operands of the short real, long real, temporary real, word integer, or short integer type, and automatically convert these operands to temporary real before performing their operations.

In addition to the basic arithmetic operations, the 8087 provides instructions to calculate the square root, adjust scale values using the power of 2, perform modulo division, round real values to integer values, extract the exponent and fraction, take the absolute value, and change the sign. These instructions operate on the top one or two stack elements, and the result is left on the top of the stack. Figure 11-23 summarizes the arithmetic instructions.

The comparison instructions compare the top of the register stack with the source operand, which may be in another register or in memory, and set the condition codes accordingly. The top stack element may also be compared to 0,

| Name | Mnemonic and Operand Forms | Description and Possible Errors |
|-----------------------|---|--|
| Load real | FLD SRC (SRC: ST(i) or memory operand of the short real, long real, or temporary real type) | Decrement ST, convert the contents of SRC to temporary real, and put the result in (ST). Errors: I,D |
| Load integer | FILD SRC (SRC: word integer, short integer, or long integer memory operand) | Decrement ST, convert the contents of SRC to temporary real, and put the result in (ST). Error: I |
| Load BCD | FBLD SRC (SRC: packed decimal memory operand) | Decrement ST, convert the contents of SRC to temporary real, and put the result in (ST). Error: I |
| Store real | FST DST (DST: ST(i) or memory operand of the short real or long real type) | Convert (ST) to the destination's real format and store the result in DST. Errors: I,O,U,P |
| Store integer | FIST DST (DST: word integer or short integer memory operand) | Convert (ST) to integer and store the result in DST. Errors: I,P |
| Store BCD and pop | FBSTP DST (DST: packed decimal memory operand) | Convert (ST) to packed decimal, store the result in DST, and increment ST. Error: I |
| Store real and pop | FSTP DST (DST: ST(i) or memory operand of the short real, long real, or temporary real type) | Convert (ST) to the destination's real format, store the result in DST, and increment ST. Errors: I,O,U,P |
| Store integer and pop | FISTP DST (DST: word integer, short integer, or long integer memory operand) | Convert (ST) to integer, store the result in DST, and increment ST. Errors: I,P |
| Exchange registers | FXCH //DST (DST: ST(i), if not specified, ST(1) is assumed) | Interchange the contents of the destination register with (ST). Error: I |

Figure 11-22 Data transfer instructions.

or examined to determine its tag, sign, or normalization. Condition code settings can be examined by the 8086 or 8088 by storing the status register of the 8087 in memory using a proper 8087 processor control instruction. There are seven instructions in the compare group and they are defined in Fig. 11-24.

The five transcendental instructions calculate $\tan \theta$ ($0 < \theta < \pi/4$), $\tan^{-1}(Y/X)$ ($0 < Y < X < \infty$), $2^X - 1$ ($0 \leq X \leq 0.5$), $Y \cdot \log_2 X$, and $Y \cdot \log_2(X+1)$ and are defined in Fig. 11-25. All transcendental instructions use ST or ST(1) as their operand(s), and the results are stored back on the stack. Other common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic, and exponential functions can be derived from these five functions through mathe-

matical identities. For example, calculation of the natural log of X is equivalent to

$$\left(\frac{1}{\log_2 e}\right) \log_2 X$$

As an example involving the trigonometric functions, consider the partial tangent instruction, FPTAN, which computes $\tan \theta$, where θ in radians is the (ST) and is between 0 and $\pi/4$. The result is a ratio Y/X , with Y replacing θ and X being

Figure 11-23 Arithmetic instructions.

| Name | Mnemonic and Operand Forms (See note for operand forms) | Description and Possible Errors |
|--------------------------------|---|--|
| Add real | FADD //SRC/DST,SRC (See note for operand forms) | (DST) \leftarrow (DST)+(SRC) Errors: I,D,O,U,P |
| Add real and pop | FADDP DST,SRC (DST:ST(i), SRC:ST) | (ST(i)) \leftarrow (ST(i))+(ST) Increment ST Errors: I,D,O,U,P |
| Add integer | FIADD SRC (SRC: word integer or short integer memory operand) | (ST) \leftarrow (ST)+(SRC) Errors: I,D,O,P |
| Subtract real | FSUB //SRC/DST,SRC (See note for operand forms) | (DST) \leftarrow (DST)-(SRC) Errors: I,D,O,U,P |
| Subtract real and pop | FSUBP DST,SRC (DST:ST(i), SRC:ST) | (ST(i)) \leftarrow (ST(i))-(ST) Increment ST Errors: I,D,O,U,P |
| Subtract real reversed | FSUBR //SRC/DST,SRC (See note for operand forms) | (DST) \leftarrow (SRC)-(DST) Errors: I,D,O,U,P |
| Subtract real reversed and pop | FSUBRP DST,SRC (DST:ST(i), SRC:ST) | (ST(i)) \leftarrow (ST)-(ST(i)) Increment ST Errors: I,D,O,U,P |
| Subtract integer | FISUB SRC (SRC: word integer or short integer memory operand) | (ST) \leftarrow (ST)-(SRC) Errors: I,D,O,P |
| Subtract integer reversed | FISUBR SRC (SRC: word integer or short integer memory operand) | (ST) \leftarrow (SRC)-(ST) Errors: I,D,O,P |
| Multiply real | FMUL //SRC/DST,SRC (See note for operand forms) | (DST) \leftarrow (DST)*(SRC) Errors: I,D,O,U,P |
| Multiply real and pop | FMULP DST,SRC (DST:ST(i), SRC:ST) | (ST(i)) \leftarrow (ST(i))*(ST) Increment ST Errors: I,D,O,U,P |
| Multiply integer | FIMUL SRC (SRC: word integer or short integer memory operand) | (ST) \leftarrow (ST)*(SRC) Errors: I,D,C,P |
| Divide real | FDIV //SRC/DST,SRC (See note for operand forms) | (DST) \leftarrow (DST)/(SRC) Errors: I,D,Z,O,U,P |
| Divide real and pop | FDIVP DST,SRC (DST:ST(i), SRC:ST) | (ST(i)) \leftarrow (ST(i))/(ST) Increment ST Errors: I,D,Z,O,U,P |
| Divide real reversed | FDIVR //SRC/DST,SRC (See note for operand forms) | (DST) \leftarrow (SRC)/(DST) Errors: I,D,Z,O,U,P |

| Name | Mnemonic and Operand Forms | Description and Possible Errors |
|-------------------------------|---|--|
| Divide real reversed and pop | FDIVRP DST,SRC (DST:ST(i), SRC:ST) | (ST(i)) \leftarrow (ST)/(ST(i)) Increment ST Errors: I,D,Z,O,U,P |
| Divide integer | FIDIV SRC (SRC: word integer or short integer memory operand) | (ST) \leftarrow (ST)/(SRC) Errors: I,D,Z,O,U,P |
| Divide integer reversed | FIDIVR SRC (SRC: word integer or short integer memory operand) | (ST) \leftarrow (SRC)/(ST) Errors: I,D,Z,O,U,P |
| Absolute value | FABS (No operands) | (ST) \leftarrow (ST) Error: I |
| Change sign | FCHS (No operands) | (ST) \leftarrow -(ST) Error: I |
| Partial remainder | FPREM (No operands) | (ST) \leftarrow (ST) mod (ST(1)) Errors: I,D,U |
| Round to integer | FRNDINT (No operands) | (ST) \leftarrow integer of (ST) (Rounding mode is determined by RC field of the control word.) Errors: I,P |
| Scale | FSCALE (No operands) | (ST) \leftarrow (ST) $\cdot 2^n$ where n is the integral part of (ST(1)) Errors: I,O,U |
| Square root | FSQRT (No operands) | (ST) $\leftarrow \sqrt{(ST)}$ Errors: I,D,P |
| Extract exponent and fraction | FXTRACT (No operands) | (Temp Reg 1) \leftarrow exponent of (ST) (Temp Reg 2) \leftarrow fraction of (ST) (ST) \leftarrow (Temp Reg 1) Decrement ST (ST) \leftarrow (Temp Reg 2) Error: I |

Note: If DST is not specified, ST is assumed and SRC must be a memory operand of the short real or long real type. If both are specified, they must be ST,ST(1) or ST(i),ST. When neither is given, then ST(1),ST is assumed and, after the arithmetic operation is performed, the stack is popped and the result is left at the new top of the stack.

Figure 11-23 Continued.

pushed onto the stack. The sine function is related to the tangent function as follows:

$$\sin \theta = \frac{\tan \theta}{\sqrt{1 + \tan^2 \theta}}$$

When $0 < \theta < \pi/4$, $\sin \theta$ can be calculated by

$$\frac{Y}{\sqrt{X^2 + Y^2}}$$

If θ is outside its acceptable range, then some procedure must be used to obtain a suitable angle that is inside the 0 to $\pi/4$ range. For example, the instruction

FPREM and the identity

$$\tan \theta = \frac{1}{\tan\left(\frac{\pi}{2} - \theta\right)}$$

can be used to reduce an angle in the range $\pi/4$ to $\pi/2$ to the valid range (see Exercise 10).

Figure 11-24 Compare instructions.

| Name | Mnemonic and Operand Forms | Description and Possible Errors |
|----------------------------|---|---|
| Compare real | FCOM //SRC (SRC:ST(i), or memory operand of the short real or long real type. If not specified, ST(1) is assumed) | (ST)-(SRC), and set condition codes as in Note 1. Errors: I,D |
| Compare real and pop | FCOMP //SRC (SRC:ST(i), or memory operand of the short real or long real type. If not specified, ST(1) is assumed) | (ST)-(SRC), and set condition codes as in Note 1. Increment ST Errors: I,D |
| Compare real and pop twice | FCOMPP (No operands) | (ST)-(ST(1)), and set condition codes as in Note 1. Increment ST by 2 Errors: I,D |
| Compare integer | FICOM SRC (SRC: word integer or short integer memory operand) | (ST)-(SRC), and set condition codes as in Note 1. Errors: I,D |
| Compare integer and pop | FICOMP SRC (SRC: word integer or short integer memory operand) | (ST)-(SRC), and set condition codes as in Note 1. Increment ST Errors: I,D |
| Test top of stack | FTST (No operands) | (ST)-0.0, and set condition codes as in Note 2. Errors: I,D |
| Examine top of stack | FXAM (No operands) | Set condition codes according to (ST) as indicated in Note 3. |

Note 1:

| Order | C3 | C0 | (ST) |
|----------------|----|----|-----------|
| (ST) > (SRC) | 0 | 0 | +Unnormal |
| (ST) < (SRC) | 0 | 1 | +NAN |
| (ST) = (SRC) | 1 | 0 | -Unnormal |
| Not comparable | 1 | 1 | -NAN |

Note 3:

| Order | C3 | C2 | C1 | C0 |
|----------------|----|----|----|----|
| (ST) > (SRC) | 0 | 0 | 0 | 0 |
| (ST) < (SRC) | 0 | 1 | 0 | 1 |
| (ST) = (SRC) | 1 | 0 | 1 | 0 |
| Not comparable | 1 | 1 | 1 | 1 |

Note 2:

| Order | C3 | C0 | (ST) |
|----------------|----|----|-----------|
| (ST) > 0.0 | 0 | 0 | +0 |
| (ST) < 0.0 | 0 | 1 | -0 |
| (ST) = 0.0 | 1 | 0 | Empty |
| Not comparable | 1 | 1 | +Denormal |

| Name | Mnemonic | Description and Possible Errors |
|---------------------------------|--------------------------|---|
| Calculate 2^{X-1} | F2XM1 (No operands) | Assuming $0 \leq (ST) \leq 0.5$, $(ST) \leftarrow 2^{\frac{X}{2}} - 1$ Errors: U, P |
| Partial arctangent | FPATAN (No operands) | Assuming $0 < (ST(1)) < (ST) < \infty$, (Temp Reg 1) $\leftarrow \tan^{-1}[(ST(1)/(ST))]$ Increment ST $(ST) \leftarrow (\text{Temp Reg 1})$ Errors: U, P |
| Partial tangent | FPTAN (No operands) | Assuming $0 < (ST) < \frac{\pi}{4}$, $(ST) \leftarrow \text{Numerator of } \tan[(ST)]$ Decrement ST $(ST) \leftarrow \text{Denominator of } \tan[(ST)]$ Errors: I, P |
| Calculate $Y \cdot \log_2 X$ | FYL2X (No operands) | Assuming $0 < (ST) < \infty$ and $-\infty < (ST(1)) < \infty$, (Temp Reg 1) $\leftarrow (ST(1)) \cdot \log_2[(ST)]$ Increment ST $(ST) \leftarrow (\text{Temp Reg 1})$ Error: P |
| Calculate $Y \cdot \log_2[X+1]$ | FYL2XP1 (No operands) | Assuming $0 < (ST) < 1 + \sqrt{2}/2$ and $-\infty < (ST(1)) < \infty$, (Temp Reg) $\leftarrow (ST(1)) \cdot \log_2[(ST)+1]$ Increment ST $(ST) \leftarrow (\text{Temp Reg})$ Error: P |

Figure 11-25 Transcendental instructions.

The 8087 provides seven instructions for generating constants. They are used to push $+0.0$, $+1.0$, π , $\log_{10}10$, $\log_e e$, $\log_{10}2$, or $\log_2 2$ onto the register stack. These constants are stored in the temporary real format and have an accuracy of up to approximately 19 decimal digits. The constant instructions are summarized in Fig. 11-26.

The last group of instructions is shown in Fig. 11-27. They are for manipulating the control and status registers, and saving and restoring various portions of the processor's state. Normally, these instructions are required in subroutines and interrupt service routines which need to use the 8087, or for error-handling and process switching. The initialization instruction resets the 8087 to its initial state and is normally required before using the 8087. Several of the instructions load the control register with new contents, thus permitting the control bits to be dynamically changed as circumstances change.

Different portions of the 8087's current state may be saved in or restored from memory for various reasons. Three actions that are often needed are:

1. **To save the status register only (FSTSW)**—used when the CPU needs to examine the condition code settings after a comparison instruction.
2. **To save the processor environment consisting of the control, status, and tag registers and the instruction and operand pointers (FSTENV)**—used in an error handling routine to identify the cause of the error.
3. **To save the entire processor state consisting of the processor environment plus the register stack (FSAVE)**—used in subroutines, interrupt service routines, and process switching.

| Name | Mnemonic | Description and Possible Errors |
|---------------------|-------------------------|--|
| Load zero | FLDZ (No operands) | Decrement ST (ST) \leftarrow 0.0 Error: I |
| Load one | FLD1 (No operands) | Decrement ST (ST) \leftarrow 1.0 Error: I |
| Load π | FLDPI (No operands) | Decrement ST (ST) $\leftarrow \pi$ Error: I |
| Load $\log_2 e$ | FLDL2E (No operands) | Decrement ST (ST) $\leftarrow \log_2 e$ Error: I |
| Load $\log_{10} 10$ | FLDL2T (No operands) | Decrement ST (ST) $\leftarrow \log_{10} 10$ Error: I |
| Load $\log_{10} 2$ | FLDLG2 (No operands) | Decrement ST (ST) $\leftarrow \log_{10} 2$ Error: I |
| Load $\log_e 2$ | FLDLN2 (No operands) | Decrement ST (ST) $\leftarrow \log_e 2$ Error: I |

Figure 11-26 Constant instructions.

In order to terminate the 8087's requests, the FSTENV and FSAVE instructions set all error masks in the 8087 to 1 (via initialization) after the registers are stored in memory. (The interrupt enable mask (IEM) bit is also set to 1 by FSAVE so that all interrupts are disabled.) The saved image can be restored with the FLDENV or FRSTOR instructions. However, in an error-handling routine it is necessary to set all saved error masks to one before the status is loaded into the 8087. Otherwise, an interrupt will occur immediately.

11-3-4 An Example

In statistical applications, a very important measure of observed random samples is the standard deviation which indicates the dispersion of these samples. For a given set of samples, X_1, X_2, \dots, X_N , the standard deviation is defined as

$$\sqrt{\frac{\sum_{i=1}^N (X_i - \text{MEAN})^2}{N - 1}}$$

where N is the number of samples and MEAN is the sample mean,

$$\text{MEAN} = \frac{\sum_{i=1}^N X_i}{N}$$

We assume that the samples have been input, converted to the long real data format, and stored in an array beginning at ARRAY_X, and that the variable N

| Name | Mnemonics and Operand Forms ¹ | Description ² |
|-------------------------|---|---|
| Initialize processor | FINIT/FNINIT (No operands) | Reset the 8087. Set control word to 03FF and set tag register to empty. Clear all error, busy, and interrupt flags. Reset ST to 0. |
| Disable interrupts | FDISI/FNDISI (No operands) | Set interrupt mask bit to 1. |
| Enable interrupts | FENI/FNENI (No operands) | Clear interrupt mask bit. |
| Clear error flags | FCLEX/FNCLEX (No operands) | Clear B,IR,P,U,O,Z,D, and I in the status register. |
| Increment stack pointer | FINCSTP (No operands) | ST ← ST+1 (It does not set the tag of the previous top of stack register to empty.) |
| Decrement stack pointer | FDECSTP (No operands) | ST ← ST-1 |
| Store status register | FSTSW/FNSTSW DST (DST: 2-byte memory operand) | (DST) ← (Status register) |
| Store control register | FSTCW/FNSTCW DST (DST: 2-byte memory operand) | (DST) ← (Control register) |
| Load control register | FLDCW SRC (SRC: 2-byte memory operand) | (Control register) ← (SRC) |
| Store environment | FSTENV/FNSTENV DST (DST: 14-byte memory operand) | (DST) ← (Control register) (DST+2) ← (Status register) (DST+4) ← (Tag register) (DST+6) ← (Instruction pointer) (DST+10) ← (Operand pointer) Sets all error masks. |
| Load environment | FLDENV SRC (SRC: 14-byte memory operand) | (Control register) ← (SRC) (Status register) ← (SRC+2) (Tag register) ← (SRC+4) (Instruction pointer) ← (SRC+6) (Operand pointer) ← (SRC+10) |

Figure 11-27 Processor control instructions.

contains the number of samples. Figure 11-28 gives a program sequence for computing the sample mean and the standard deviation, and storing them in MEAN and STD_DEV, respectively.

The first instruction initializes the 8087 so that all the registers are marked empty, the error and interrupt flags are cleared, the error masks are set, ST is set to 0, and the default rounding, precision, and infinity controls are set. If this program sequence is part of a procedure, it would be necessary to save the state for the calling program before the initialization is done.

The 8087 relies on the CPU to perform looping, conditional branching, and indexing. With a proper program design, maximum parallelism between the two processors can be achieved. For example, it is better to place XOR and MOV after FLDZ because the execution time of FINIT is faster than that of FLDZ.

| Name | Mnemonics and Operand Forms ¹ | Description ² |
|---------------|---|---|
| Save state | FSAVE/FNSAVE DST (DST: 94-byte memory operand) | Save control register, status register, tag register, instruction pointer, operand pointer, ST, ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7) into DST. Initialize the processor. |
| Load state | FRSTOR SRC (SRC: 94-byte memory operand) | Load control register, status register, tag register, instruction pointer, operand pointer, ST, ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7) from SRC. |
| Free register | FFREE DST (DST: ST(i)) | Mark the tag of ST(i) to empty. |
| No operation | FNOP (No operands) | (ST) ←— (ST) |

¹The alternate mnemonics with a second character of N are intended for use in situations in which a WAIT instruction is not needed. Thus, the assembler does not prefix the instruction with a WAIT instruction.

²No errors are possible.

Figure 11-27 Continued.

It is important to note that a memory operand can only be loaded into the 8087 by a push operation. Therefore, after each iteration of the loop beginning at LOOPSTD, the first three registers from the top of the stack contain $(X_i - \text{MEAN})^2$, the partial sum of $\sum_{i=1}^N (X_i - \text{MEAN})^2$, and the contents of MEAN. In order to load X_{i+1} in the same top of the stack element at the beginning of the next iteration,

Figure 11-28 Calculation of sample mean and standard deviation.

Other code for directives, data segments, register setup, input, and number conversions

```

FINIT          ;ALL ERROR MASKS ARE SET TO 1
FLDZ           ;PUSH 0.0 TO ST, USED AS ACCUMULATOR
XOR  BX,BX     ;CLEAR BX AS INDEX
MOV  CX,N      ;USE CX AS COUNTER
LOOPMEAN:
FADD ARRAY_X[BX] ;ADD X(I) TO ST
ADD  BX,8       ;POINT TO NEXT ELEMENT
LOOP  LOOPMEAN  ;REPEAT FOR N TIMES, ST HAS SUM OF X(I)
FIDIV N        ;(ST) = (SUM OF X(I))/N
FST   MEAN      ;STORE MEAN
FLDZ           ;PUSH 0.0 TO ST
XOR  BX,BX     ;CLEAR INDEX
MOV  CX,N      ;
LOOPSTD:
FLD   ARRAY_X[BX] ;PUSH X(I)
ADD  BX,8       ;
FSUB ST,ST(2)   ;(ST) = X(I)-MEAN
FMUL ST,ST(0)   ;(ST) = (X(I)-MEAN)**2
FADDP ST(1),ST  ;
LOOP  LOOPSTD   ;(ST) = SUM OF (X(I)-MEAN)**2
FLD1            ;PUSH 1 TO ST
FISUBR N        ;(ST) = N-1.0
FDIVR ST,ST(1)   ;(ST) = ((SUM OF (X(I)-MEAN)**2))/(N-1)
FSQRT           ;(ST) = STANDARD DEVIATION
FST   STD_DEV    ;STORE (ST) IN STD_DEV

```

Other code for output and number conversions

the “pop” form of the add instruction (FADDP) should be used. This removes $(X_i - \text{MEAN})^2$ after it has been added into ST(1):

11-4 THE 8089 I/O PROCESSOR

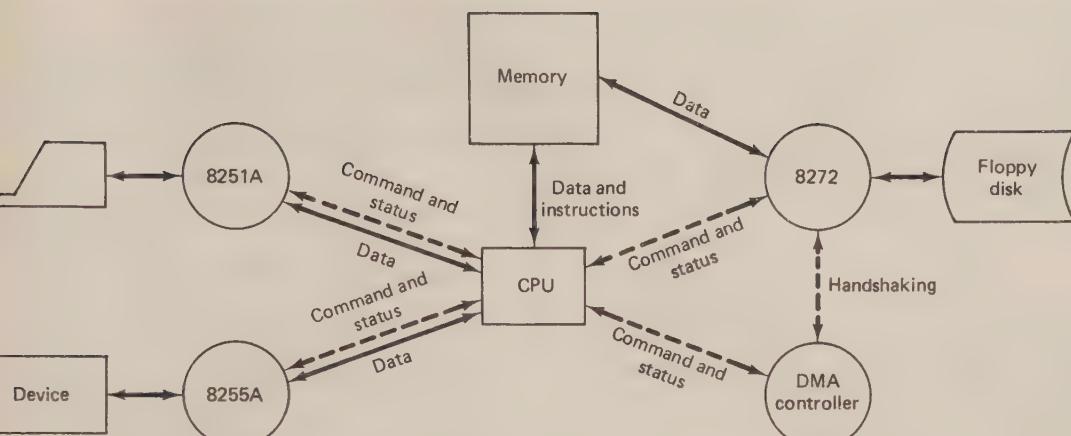
Controller and I/O interface devices such as those discussed in Chap. 9 significantly simplify interface design, but for non-DMA transfers the CPU is required to set up and perform the actual data transfer. For high-speed devices, data are transferred using DMA, but the CPU still needs to set up the device controller, initiate the DMA operation, and examine the post-transfer status after the completion of each DMA operation. Figure 11-29(a) illustrates the general command and data flow when I/O is handled by the CPU.

The 8089 I/O processor (IOP) is designed to handle the details involved in I/O processing. Unlike a DMA controller, an IOP can fetch and execute its own instructions. These instructions are specifically tailored for I/O operations, but in addition to data transfer, they can perform arithmetic and logic operations, branches, searching, and translation.

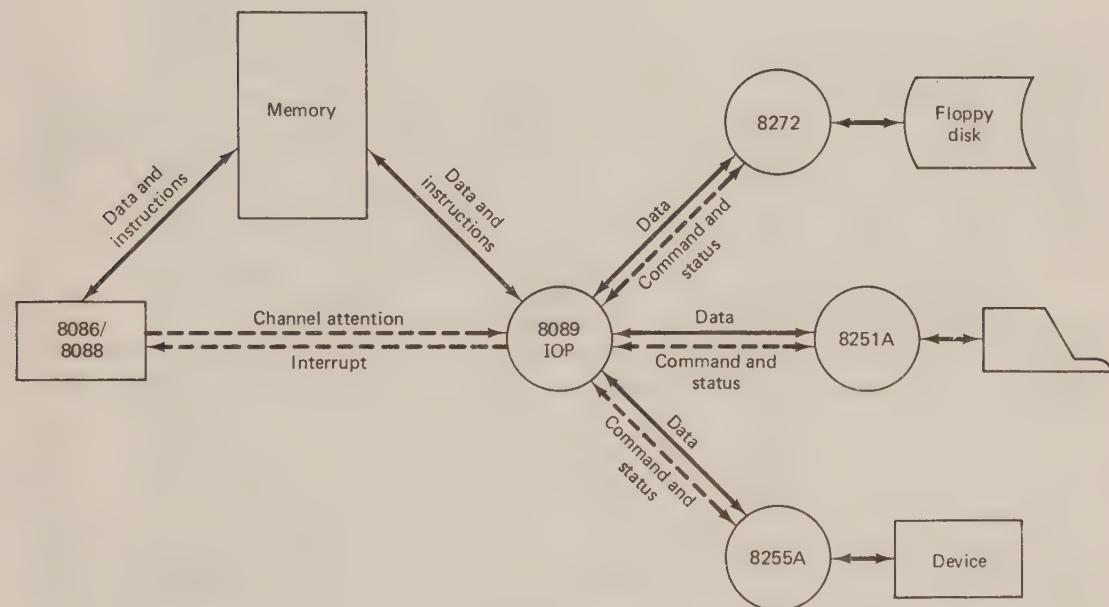
The CPU communicates with the 8089 through memory-based control blocks. The CPU prepares control blocks that describe the task to be performed, and then dispatches the task to the IOP through an interrupt-like signal. The IOP reads the control blocks to locate a program sequence, called a *channel program*, which is written in the 8089's instruction set. Then the 8089 performs the assigned task by fetching and executing instructions from the channel program. When it has finished, the IOP notifies the CPU either through an interrupt or by updating a status location in memory.

As shown in Fig. 11-29(b), the IOP assumes all of the work involved in an I/O transfer including device setup, programmed I/O, and DMA operation, thereby removing the burden of the I/O processing from the CPU. Such a design allows the CPU to concentrate on higher-level tasks while the IOP is dedicated to I/O processing. This distributed processing approach greatly simplifies system software and hardware efforts, yet improves system performance and flexibility. The 8089 may be operated in a local (closely coupled) configuration or a remote (loosely coupled) configuration. As shown in Figs. 11-7 and 11-8, in a local configuration the IOP shares the bus interface with the host by using its $\overline{\text{RQ}}/\overline{\text{GT}}$ pins. All resources are accessed through the system bus.

In a remote configuration, the IOP may have its own local I/O bus and requires a bus arbiter and controller, address latches, and data transceivers for accessing the shared system bus. An example is illustrated in Fig. 11-30. The $\overline{\text{RQ}}/\overline{\text{GT}}$ pin on the IOP could be used to interact with another IOP, which acts as the slave and shares the buses with the host IOP. The IOP accesses the I/O devices dedicated to it through the local bus while it communicates with the CPU through the system memory. A high-speed controller would request a transfer through one of two DRQ pins and terminate a DMA operation through one of two EXT pins. To reduce the system bus loading and enhance concurrent processing, local memory



(a) I/O handled by microprocessor



(b) I/O handled by 8089

Figure 11-29 I/O subsystems.

can be included to store the channel programs or to provide storage areas. However, the local memory must respond to I/O bus commands instead of memory read and memory write commands (i.e., it must be I/O mapped memory).

Unlike the 8086 or 8088, the IOP's I/O bus need not have the same data width as the memory bus. This allows the IOP to transfer data from an 8-bit source

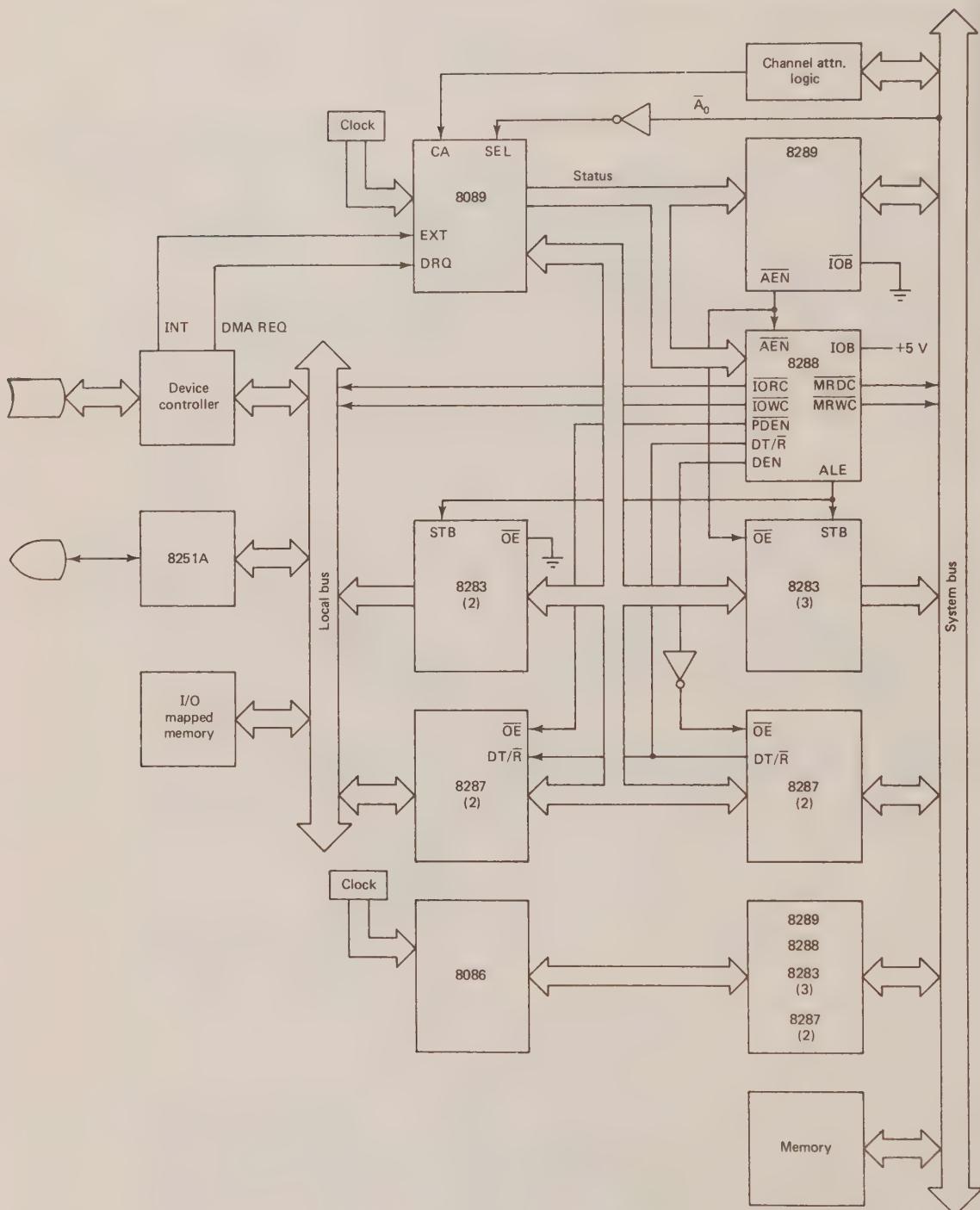


Figure 11-30 Remote configuration.

to a 16-bit destination, and vice versa. Because the I/O bus has only 16 address lines, the capacity of the local space (I/O space) is only 64K bytes. On the other hand, the system space (memory space), which is addressed by the system bus, has a capacity of 1M bytes. Furthermore, the IOP instructions access I/O ports using the same addressing modes as are used for memory operands. Whether an address is in the I/O space or in the system space is determined by the tag bit of the pointer register used.

11-4-1 IOP Architecture

Figure 11-31 shows the internal structure of the 8089. Each of the two channels can be programmed and operated independently while sharing the common control logic and ALU. The channel control pointer (CCP) cannot be manipulated by the user. It stores the address of the control block (CB) for channel 1 during the initialization sequence. For channel 2, its CB starts at the address that is indicated by the contents of the CCP plus 8. To dispatch a task to either channel, the CPU sends out a channel attention signal along with a SEL signal which selects channel 1 (SEL = Low) or channel 2 (SEL = High). Since the channels occupy two consecutive I/O port addresses, the A0 address line is connected to the SEL pin.

Each channel has an identical set of registers, each set being broken into two groups according to size. The pointer group consists of those registers having 20 bits, and the register group of those having 16 bits. Each pointer, with the exception of PP, has an associated tag bit. When used to access a memory operand, the tag bit indicates whether the contents of that pointer represents a 20-bit system (memory) space address (tag bit = 0) or a 16-bit local (I/O) space address (tag bit = 1). In accessing the local space, only the low-order 16 bits of the pointer are used as the address. Register PP always points to an address in the system space.

The registers GA, GB, GC, BC, IX, and MC can be used as general-purpose registers in a channel program for arithmetic and logic operations. In addition, they perform special functions when addressing memory operands and executing DMA operations. A memory operand can only be addressed by using one of the pointers GA, GB, GC, or PP as a base register. During a DMA operation, GA and GB are used for the source and destination pointers. If GA points to the source, then GB points to the destination, and vice versa. When a translation operation is performed along with a DMA transfer, the contents of GC are used as the base address of a 256-byte translation table. During a DMA transfer, register BC is used as the byte counter and is decremented by 1 after each byte transfer and by 2 after each word transfer. For a masked compare operation, register MC contains the bit pattern to be compared against in bits 7 through 0 and a mask in bits 15 through 8. A masked compare is executed according to the expression:

$$((OPERAND\ BYTE) \oplus (MC)_{7-0}) \wedge (MC)_{15-8}$$

The results can be used as a DMA termination condition or to determine whether or not a branch is to be made by a masked compare branch instruction. The register IX is used as an index register. In two of the memory operand addressing modes, its contents are added to those of a base register to form the operand address.

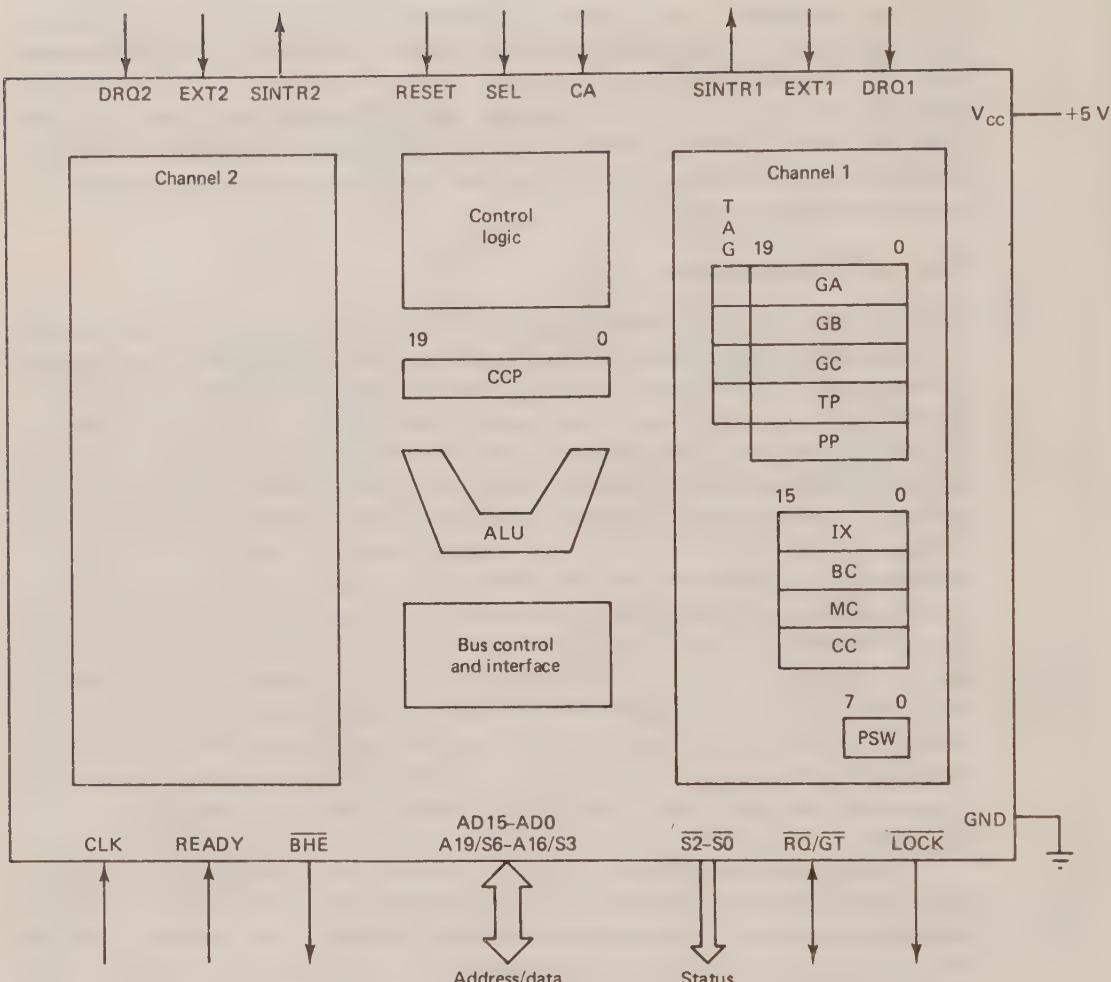


Figure 11-31 Block diagram of the 8089 IOP.

The task pointer (TP) stores the address of the next instruction to be executed and is equivalent to the PC in a CPU. It also has a tag bit for indicating whether the next instruction is stored in the system or I/O space. The parameter pointer (PP) is not programmable by the user, but is automatically filled by the 8089 while initializing a task. It points to the address of the parameter block, which is discussed later.

Each channel also has an 8-bit program status register (PSW) which contains the current channel status. This status concerns such things as the source and destination address widths, channel activity, interrupt control and servicing, bus load limit, and priority. The PSW cannot be manipulated by the user, but can be updated by a channel command. It is saved along with the TP pointer and the 4 tag bits in the first two words of the parameter block when a channel program is

suspended. This allows the channel to resume the suspended channel program upon receipt of a resume command.

One of the unique features of the IOP is its capability to perform DMA transfers using a variety of options. The transfer direction can be specified as memory to memory, I/O to I/O, or memory to/from I/O. For each transfer, the IOP fetches a byte or word, stores the data in the destination, and updates GA, GB, and BC accordingly. If data are transferred from an 8-bit source to a 16-bit destination, the IOP can fetch 2 bytes and store them as one word. Conversely, if the transfer is made in the other direction, a word can be broken into 2 bytes before the data are sent to the destination. Between the fetch and store cycles of the DMA, the data byte can be compared or translated. Furthermore, a DMA operation can be terminated by an external request, a match or mismatch detected by a masked compare, or a zero byte count. These options are specified by the contents of the channel control (CC) register whose format is defined as follows:

1. **Function Control (Bits 15 and 14)**—Specifies one of the four data transfer modes: memory to memory (11), I/O port to memory (10), memory to I/O port (01), and I/O port to I/O port (00). During a memory-to-memory data transfer, both the source and destination pointers are auto-incremented, but during and after an I/O to I/O data transfer, both pointers remain unchanged. These two modes are not supported by most conventional DMA controllers. They are useful in moving a block of code or data from one memory area to another and in direct device communications.
2. **Translation Mode (Bit 13)**—Indicates that the data bytes are to be translated through a 256-byte lookup table during DMA (if 1). The base address of the translation table should be stored in GC.
3. **Synchronization Control (Bits 12 and 11)**—Specifies how the data transfer is to be synchronized. An unsynchronized transfer (00) begins the next transfer cycle whenever a bus cycle is available. A source synchronized transfer (01) starts the read operation of the next transfer cycle upon receiving the DRQ signal and a destination synchronized transfer (10) starts the write operation of the next transfer cycle when the DRQ is received.
4. **Source/Destination Indicator (Bit 10)**—Denotes that register GA is used as the source pointer (0) or the destination pointer (1). In either case, GB is used as the other pointer.
5. **Lock Control (Bit 9)**—Activates the 8089's LOCK output during the DMA transfer cycle (if 1).
6. **Chaining Control (Bit 8)**—Gives the other channel the highest priority. (This bit is not specifically used for a DMA operation.)
7. **Single Transfer Mode (Bit 7)**—Terminates the DMA after a single transfer (if 1) and then executes the next instruction pointed by TP.
8. **Termination Control (Bits 6-0)**—Specifies how the DMA is to be terminated and where to fetch the next instruction upon the completion of the DMA operation. As indicated in Fig. 11-32, a DMA transfer can be terminated

| Control bits of CC | | | Termination condition and offset |
|--------------------|---|---|--|
| Bit 6 Bit 5 | | | |
| 0 | 0 | | No external termination |
| 0 | 1 | | Terminates when EXT is high; offset is set to 0 |
| 1 | 0 | | Terminates when EXT is high; offset is set to 4 |
| 1 | 1 | | Terminates when EXT is high; offset is set to 8 |
| Bit 4 Bit 3 | | | |
| 0 | 0 | | No termination by byte counter |
| 0 | 1 | | Terminates when BC=0; offset is set to 0 |
| 1 | 0 | | Terminates when BC=0; offset is set to 4 |
| 1 | 1 | | Terminates when BC=0; offset is set to 8 |
| Bit 2 Bit 1 Bit 0 | | | |
| 0 | 0 | 0 | No termination by masked comparison |
| 0 | 0 | 1 | Terminates when comparison matches; offset is set to 0 |
| 0 | 1 | 0 | Terminates when comparison matches; offset is set to 4 |
| 0 | 1 | 1 | Terminates when comparison matches; offset is set to 8 |
| 1 | 0 | 0 | No effect |
| 1 | 0 | 1 | Terminates when there is no match; offset is set to 0 |
| 1 | 1 | 0 | Terminates when there is no match; offset is set to 4 |
| 1 | 1 | 1 | Terminates when there is no match; offset is set to 8 |

Figure 11-32 Termination of a DMA transfer.

after the current transfer cycle based on external control (bits 6 and 5), byte count (bits 4 and 3), a comparison (bits 2, 1, and 0), or on a combination of these choices. If external control is selected, the channel terminates the DMA when the channel's EXT (external termination) input is activated. If byte count is specified, a "0" in the channel's BC register causes the DMA to terminate. Whether or not a comparison is to result in a termination and whether a match or mismatch is to cause the termination is determined by bits 2 through 0.

Upon the termination of a DMA operation the channel executes the instruction whose address is the contents of TP plus an offset. Therefore, when more than one termination condition is specified it is possible to use the offset in conjunction with a branch instruction to enter different DMA completion routines depending on the actual cause of the DMA termination. If more than one of the selected conditions occur at the same time, the largest offset that corresponds to a satisfied condition is used. To initiate a DMA transfer, the channel program should contain instructions for setting up the source and destination pointers; CC, BC and, if necessary, GC and MC; and the I/O bus width.

11-4-2 Communication between CPU and IOP

Interprocessor communication, including the IOP initialization and task dispatching, is memory based and is accomplished by means of a linked list of control blocks. The first control block in the linked list is stored beginning at the fixed location FFFF6 and the others may reside in user-defined areas, each of which is pointed to by the previous control block.

The IOP communication areas are depicted in Fig. 11-33. The system configuration pointer block (SCPB) contains three words starting at location FFFF6 in the system memory. The least significant byte (SYSBUS) specifies the width of the system bus: 16 bits if SYSBUS = 1 and 8 bits if SYSBUS = 0. The succeeding two words store the offset and segment address of the location of the system configuration block (SCB).

The SCB does not need to be stored at a fixed location; however, it must reside in the system space. The least significant byte of the SCB is the system operation command (SOC). Bits 0 and 1 of SOC define the width of the I/O bus and the $\overline{RQ}/\overline{GT}$ mode as follows:

Bit 0 = 0 indicates an 8-bit I/O bus.

Bit 0 = 1 indicates a 16-bit I/O bus.

Bit 1 = 0 indicates the standard $\overline{RQ}/\overline{GT}$ mode.

Bit 1 = 1 indicates the modified $\overline{RQ}/\overline{GT}$ mode for use with multiple IOPs.

The last two words in the SCB contain the offset and segment address of the beginning of two consecutive channel control blocks (CBs) in the system space. There is one control block for each channel and the first byte of each control block, which is called the channel command word (CCW), indicates the action to be taken by the channel. The next byte (BUSY) indicates the busy status of the channel (FF for busy and 00 for not busy), and the last two words contain the address of a parameter block. A parameter block is for providing the beginning address of the channel program and passing information to and from this program.

The format of a CCW is defined in Fig. 11-34. It includes a 3-bit command field, a 2-bit interrupt control field, a bus load limit bit, and a priority bit. The command field specifies one of the following six possible commands:

Update PSW—Causes the PSW to be updated.

Start Channel Program (I/O Space)—Initiates the execution of a channel program that is stored in the I/O space.

Start Channel Program (System Space)—Initiates the execution of a channel program that is stored in the system space.

Suspend Channel Operation—Suspends the operation currently being conducted by the channel until a resume command is given. (The command is *not* normally used to nest channel programs, but is simply for suspending an I/O operation.)

Resume Channel Operation—Causes a suspended operation to be continued from the point at which it was stopped.

Halt Channel Operation—Aborts the current channel operation.

The interrupt control field is for enabling and disabling interrupt requests and for removing previous requests. When an IOP sends out an interrupt request it sets a

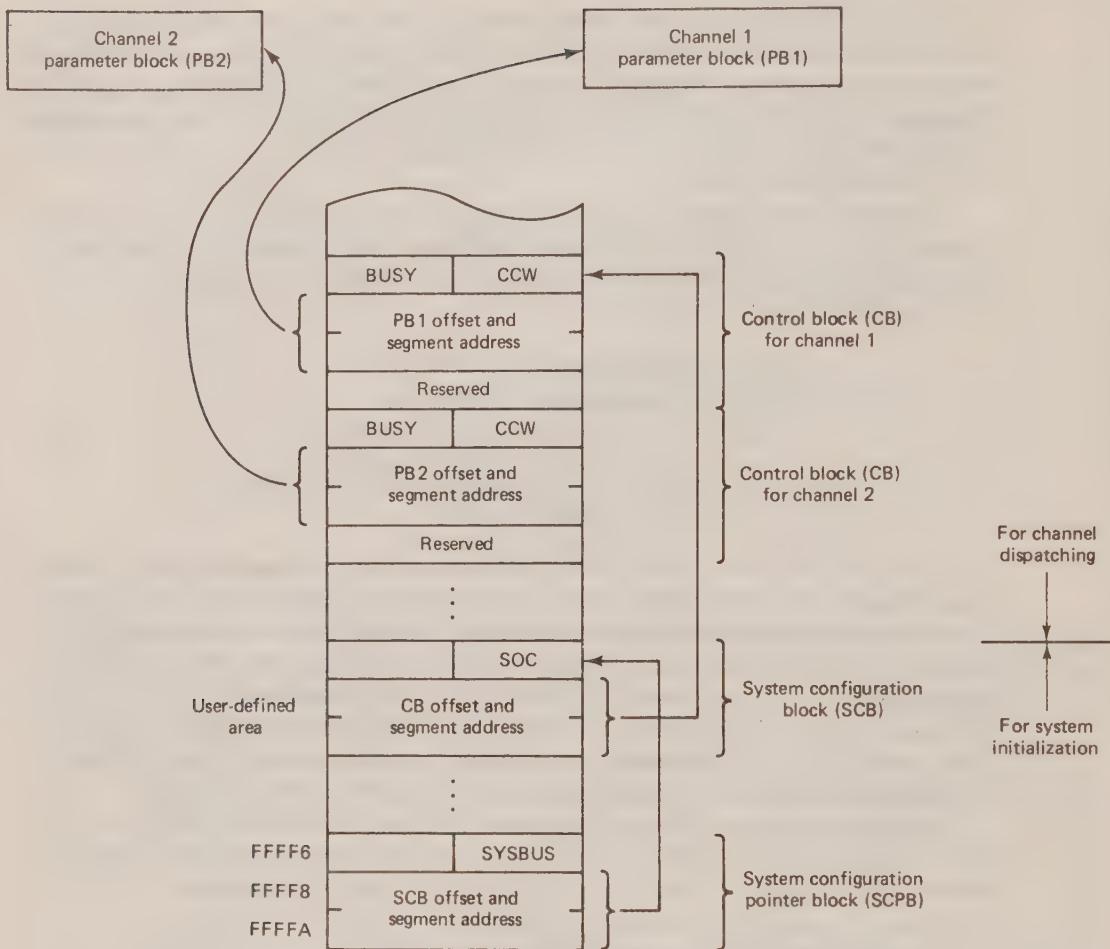
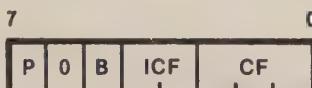


Figure 11-33 Communication structure.

service bit in its PSW. While the interrupt is being serviced, this bit must be cleared by sending the IOP a command with 01 in the interrupt control field; otherwise, a request would block other requests.

When the bus load limit bit is 1, the IOP can execute only one instruction every 128 clock cycles. This prevents the IOP from monopolizing the bus in situations in which there is no need for it to be the dominant processor. The priority bit in the PSW is set or cleared according to the priority bit in the CCW.

Flowcharts illustrating the interaction between a CPU and an IOP are given in Fig. 11-35. The responsibilities of the CPU's program are shown in part (a) of the figure and the actions taken by the IOP are indicated in part (b). It is seen that both the initialization and task phases are initiated by a channel attention (CA). As mentioned previously, the IOP is associated with two addresses and the

**CF COMMAND FIELD**

- 000 UPDATE PSW
- 001 START CHANNEL PROGRAM LOCATED IN I/O SPACE.
- 010 (RESERVED)
- 011 START CHANNEL PROGRAM LOCATED IN SYSTEM SPACE.
- 100 (RESERVED)
- 101 RESUME SUSPENDED CHANNEL OPERATION
- 110 SUSPEND CHANNEL OPERATION
- 111 HALT CHANNEL OPERATION

ICF INTERRUPT CONTROL FIELD

- 00 IGNORE, NO EFFECT ON INTERRUPTS.
- 01 REMOVE INTERRUPT REQUEST; INTERRUPT IS ACKNOWLEDGED.
- 10 ENABLE INTERRUPTS.
- 11 DISABLE INTERRUPTS.

B BUS LOAD LIMIT

- 0 NO BUS LOAD LIMIT
- 1 BUS LOAD LIMIT

P PRIORITY BIT

Figure 11-34 Definition of the channel command word (CCW). (Reprinted by permission of Intel Corporation. Copyright 1979.)

A0 line is connected to the SEL pin. A CA consists of placing one of the IOP's two addresses on the address lines; the signals on the data lines are ignored. Issuing a CA can be accomplished by an OUT instruction. Except during initialization, when SEL = 0 causes the IOP to be a master and SEL = 1 causes it to be a slave, the SEL input determines which channel is to receive the attention (SEL = 0 specifies channel 1 and SEL = 1 specifies channel 2).

After an IOP is reset (normally by a power-up sequence), it must be initialized before it can perform the first task. To initialize an IOP, the CPU's program must fill in the SCPB and SCB; set the BUSY bytes for channels 1 and 2 to FF and 00, respectively; issue a CA to specify whether the IOP is to be a master or a slave; and wait until the BUSY byte for channel 1 is cleared by the IOP. A typical assembler language implementation of the initialization process is given in Fig. 11-36. This sequence assumes that the IOP is assigned the I/O port addresses 0080 and 0081 and is to be operated in the master mode. It further assumes that the system bus width is 16 bits and the I/O bus width is 8 bits. If the IOP's port addresses were in the system space instead of the I/O space, then the OUT instruction would need to be replaced with a MOV instruction.

During the initialization phase the CA causes the IOP to automatically carry out the following sequence:

1. Input SYSBUS from location FFFF6 so that the IOP can adjust itself to the system bus.

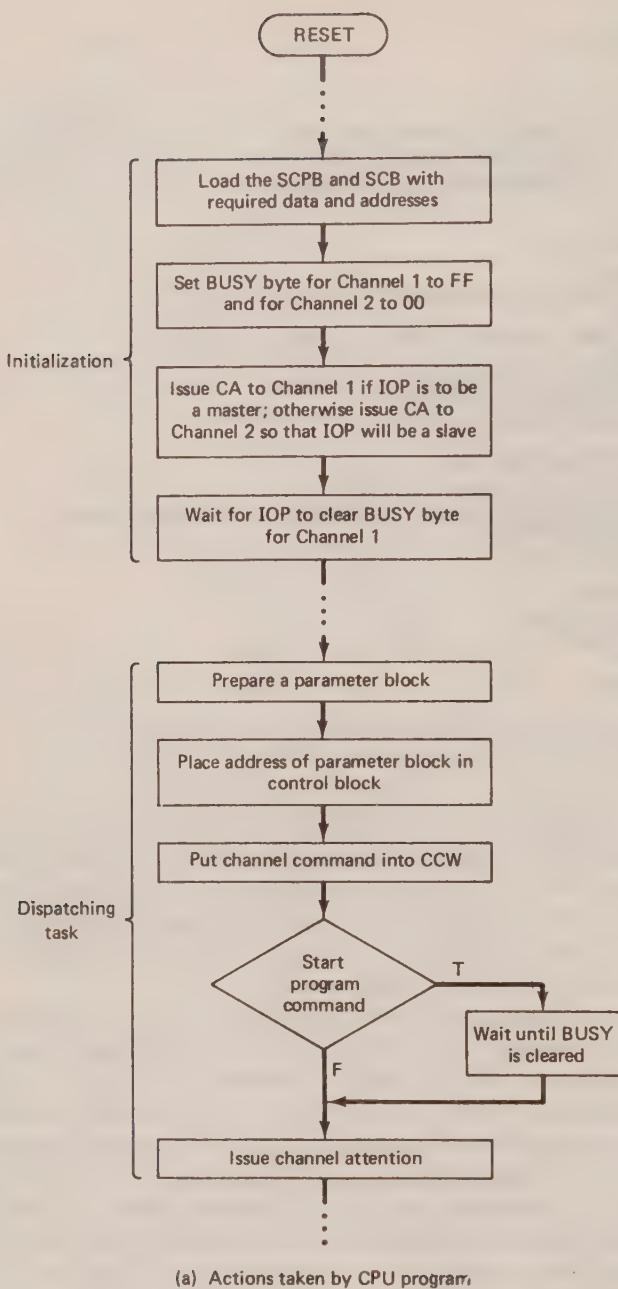
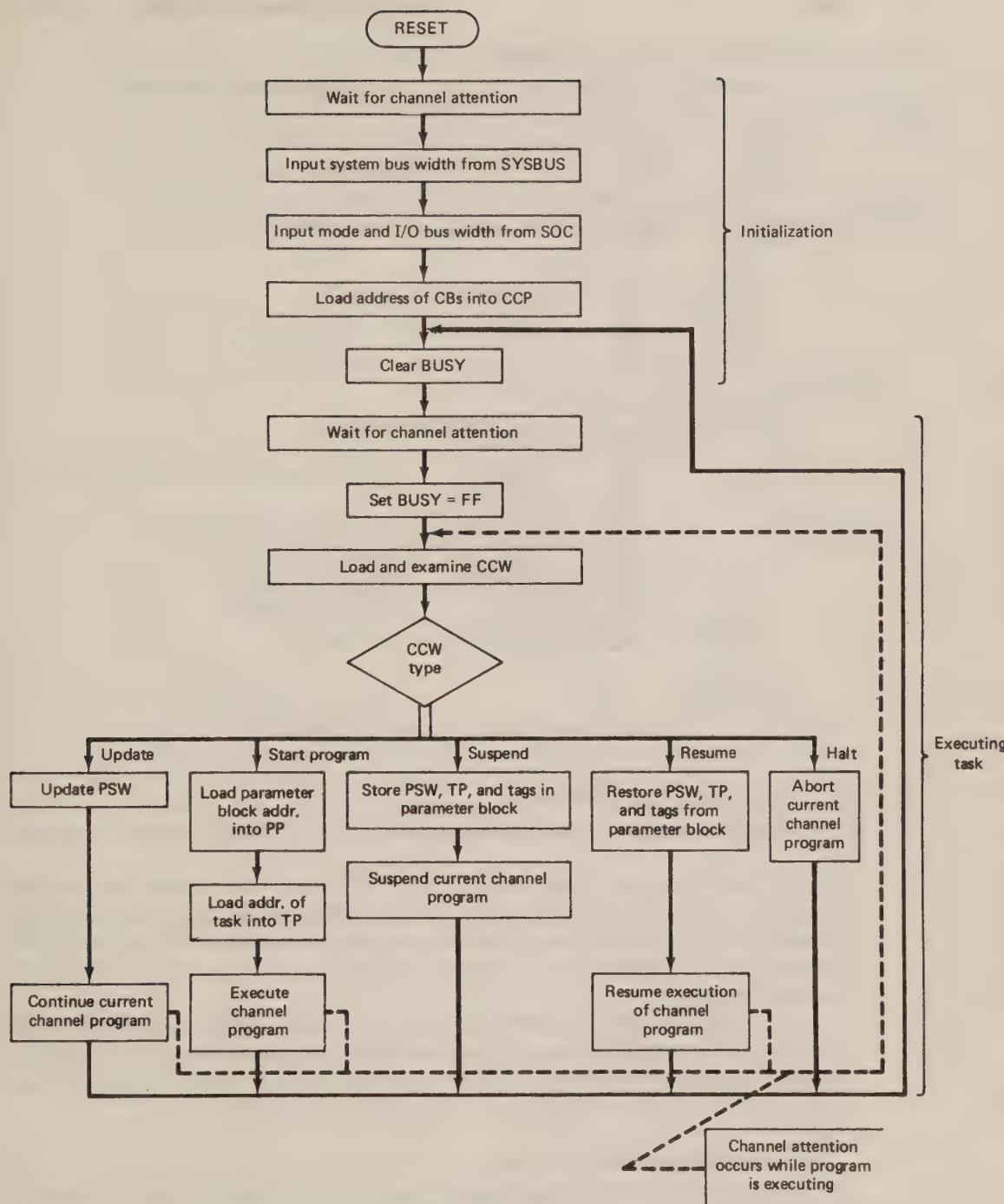


Figure 11-35 Flowcharts of the initialization and command actions for the 8089.

2. Load the address of the SCB from locations FFFF8 and FFFFA.
3. Input SOC from the SCB so that the IOP will be assigned a mode and I/O bus width.
4. Load the address of the control block from SCB + 2 and SCB + 4 into the IOP's CCP register.



(b) Actions taken by IOP

Figure 11-35 Continued.

```

SCPBLOCK SEGMENT AT FFFFH
    ORG 6
    SYSBUS DB 1 ;SYSTEM BUS WIDTH IS 16 BITS
    DB ?
    SCBP DD SOC
SCPBLOCK ENDS
SCBLOCK SEGMENT
    SOC DB ?
    DB ?
    CBP DW ?,?
SCBLOCK ENDS
CCBIOPX SEGMENT
    CCW1 DB ?
    BUSY1 DB ?
    PBP1 DW ?,?
    DW ?
    CCW2 DB ?
    BUSY2 DB ?
    PBP2 DW ?,?
    DW ?
CCBIOPX ENDS
INIT CODE SEGMENT
    ASSUME CS:INIT_CODE, DS:CCBIOPX, ES:SCBLOCK
        MOV AX, SCBLOCK
        MOV ES, AX
        MOV ES, SOC, 0 ;I/O BUS WIDTH IS 8, STANDARD RQ/GT
        MOV AX, CCBIOPX
        MOV DS, AX
        MOV ES, CBP+2, AX ;STORE SEGMENT ADDR OF
        MOV AX, OFFSET CCW1 ;CCW1 TO CBP+2
        MOV ES, CBP, AX ;STORE OFFSET OF CCW1 TO CBP
        MOV BUSY1, OFFH ;SET BUSY1 TO FF
        MOV BUSY2, 0 ;SET BUSY2 TO 00
        OUT 80H, AL ;MAKE IOP MASTER
        TEST BUSY1, OFFH ;LOOP UNTIL
        JNE WAITING ;NOT BUSY
        .
        .
WAITING:    .

```

Figure 11-36 Instruction sequence for initializing an IOP.

5. Set BUSY in CB + 1 to 00.

6. Wait for the next CA signal from the CPU to start the execution of a task.

If the system has more than one IOP, the CPU may, one by one, initialize the other IOPs using the same procedure and the same areas for the SCPB and SCB. Clearly, the contents of the SCB must be updated before initializing each IOP because each IOP must have a separate control block. Also, the modes and I/O bus widths may vary.

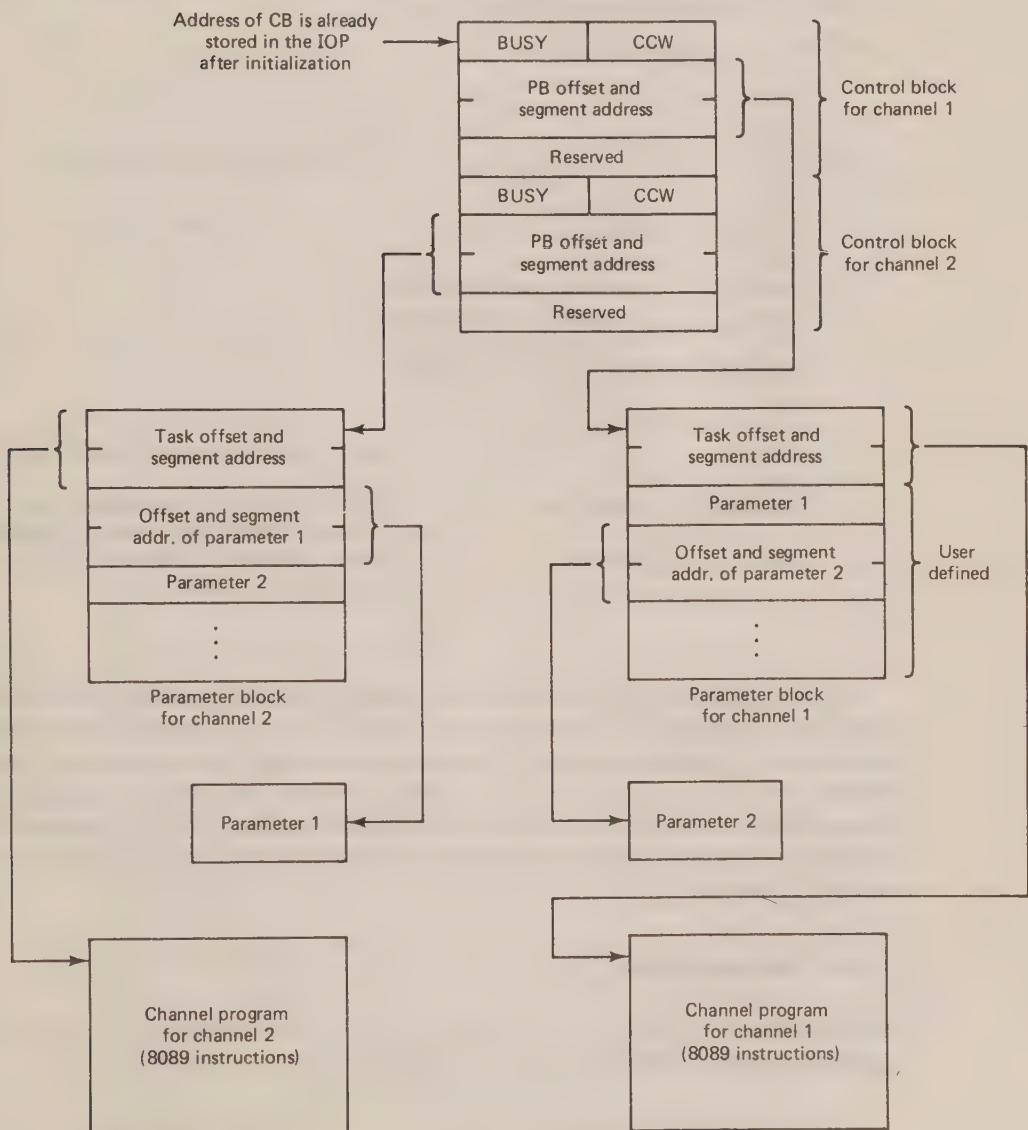
Because the initialized IOP has already saved the beginning address of the control blocks in its CCP register and established the other necessary system configuration information, either of the two channels is now able to start a channel program. The CPU can dispatch a task to either channel by:

1. Preparing a parameter block.
2. Placing the address of parameter block in CB + 2 and CB + 4 for channel 1 or in CB + 10 and CB + 12 for channel 2.
3. Putting the desired channel command into the channel's CCW.
4. Issuing a CA to the channel.

The linkage of the message blocks is shown in Fig. 11-37. Note that in the parameter block, the first two words always point to the channel program and the format of the remaining words is user defined. (Like a parameter table in calling a subroutine, they may contain either parameters or parameter addresses.) The parameter block always resides in the system space.

As an example, consider a channel program to read in a line of text from an 8251A that is connected to an IOP. The entry point of the channel program is

Figure 11-37 Linkage of the message blocks.



INPUTPG and the text string is to be stored at MSG. Let us assume that the control block is defined as in Fig. 11-36 and that the parameter block for channel 1 is:

```
PB1      SEGMENT
    TB1      DW  ?,?
    MSGP     DW  ?,?
    NUM      DW  ?
PB1      ENDS
```

The following instruction sequence would initiate the channel program for channel 1, whose I/O port address is 0080:

```
EXTRN      INPUTPG:FAR
.

ASSUME DS:CCBIOPX, ES:PB1           ; PB1 IS START OF PARM BLOCK 1
MOV       AX,CCBIOPX
MOV       DS,AX
MOV       AX,PB1
MOV       ES,AX
MOV       ES:TB1,OFFSET INPUTPG
MOV       ES:TB1 + 2,SEG INPUTPG
MOV       ES:MSGP,OFFSET MSG
MOV       ES:MSGP + 2,SEG MSG
MOV       PBP1,OFFSET TB1
MOV       PBP1 + 2,AX
TY: TEST    BUSY1,0FFH          ;WAIT UNTIL CHANNEL IS NOT BUSY
JNE       TY
MOV       CCW1,13H           ;START CHANNEL PROGRAM LOCATED IN
                             ;SYSTEM SPACE AND ENABLE INTERRUPT
OUT      80H,AL              ;ISSUE CHANNEL ATTENTION TO
                             ;CHANNEL 1
.

.
```

Once an IOP is initialized, for subsequent channel attentions it loads CCW from CP + 0 or CP + 8 depending on the state of the SEL input. If the CCW is a start channel program command, the addressed channel loads the channel program address from the parameter block into the channel's task pointer (TP) and starts the execution. The CPU can detect the completion of a channel program either through an interrupt (if the interrupt enable bit in the CCW is set) or by testing the BUSY status.

11-4-3 Instruction Set

The 8089 has a total of 53 instructions, which are summarized in Fig. 11-38. The instructions can be divided into the following types:

1. General data transfers.

2. 8- or 16-bit arithmetic operations.
3. 8- or 16-bit logic operations.
4. Pointer load and store instructions.
5. Conditional and unconditional branches and subroutine calls.
6. Bit manipulation and testing operations.
7. Processor control instructions.

Figure 11-38 8089 instruction set.

| Name | Mnemonic | Notes | Operation |
|----------------------------------|----------|------------|---|
| Add memory word | ADD | DST, SRC | (1) $(DST) \leftarrow (DST) + (SRC)$ |
| Add memory byte | ADDB | DST, SRC | (2) $(DST) \leftarrow (DST) + (SRC)$ |
| Add immediate byte | ADDBI | DST, I8 | (6) $(DST) \leftarrow (DST) + I8$ |
| Add immediate word | ADDI | DST, I16 | (5) $(DST) \leftarrow (DST) + I16$ |
| And memory word | AND | DST, SRC | (1) $(DST) \leftarrow (DST) \wedge (SRC)$ |
| And memory byte | ANDB | DST, SRC | (2) $(DST) \leftarrow (DST) \wedge (SRC)$ |
| And immediate byte | ANDBI | DST, I8 | (6) $(DST) \leftarrow (DST) \wedge I8$ |
| And immediate word | ANDI | DST, I16 | (5) $(DST) \leftarrow (DST) \wedge I16$ |
| Call | CALL | M, DST | (7) $(M) \leftarrow (TP) + \text{tag bit}$ $(TP) \leftarrow (TP) + [DST - (TP)]$ |
| Clear selected bit to zero | CLR | M, I3 | $(M)_{I3} \leftarrow 0$ (Clear bit I3 in M) |
| Decrement word by 1 | DEC | DST | (5) $(DST) \leftarrow (DST) - 1$ |
| Decrement memory byte by 1 | DECB | M | $(M) \leftarrow (M) - 1$ |
| Halt channel program | HLT | | (BUSY in CB) $\leftarrow 0$ and stop channel program |
| Increment word by 1 | INC | DST | (5) $(DST) \leftarrow (DST) + 1$ |
| Increment memory byte by 1 | INCB | M | $(M) \leftarrow (M) + 1$ |
| Jump if selected bit is true | JBT | M, I3, DST | (7) If $(M)_{I3} = 1$, $(TP) \leftarrow (TP) + [DST - (TP)]$ |
| Jump if masked compare is equal | JMCE | M, DST | (7) If $((M) \oplus (MC)_{7:0}) \wedge (MC)_{15:8} = 0$, $(TP) \leftarrow (TP) + [DST - (TP)]$ |
| Jump if masked compare not equal | JMCNE | M, DST | (7) If $((M) \oplus (MC)_{7:0}) \wedge (MC)_{15:8} \neq 0$, $(TP) \leftarrow (TP) + [DST - (TP)]$ |
| Jump unconditionally | JMP | DST | (7) $(TP) \leftarrow (TP) + [DST - (TP)]$ |
| Jump if selected bit not true | JNBT | M, I3, DST | (7) If $(M)_{I3} = 0$, $(TP) \leftarrow (TP) + [DST - (TP)]$ |

| Name | Mnemonic | Notes | Operation |
|--|----------|-------------------------|---|
| Jump if word is not zero | JNZ | SRC,DST (7), (10) | If (SRC) ≠ 0, (TP) ← (TP) + [DST-(TP)] |
| Jump if memory byte not zero | JNZB | M,DST (7) | If (M) ≠ 0, (TP) ← (TP) + [DST-(TP)] |
| Jump if word is zero | JZ | SRC,DST (7), (10) | If (SRC)=0, (TP) ← (TP) + [DST-(TP)] |
| Jump if memory byte is zero | JZB | M,DST (7) | If (M)=0, (TP) ← (TP) + [DST-(TP)] |
| Long call | LCALL | M,DST (8) | (M) ← (TP) + tag bit (TP) ← (TP) + [DST-(TP)] |
| Long jump if selected bit true | LJBT | M,I3,DST (8) | If (M) ₁₅ =1, (TP) ← (TP) + [DST-(TP)] |
| Long jump if masked compare equal | LJMCE | M,DST (8) | If ((M) ⊕ (MC) ₁₅₋₀) ∧ (MC) ₁₅₋₈ =0, (TP) ← (TP) + [DST-(TP)] |
| Long jump if masked compare is not equal | LJMCNE | M,DST (8) | If ((M) ⊕ (MC) ₁₅₋₀) ∧ (MC) ₁₅₋₈ ≠0, (TP) ← (TP) + [DST-(TP)] |
| Long jump unconditionally | LJMP | DST (8) | (TP) ← (TP) + [DST-(TP)] |
| Long jump if selected bit is not true | LJNBT | M,I3,DST (8) | If (M) ₁₅ =0, (TP) ← (TP) + [DST-(TP)] |
| Long jump if word not zero | LJNZ | SRC,DST (8), (10) | If (SRC) ≠ 0, (TP) ← (TP) + [DST-(TP)] |
| Jump if memory byte not zero | LJNZB | M,DST (8) | If (M) ≠ 0, (TP) ← (TP) + [DST-(TP)] |
| Long jump if word is zero | LJZ | SRC,DST (8), (10) | If (SRC)=0, (TP) ← (TP) + [DST-(TP)] |
| Long jump if memory byte zero | LJZB | M,DST (8) | If (M)=0, (TP) ← (TP) + [DST-(TP)] |
| Load pointer from memory | LPD | P,M | (P) ← 20-bit address formed by offset and segment address stored beginning at M. Tag bit ← 0 |
| Load pointer from immediate data | LPDI | P,SRC (11) | (P) ← 20-bit address formed by offset of SRC and corresponding segment address Tag bit ← 0 |
| Move word | MOV | DST,SRC (3) | (DST) ← (SRC) |
| Move byte | MOVB | DST,SRC (4) | (DST) ← (SRC) |
| Move immediate byte | MOVBI | DST,I8 (6) | (DST) ← I8 |
| Move immediate word | MOVI | DST,I16 (5) | (DST) ← I16 |
| Move pointer | MOVP | DST,SRC (12) | (DST) ← (SRC) (Including 20-bit address and tag bit. Memory operand has three bytes.) |
| No operation | NOP | | No operation |

Figure 11-38 Continued.

| Name | Mnemonic | Notes | Operation |
|----------------------------------|--------------------------|-------|--|
| Not word | NOT DST or NOT R,M | (5) | $(DST) \leftarrow \overline{(DST)}$ or $(R) \leftarrow \overline{(M)}$ |
| Not byte | NOTB M or NOTB R,M | | $(M) \leftarrow \overline{(M)}$ or $(R) \leftarrow \text{sign-extended } \overline{(M)}$ |
| Or word | OR DST,SRC | (1) | $(DST) \leftarrow (DST) \vee (SRC)$ |
| Or byte | ORB DST,SRC | (2) | $(DST) \leftarrow (DST) \vee (SRC)$ |
| Or immediate byte | ORBI DST,I8 | (6) | $(DST) \leftarrow (DST) \vee I8$ |
| Or immediate word | ORI DST,I16 | (5) | $(DST) \leftarrow (DST) \vee I16$ |
| Set selected bit to 1 | SETB M,I3 | | $(M) \leftarrow 1$ (Set bit I3 in M.) |
| Set interrupt request bit | SINTR | | Activate SINTR pin if (CCW) _{4,3} = 10 |
| Test and set while locked | TSL M,I8,DST | (9) | If $(M)=0$, $(M) \leftarrow I8$, otherwise $(TP) \leftarrow (TP) + [DST - (TP)]$. Activate the LOCK pin during instruction execution. |
| Set logical bus width | WID W1,W2 | | Set DMA source data width to W1 (W1: 8 or 16). Set DMA destination data width to W2 (W2: 8 or 16) |
| Start DMA after next instruction | XFER | | DMA transfer begins following the execution of the next instruction. |

Notes on operand forms:

- (1) DST,SRC R,M(word) or M(word),R
- (2) DST,SRC R,M(byte) or M(byte),R
- (3) DST,SRC R,M(word) or M(word),R or M(word),M(word)
- (4) DST,SRC R,M(byte) or M(byte),R or M(byte),M(byte)
- (5) DST R or M(word)
- (6) DST R or M(byte)
- (7) DST Label--Distance [DST-(TP)] may be 8 or 16 bits long.
- (8) DST Label--Distance [DST-(TP)] is 16 bits long.
- (9) DST Label--Distance [DST-(TP)] must be 8 bits long.
- (10) SRC R or M(word)
- (11) SRC 16-bit constant or memory address
- (12) DST,SRC P,M (3 bytes) or M,P

Legend

R: GA,GB,GC,BC,IX,MC,CC, or TP

P: GA,GB,GC, or TP

M: Memory operand

In: Integer that can be represented by n bits

Figure 11-38 Continued.

For the arithmetic and logic instructions, an operand may be a pointer (20 bits), a register (16 bits), a memory operand (byte or word), or a constant (8 bits or 16 bits), and the source and destination need not be of the same length. During an arithmetic operation, the byte and word operands are sign-extended to 20 bits. If the destination is not a pointer, the high-order 4 bits of the results are truncated before being stored in the destination. During a logic operation, a byte operand is sign-extended to 16 bits if necessary. When the results of a logic operation are stored in a pointer, the upper 4 bits are undefined.

A move instruction, MOV, MOVB, MOVI or MOVBI, also extends the sign bit or truncates the high-order bits of the operand to fit the size of the destination. If the destination is a pointer, then its tag bit is set to 1. This provides an efficient way of loading an address in the I/O space into a pointer.

A memory operand must be addressed indirectly through a pointer (GA, GB, GC or PP). As indicated in the preceding section, the PP register is automatically loaded with the address of a channel's parameter block when a task is dispatched to the channel. This register is used primarily to access the parameters or their addresses. The GA, GB, and GC registers can be loaded using the MOVP and pointer load instructions.

The four data-related memory addressing modes are:

1. **Based Addressing Mode**—The effective address is the contents of the specified pointer.
2. **Offset Addressing Mode**—The effective address is the sum of the contents of the specified pointer plus an 8-bit unsigned offset. The offset may also be specified as a symbol which represents an element in a structure.
3. **Indexed Addressing Mode**—The effective address is the sum of the contents of the IX register, which are interpreted as an unsigned integer, plus the contents of the specified pointer.
4. **Auto-increment Indexed**—The effective address is formed in the same way as in the indexed address mode. However, after the effective address is calculated, IX is updated according to the operation. It is incremented by 1 for a byte operation, by 2 for a word operation, and by 3 for a move pointer instruction.

The resulting effective address contains 20 bits. If the tag bit associated with the specified pointer is 0, all 20 bits are used to access an operand in the system space. If the tag bit is 1, only the low-order 16 bits are used to access an operand in the I/O space. The format of the addressing modes in the ASM-89 assembler language along with some coding examples are given in Fig. 11-39.

All branch instructions, including subroutine calls, use the relative addressing mode. The ASM-89 assembler calculates the distance between the destination and the address of the next instruction in sequence [i.e., the address of the label associated with the point being branched to minus the (TP)] and inserts it into the branch instruction. If a jump is performed, this offset is added to (TP).

| Addressing Mode | Notation | Example |
|------------------------|------------------|----------|
| Based | [pointer] | [GA] |
| Offset | [pointer].offset | [GB].5 |
| Indexed | [pointer+IX] | [PP+IX] |
| Auto-increment indexed | [pointer+IX+] | [GA+IX+] |

Figure 11-39 Memory operand addressing modes.

11-4-4 Examples

To illustrate how to program the IOP, two examples are given in this section. The programs were kept basic and tutorial to avoid unnecessary details which are dependent on the actual system. The first example reads a line of a message from a terminal connected to an 8251A interface, stores the message in a buffer area, and returns the number of characters read. If a rubout (delete) character is input, the previous character is deleted. Each character that is input is echoed back to the terminal. It is assumed that the data register address and control/status register address of the 8251A are assigned 1000 and 1001, respectively, and that the 8251A has been initialized for the proper data format, baud rate, and I/O mode. The channel program written in the assembler language ASM-89 is given in Fig. 11-40. This program can be linked to an 8086/8088 program module that dispatches the task to the 8089 such as the example shown in the preceding section.

This example shows how programmed I/O can be performed and how a character string is scanned by the masked compare operation. After a carriage return is detected, the channel program returns the number of received characters

```

IOPROG1      SEGMENT
PUBLIC      INPUTPG
INPUTPG:    LPD      GA,[PP].4      ;LOAD BUFFER ADDR FROM PARAMETER TABLE
            MOVI     GB,1000H   ;PUT DATA REGISTER ADDR OF 8251A IN GB
            MOVI     GC,1001H   ;PUT STATUS REG. ADDR OF 8251A IN GC
            MOV     IX,0       ;SET INDEX REGISTER TO 0
            JNBT    [GC],1,INPWAIT ;IS AN INPUT CHARACTER READY?
                           ;IF NOT WAIT
            MOVB    BC,[GB]    ;OTHERWISE, MOVE IT TO BC
            MOVI    MC,7F7FH   ;COMPARE WITH RUBOUT
            JMCNE  [GB],STORE ;IF NOT A RUBOUT, GO TO STORE
            DEC     IX        ;POINT TO PREVIOUS CHARACTER IN BUFFER
            MOVBI   BC,08H    ;LOAD BC WITH BACKSPACE
INPWAIT:    JNBT    [GC],0,OUTWAIT ;IF OUTPUT DATA NONEMPTY, WAIT
            MOVB    [GB],BC    ;OUTPUT ASCII CODE OF BACKSPACE
            JMP     INPWAIT   ;INPUT THE NEXT CHARACTER
OUTWAIT:    JNBT    [GC],0,OUTWAIT ;IF OUTPUT DATA NONEMPTY, WAIT
            MOVB    [GA+IX],BC ;STORE CHARACTER IN BUFFER AREA
            ECHO:   JNBT    [GC],0,ECHO   ;IS OUTPUT DATA REGISTER EMPTY?
            MOVB    [GB],BC    ;IF TRUE, ECHO CHARACTER
            MOVI    MC,7F0DH   ;COMPARE WITH CR
            JMCNE  [GA+IX+],INPWAIT ;IS IT CR? IF NO, INPUT
                           ;NEXT CHARACTER AND INCREMENT IX
            MOV     [PP].8,IX    ;RETURN LENGTH OF CHARACTER
                           ;STRING TO PARAMETER BLOCK
            SINTR   HLT       ;INTERRUPT THE 8086/8088
                           ;WAIT FOR NEXT CA TO START
                           ;ANOTHER CHANNEL PROGRAM
IOPROG1      ENDS

```

Figure 11-40 Sample channel program to input a line of characters

to the parameter block, notifies the CPU with an interrupt request, clears the busy status, and waits for the next channel attention.

The second example writes a given number of bytes to a floppy disk controlled by an 8272 disk controller. The parameter block is assumed to be set up as follows:

Channel program pointer (4 bytes)
Buffer pointer (4 bytes)
Number of bytes to be written (2 bytes)
Drive unit (1 byte)
Cylinder number (1 byte)
Head address (1 byte)
Record (1 byte)
Number of bytes per sector (1 byte)
End of track (1 byte)
Gap length (1 byte)
Data length (1 byte)

The next 3 bytes are for storing the status bytes ST0, ST1, and ST2 after the write has been completed. Also assume that the main status register, data register, and terminal count (see Exer. 15) of the 8272 are assigned the addresses 8000, 8001, and 8002. As shown in Fig. 11-41, prior to the XFER instruction the program sets up source and destination pointers, specifies the contents of the channel control register, specifies the bus widths, and loads all of the parameters except the last one into the 8272. The XFER instruction causes the channel to start its DMA operation after the succeeding instruction has been executed. This will allow the IOP enough time to enter the DMA mode even though the device may begin transferring data immediately upon receipt of the last parameter.

The channel control register is loaded with 5008H to indicate that the DMA operation is to be performed from memory to I/O, is not to include translation, is to be synchronized by the destination, and is to use GA as the source pointer. Upon loading the last parameter, the DMA is to begin. When the DMA stops because the byte count has expired, the terminal count signal is sent, the status bytes ST0, ST1, and ST2 are returned to the parameter block, and the 8272 parameters C, H, R and N are updated.

BIBLIOGRAPHY

1. *The 8086 Family User's Manual, Numerics Supplement* (Santa Clara, Calif.: Intel Corporation, 1980).
2. *8086/8087/8088 Macro Assembly Language Reference Manual* (Santa Clara, Calif.: Intel Corporation, 1980).
3. *The 8086 Family User's Manual* (Santa Clara, Calif.: Intel Corporation, 1979).
4. *8089 Assembler User's Guide* (Santa Clara, Calif.: Intel Corporation, 1979).

```

DISKWT      SEGMENT
PUBLIC WRITE
    WRITE:      MOVI    GC,8000H      ;GC POINTS TO STATUS REGISTER OF 8272
                MOVI    MC,OC080H      ;SET MASK. COMPARE BYTE TO 10XXXXXX
                LPD     GA,[PP].4      ;GA POINTS TO SOURCE
                MOVI    GB,8001H      ;GB POINTS TO DESTINATION
                MOVI    CC,5008H      ;SET CC FOR PROPER DMA OPERATION
                WID     16,8          ;SET SOURCE BUS=16, DESTINATION BUS=8
TRY1:       JMCNE   [GC],TRY1      ;IS 8272 READY TO ACCEPT NEXT COMMAND
                MOVBI   [GC].1,5      ;LOAD WRITE COMMAND
                MOVI    IX,10         ;SET UP INDEX REGISTER TO LOAD
                MOVI    BC,7          ;NEXT 7 PARAMETERS INTO 8272
TRY2:       JMCNE   [GC],TRY2      ;IS 8272 READY FOR NEXT PARAMETER
                MOVB   [GC].1,[PP+IX+]  ;LOAD NEXT PARAMETER
                DEC    BC             ;REPEAT 7 TIMES
                JNZ    BC,TRY2        ;LOAD NO. OF BYTES TO BYTE COUNTER
                MOV    BC,[PP].8      ;START DMA AFTER NEXT INSTRUCTION
TRY3:       JMCNE   [GC],TRY3      ;LOAD LAST PARAMETER INTO 8272
                XFER   MOV,[GC].1,[PP].17 ;AND START DMA TRANSFER
                MOV    [GC].1,[PP].17  ;IF DONE, SEND TERMINAL COUNT
                MOVBI  [GC].2,0        ;SET MASK COMPARE BYTE TO 11XXXXXX
                MOVI   BC,3           ;STORE ST0, ST1, AND ST2
                MOVI   IX,18          ;DEC
                MOVI   MC,OCOCOH      ;JNZ
                JMCNE   [GC],TRY4      ;BC,TRY4
                MOVB   [PP+IX+],[GC].1 ;MOV
                DEC    BC             ;BC,4
                JNZ    BC,TRY4        ;IX,11
TRY5:       JMCNE   [GC],TRY5      ;UPDATE PARAMETERS C,H,R, AND N
                MOVB   [PP+IX+],[GC].1 ;DEC
                DEC    BC             ;JNZ
                JNZ    BC,TRY5        ;SINTR
                SINTR  HLT           ;INTERRUPT THE CPU
                HLT              ;WAIT FOR NEXT CHANNEL ATTN
WRITE      ENDS

```

Figure 11-41 Sample channel program for writing a block of bytes to a disk.

EXERCISES

1. Construct a table to summarize the architectural features of the 8086/8088 that support multiprocessor design. The first column should list all the special pins and instructions, and the second column should list the primary purpose of that pin or instruction as related to multiprocessing applications.
2. Assume that a loosely coupled multiprocessor system consists of the following three modules:

Module A—An 8086 with a local memory.

Module B—An 8086 and an 8087.

Module C—Two 8089s with a local I/O bus.

Determine the major bus interface devices required for each module.

3. Modify Fig. 11-14 so that the 8086 could have access to two system buses.
4. Explain why the processor utilization rate can be improved in a multiprocessor system by an instruction queue.
5. Suppose that the following single operand instructions are to be executed:
 1. 500 general instructions; each instruction requires two cycles for execution plus two cycles each for the instruction fetch and operand reference.

2. 300 floating point instructions; each instruction requires 15 cycles for execution plus four cycles each for the instruction fetch and operand reference.
 3. 200 I/O related instructions; each instruction requires one cycle for execution plus two cycles each for the instruction fetch and operand reference.
- Assuming a bus bandwidth of 10^6 cycles per second, for each of the following system configurations determine the amount of time required to execute all of these instructions and the bus utilization rate:
- (a) The system is implemented by a single processor.
 - (b) The system consists of two processors, one to execute general and I/O-related instructions, and the second one to execute floating point instructions under an ideal concurrent processing situation.
 - (c) The system consists of three processors, one for each instruction category. Assume an ideal concurrent processing situation.
6. Assume that the following hexadecimal numbers represent short real numbers:
 (a) CA5B2000 (b) 3C630000 (c) 4C341200
 Determine their decimal equivalents.
 7. Convert the following decimal numbers to short real numbers (write your answers in hexadecimal):
 (a) -123450000 (b) -148.4375×10^{-3} (c) 1.2345625×10^4
 8. Write an instruction sequence in 8086/8087 assembler language that will convert an unsigned decimal number to its equivalent short real number. Assume that the integral part, which is up to 10 decimal digits, is stored in array INTARY, the fractional part, which is up to 10 decimal digits, is stored in array FRARY, and the result is to be stored in REALVAR. Further assume that a 10-byte array labeled WORK has been reserved as a working area.
 9. Assume that N real numbers are stored in the short real format starting at location REALARY. Write an instruction sequence in 8086/8087 assembler language that would multiply the sum of all positive numbers by the sum of all negative numbers and then store the result in PRODUCT.
 10. Write an instruction sequence in 8086/8087 assembler language that would calculate $\sin \theta$ and $\tan \theta$, where θ is greater than 0 and less than $\pi/2$. Assume that THETA contains the angle in the short real format and that the results are to be stored in the long real format.
 11. Discuss how to solve the critical section problem when using an 8087 in a multiprocessor system.
 12. For the examples of IOP initialization and task dispatching given in Sec. 11-4-2, show how the SEL and CA pins should be connected to the system bus.
 13. Write a channel program to reimplement the priority polling given in Fig. 6-7.
 14. Assume that the keyboard/display unit given in Fig. 9-35 is controlled by the 8089. Write two channel programs to replace the keyboard input routine and the display routine discussed in Sec. 9-4-3. The addresses of KEYS and DIGITS should be passed to the channel programs as parameters.
 15. For the programming example given in Fig. 11-41, show the required logic to be connected to the 8272 so that a termination signal will be received by the disk controller upon the completion of a DMA operation.
 16. Discuss how to solve the critical section problem when using an 8089 in a multiprocessor system.

VLSI Processing and Supporting Devices

When a manufacturer decides to develop new VLSI devices or is able to put more logic on a single chip, it must decide on the most advantageous way to utilize the added logic. Some of the more prominent directions that could be followed are to:

1. Increase the number of working registers and/or extend their width. If the width of the registers is increased, the number of data and/or address pins may also be made larger so that the data transfer rate and memory capacity are increased. Changes of this nature generally require considerably more logic. They are the most drastic, and they normally introduce a new family of processors and associated devices.
2. Improve the computational capability of the processor. This is normally done by designing a separate high-speed numerical processing device that is compatible with the manufacturer's CPU. A specially designed numeric processor that can perform multiple-precision and floating point operations significantly increases a system's throughput for computationally bound applications.
3. Enhance the ability to access high-speed mass storage. This could be done by placing a DMA controller on the CPU chip, but for high-performance I/O applications a separate device is needed which can perform I/O-related processing as well as execute DMA transfers.
4. Include special-purpose or system software in ROM that is incorporated onto the CPU chip or on a quickly accessible separate chip that may also contain special system hardware components.
5. Put a set of components such as I/O interfaces, timing elements, DMA con-

trollers, interrupt managers, and sections of memory on the CPU chip so that the overall chip and pin counts of a system can be reduced. (The number of pins on the CPU chip, however, would increase because of the extra lines that connect directly to the various interfaces and devices.) If this approach is used, much of a small system, except for I/O devices, could be reduced to a single chip, thus decreasing the cost of producing the system and increasing its reliability. This approach is most often directed toward controller applications where mass storage and I/O are minimal.

6. Put memory management logic on the same chip as the CPU or develop a separate device that has an extensive memory management capability. The former approach has the advantages of eliminating the delays caused by passing through a separate memory management device and of reducing the overall pin count (even though the pin count on the CPU chip would need to be increased), but is usually more limited. This type of enhancement is aimed at multiprogramming applications.

With regard to item 1 several manufacturers have, in one form or another, developed 32-bit single-chip processing capability. Intel's endeavors in this area are represented by their 80386 processor. Intel's chips for satisfying items 2 and 3 are the 8087 and 8089, which were discussed in Chap. 11. The purpose of this chapter is to describe briefly Intel's devices for fulfilling the applications indicated in items 4 and 5, and item 6, which is fulfilled by the 80286, will be discussed in Chap. 13. It should be pointed out that other manufacturers have made efforts in these areas, but space does not permit an examination of their products. The Intel devices are presented as representative examples.

12-1 THE 80130

The 80130 is a device that is specifically designed to support the iRMX 86 operating system introduced in Sec. 7-1. It is designed to be used with a maximum mode 8086/8088 and together with a CPU it forms what Intel refers to as an operating system processor (OSP). The internal architecture of the 80130 is shown in Fig. 12-1. Its primary component is a 16K-byte ROM that is implanted with the code needed for 35 of the iRMX 86 system calls. The system calls are accomplished through software interrupts and the interrupt pointers in low memory would need to be initialized to point to the appropriate places in this ROM. In addition to the ROM, the 80130 contains interrupt management logic that is very similar to that of the 8259A and can handle up to seven external interrupts; timing circuits for providing the time periods specified in SLEEP and other commands and for regulating a programmable baud rate generator; and all the necessary bus buffering and control logic. The interrupt logic can be cascaded with slave 8259As to form a system capable of handling up to 56 interrupts.

Figure 12-2 gives a pin diagram for the 80130 and Fig. 12-3 shows how it is connected into a typical system. Notice that the 80130 is placed on the CPU side

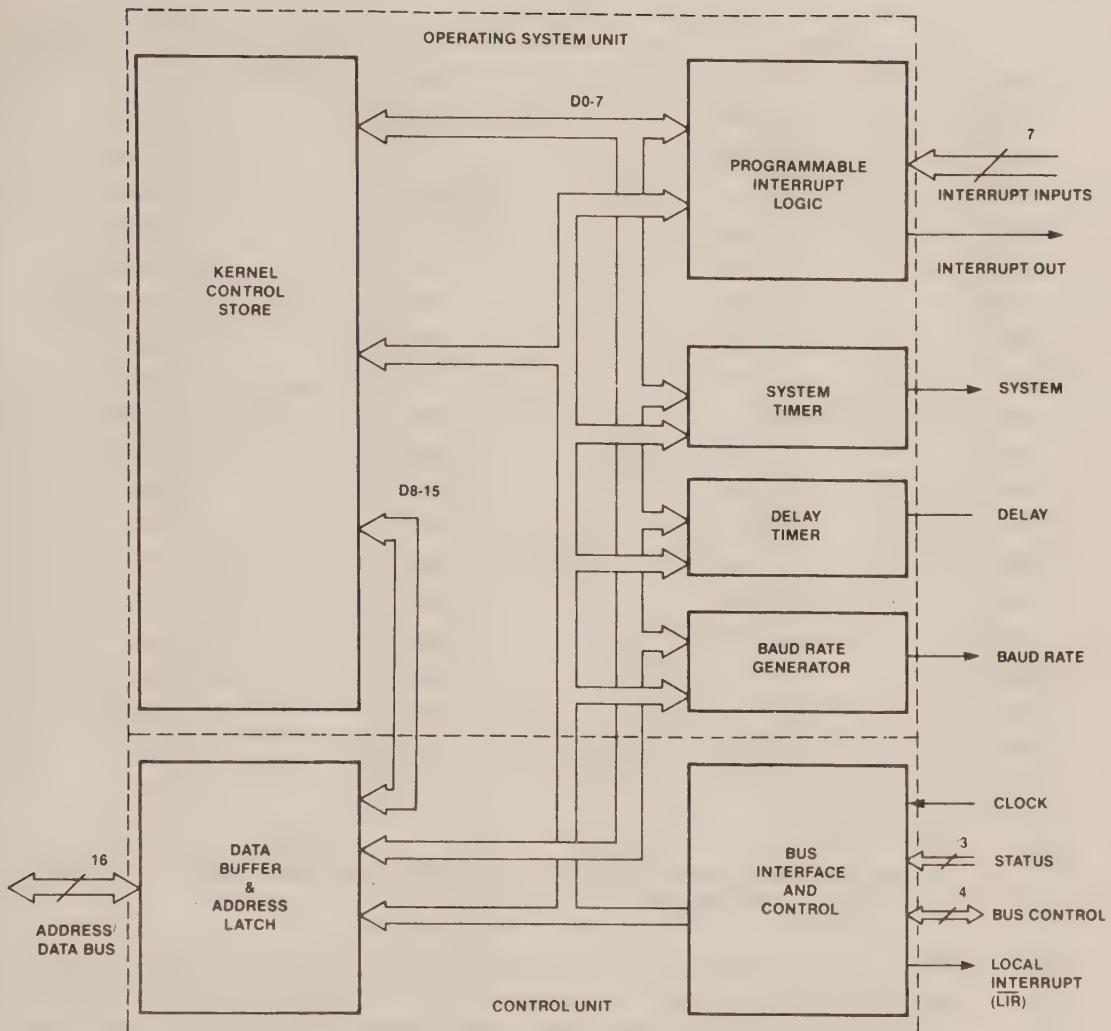


Figure 12-1 Internal architecture of the 80130. (Reprinted by permission of Intel Corporation. Copyright 1982.)

of the bus control logic and must, therefore, be able to decode the status signals S_2-S_0 . It outputs three timing signals: the system timing signal SYSTICK, a delay timing signal (that is not currently used), and the output of the baud rate generator BAUD. Although there are eight interrupt request pins, one of them must be used for system timing and be connected to SYSTICK, leaving only seven of them available to handle interrupt requests. The LIR output is for programmed local slave interrupt control. An 80130 would signal the 8086/8088 that an interrupt has occurred through its INT pin and detect an acknowledge when all of the status pins S_2-S_0 are 0.

To access the 80130's memory and I/O registers, the address bus (including

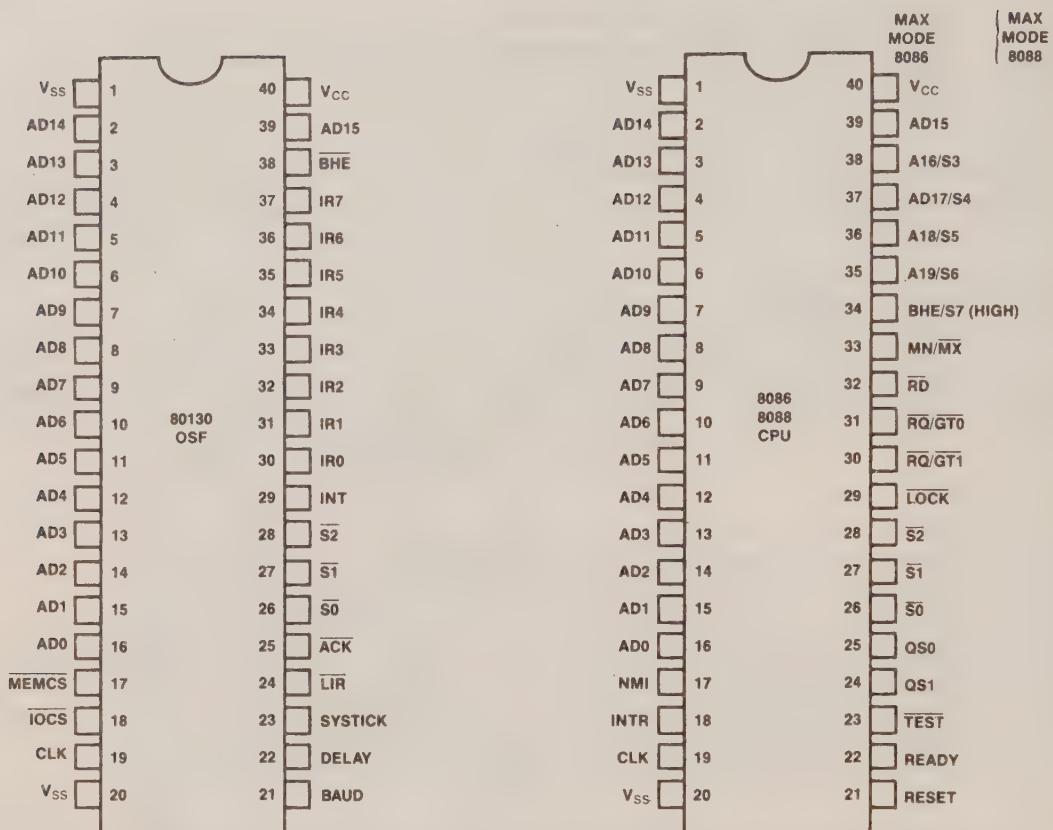


Figure 12-2 Pin diagrams for the 80130 and 8086/8088. (Reprinted by permission of Intel Corporation. Copyright 1982.)

BHE) is input to address decode logic. Since the registers associated with the interrupt management and timing logic are in the I/O address space and the ROM is in the memory address space, the decode logic outputs two chip select signals, which are input through the IOCS and MEMCS pins. The ACK signal is low when an 80130 resource is being accessed and is ANDed with the DEN line to control the 8286 data transceivers. In order to transfer data directly to or from the 80130's I/O registers and from its ROM, the AD15–AD0 lines from the 8086 (or AD7–AD0 lines from an 8088) are connected directly to the AD15–AD0 (AD7–AD0) pins of the 80130.

Although an 8087 is not shown in Fig. 12-3, one could easily be incorporated into the system using the designs shown in Chap. 11. In fact, an existing maximum mode 8086/8088-based system can be upgraded to include an 8087. Also, the designs in Chap. 11 that include 8089s could be modified to include an 80130.

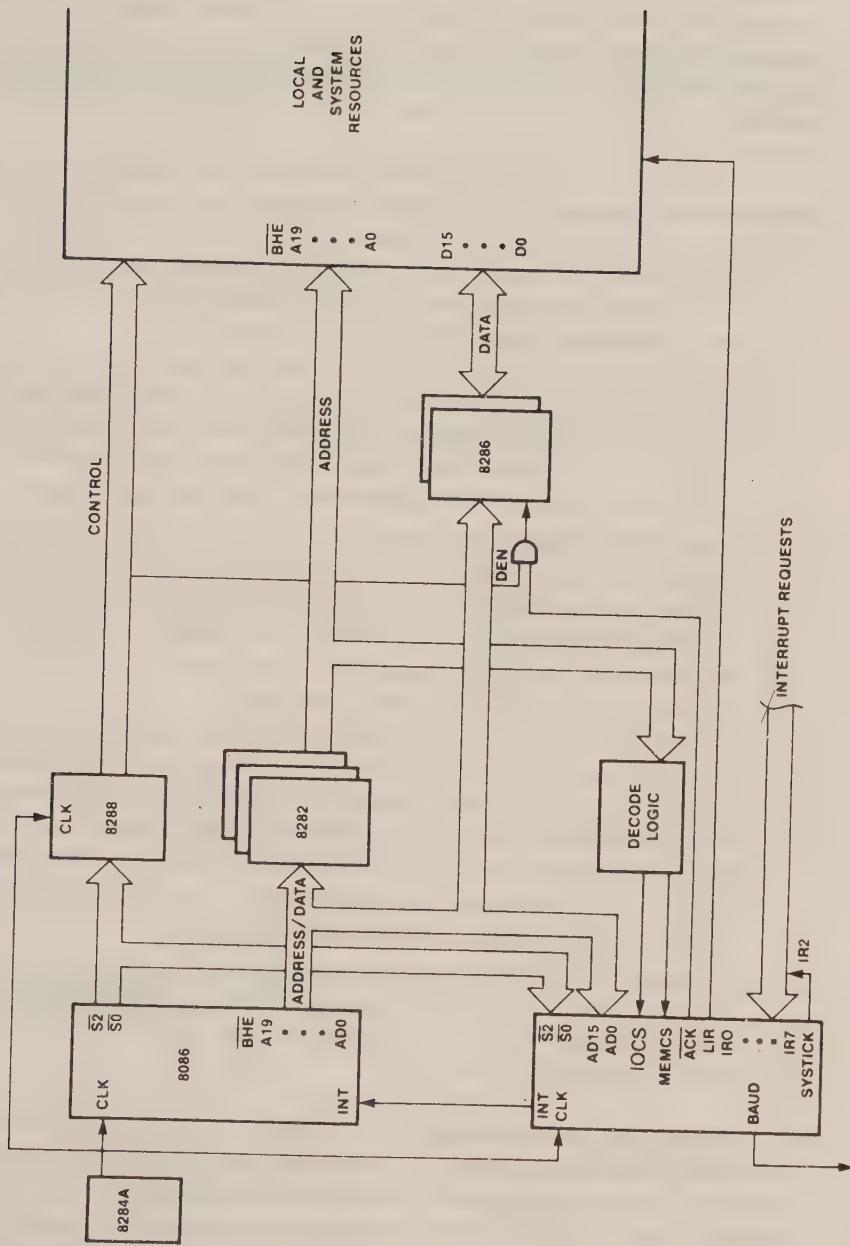


Figure 12-3 Typical OSP system. (Reprinted by permission of Intel Corporation. Copyright 1982.)

12-2 THE 80186

The 80186 is an enhanced version of the 8086 processor, which includes integrated peripheral devices and has a faster instruction execution time. It also has an 8-bit version, the 80188, which has the same architecture except that it uses an 8-bit external data bus. The 80186 family maintains upward object code compatibility with the 8086/8088. The 80186 instruction set includes seven new instructions and extends the PUSH, IMUL, and shift/rotate instructions to include immediate operands.

In addition to the expanded instruction set, the execution of certain instructions has been improved in the 80186. The major execution improvements are:

1. For the 80186, when certain unused opcodes such as 63H, 64H, and so on are executed, an interrupt with type 6 (invalid instruction exception) will occur. The 8086 simply will ignore such opcodes.
2. When the 80186 executes a rotate or shift instruction, the number of bits to shift is the count specified in the instruction modulo 32. This limits the number of shifts to 31. Without this change, for the worst case a shift/rotate instruction could require the 8086 more than 1,000 clock cycles to complete and this could cause a long delay in responding to an interrupt request.
3. If the 80186 is interrupted during the execution of a repeated string primitive with additional prefixes, the return address will point to the first prefix of the repeated string primitive. Therefore, the interrupted string instruction can resume properly upon return from the interrupt.
4. For the 80186, the LOCK signal will not be activated until the locked instruction begins its operand reference bus cycles.
5. An ESC instruction can be executed either in the same way as on the 8086 or it can cause a type 7 interrupt. This option is enabled by setting the most significant bit of the 80186 relocation register (see Sec. 12-2-2) to 1 and is designed to facilitate emulation of floating point instructions. The interrupt's return address will point to the ESC instruction.
6. For the IDIV instruction, the 80186 extends the range of negative numbers allowed as a quotient to include 8000H and 80H.
7. When a word is written into a location with offset = FFFF, the 80186 writes the second byte into the location with offset = 10000 instead of 0000. Similarly, if a push with SP = 1 is executed, the 80186 writes the word into two consecutive physical locations.

Due to the hardware improvements and the fact that effective addresses are calculated by hardware instead of microcode, for most instructions the execution time in clock cycles is smaller than that of the 8086. For example, an IMUL instruction with a memory word operand and a memory to register ADD instruction require 134 to 160 plus EA and 16 + EA clock cycles, respectively, for the 8086.

On the other hand, the same instructions can be executed by the 80186 in 40 to 43 and 10 clock cycles, respectively.

12-2-1 Instruction Set

The 80186 extends the addressing modes of the PUSH, IMUL, and all shift/rotate instructions to include immediate operands. The format of the push immediate instruction is

PUSH DATA

which pushes a word with value specified by the immediate operand onto the stack. This instruction helps pass parameter values or addresses to a procedure through the stack without first moving them into a register. For example,

PUSH 5
PUSH OFFSET PARAMETERA

pushes two words, 5 and the offset of variable PARAMETERA, onto the stack.

The signed multiply immediate instruction requires three operands and has the format:

IMUL REG,SRC,DATA

This instruction multiplies the source operand with an immediate value and stores the lower 16 bits of the product into the first operand. The upper 16 bits of the result are discarded. The source operand may be either a memory or register operand, but it must be a word, and the destination operand must be a register. For example,

IMUL BX,WORDOPR,-5

will multiply the contents of the word variable WORDOPR by -5 and store the lower 16 bits of the product into BX.

For each of the seven shift/rotate instructions, ROL, ROR, RCL, RCR, SHL/SAL, SHR, and SAR, the number of bits to shift may be specified as an immediate value with the format:

Mnemonic OPR,D8

where D8 is a positive integer less than 256. As mentioned, the number of bits to shift is D8 modulo 32. For example,

SHR BX,35

will shift BX to the right by 3 bits.

The 80186 instruction set also includes seven instructions that are not available in the 8086. These new instructions are summarized in Fig. 12-4.

The PUSHA (push all) and POPA (pop all) instructions allow all general data and pointer registers, AX, CX, DX, BX, SP, BP, SI and DI, to be saved into and restored from the stack by single instructions. It should be noted that during

| Name | Mnemonic and Format | Description |
|---------------------------|---|--|
| Push all | PUSHA | Push AX,CX,DX,BX,SP,BP,SI, and DI onto the stack in that order. The SP value pushed is the contents of the SP register before AX is pushed. |
| Pop all | POPA | Pop DI,SI,BP,SP,BX,DX,CX, and AX from the stack in that order. The popped SP value is discarded. |
| Input string | INS DST,PORT Alternate forms: INSB (Input byte string) INSW (Input word string) | Transfer a byte or word from the I/O port pointed by DX to the memory location pointed by DI. After the transfer, DI is updated using the same rules as other string primitives. |
| Output string | OUTS PORT,SRC Alternate forms: OUTSB (Output byte string) OUTSW (Output word string) | Transfer a byte or word from the memory location pointed by SI to the I/O port pointed by DX. After the transfer, SI is updated using the same rules as other string primitives. |
| Detect value out of range | BOUND REG,SRC | If $(REG) < (SRC)$ or $(REG) > (SRC+2)$ then an interrupt with type 5 occurs. |
| Block entry | ENTER SIZE,LEVEL | <ol style="list-style-type: none"> 1. Push BP. Save SP to a register, TEMP. 2. If LEVEL=0, go to Step 8. 3. If LEVEL-1=0, go to Step 7. 4. $(BP) \leftarrow (BP)-2$ 5. Push the word pointed to by BP. 6. $LEVEL \leftarrow LEVEL-1$; go to Step 3. 7. Push TEMP. 8. $(BP) \leftarrow (TEMP)$. 9. $(SP) \leftarrow (SP)-SIZE$ |
| Block exit | LEAVE | $(SP) \leftarrow (BP)$. Pop stack into BP |

Flags affected: No flags are affected.
 Addressing modes: For INS and OUTS the operands are actually ignored.
 For BOUND, SRC must point to two consecutive words in memory. For ENTER, SIZE is a nonnegative integer less than 65,536 and LEVEL is a nonnegative integer less than 256.

Figure 12-4 Summary of new instructions in the 80186.

the execution of PUSHA, the saved SP is the SP value before AX is pushed. On the other hand, the popped SP value in the POPA instruction is *not stored into the SP register but is simply discarded*.

The INS (input string) and OUTS (output string) are string primitives. Therefore, they can be combined with the REP (repeat) prefix to transfer a string of bytes or words between memory and a given I/O port. The address of the I/O port must be stored in the DX register and DX will not be changed between transfers. Just like other string primitives, the destination string in the INS instruction is

specified by ES:DI and segment register ES may not be overridden. For the OUTS instruction, the source string is specified by DS:SI. After each transfer, DI or SI will be updated according to the usual rules for string primitives, which depend on the DF setting and whether the transfer is by bytes or words.

The BOUND instruction is primarily for detecting an array subscript that is out of range. The instruction has two operands. The first operand must be the register containing the subscript to be checked. The second operand points to two consecutive memory words. The first word should contain the lower limit of the array and the second word the upper limit. If the contents of the register operand are out of range, an interrupt with type 5 will occur. Otherwise, control proceeds to the following instruction.

As an example of how to use the BOUND instruction, let us consider a three-dimensional integer array, which is defined and accessed in Pascal as follows:

```
VAR    ARY:ARRAY[5..18,-2..2,1..6] OF INTEGER;
VAR    I,J,K:INTEGER;
```

```
ARY[I,J,K]:=0;
```

A possible implementation at assembly language level would be

| | | |
|--------|----|---------------|
| LIMITS | DW | 5,18,-2,2,1,6 |
| ARY | DW | 420 DUP(?) |
| I | DW | ? |
| J | DW | ? |
| K | DW | ? |

| | |
|-------|--------------|
| MOV | AX,I |
| BOUND | AX,LIMITS |
| SUB | AX,5 |
| IMUL | AX,AX,30 |
| MOV | BX,AX |
| MOV | AX,J |
| BOUND | AX,LIMITS +4 |
| SUB | AX,-2 |
| IMUL | AX,AX,6 |
| ADD | BX,AX |
| MOV | AX,K |
| BOUND | AX,LIMITS +8 |
| SUB | AX,1 |
| ADD | BX,AX |
| ADD | BX,BX |
| MOV | ARY[BX],0 |

For storage reservation, the lower and upper limits of each subscript are defined in addition to the array itself, which consists of $14 \times 5 \times 6 = 420$ words. Before

accessing any element $\text{ARY}_{i,j,k}$, each subscript is checked against its range and then its index from the beginning of the array is calculated. For a three-dimensional array, the index of an element at i,j,k is given by

$$2 \times [(i - (\text{lower limit of } i)) \times (\text{range of } j) \times (\text{range of } k) + (j - (\text{lower limit of } j)) \times (\text{range of } k) + (k - (\text{lower limit of } k))].$$

The ENTER and LEAVE instruction pair are designed primarily to facilitate implementation of high-level languages that have block structures. An example of such a language is Pascal. In Pascal, each procedure block has the same basic structure as the main block and may contain local variables and other nested procedure blocks. A procedure has a lexical level associated with it. The main block has the highest lexical level, which is level 0, and a procedure declared in a level i procedure has level $i+1$. A level i procedure can only call directly any level $i+1$ procedure that is defined in the current procedure or another level i procedure that is declared before the calling procedure in the same level $i-1$ procedure.

Whenever a procedure is called, dynamic storage is allocated for its local variables. The procedure has access to its local variables and all variables of the higher-level procedures in which this procedure is declared. Local variables are undefined and cannot be accessed outside the procedure in which they are declared. Consider the skeleton Pascal program shown in Fig. 12-5. The main program can call only procedure LEVEL1 which has access to variables LC1A, LC1B, LC0A, LC0B, and LC0C. Procedure LEVEL2A can be called in itself, LEVEL2B and LEVEL1, but not in MAIN. Within LEVEL2A, all variables defined in LEVEL2A, LEVEL1, or MAIN can be referenced directly. However, LEVEL2A cannot directly access variables defined in LEVEL2B.

Local variables are not assigned at compilation time. A stack frame is allocated to provide the local variables when the procedure is actually entered at run-time. In fact, their addresses may change if the procedure is called again later on. Therefore, a local variable is addressed by the base address of the stack frame and its offset from that base. In order to access variables defined in higher lexical level procedures, this stack frame also should include a list of pointers to the stack frames created by all higher-level lexical procedures as well as its own stack frame. Such a stack frame can be created by the ENTER instruction.

The ENTER instruction has two operands. The first operand specifies the size (in bytes) of the local storage to be allocated and must be less than 2^{16} . The other operand specifies the lexical level of the block and is limited to the range of 0 to 255. As shown in Fig. 12-4, the ENTER instruction saves the old BP (which has the previous frame pointer), copies all the frame pointers from the previous frame, pushes the pointer of this new frame, and then subtracts SIZE from SP. For example, if

ENTER 20,3

is executed at the entry of a level 3 procedure, a new stack frame will be created, as shown in Fig. 12-6. In that procedure, registers can be saved to the area above

```

PROGRAM MAIN;
  VAR LCOA,LCOB,LCOC:INTEGER;
  PROCEDURE LEVEL1 (Parameters);
    VAR LC1A,LC1B:INTEGER;
    PROCEDURE LEVEL2A (Parameters);
      VAR LC2A,LC2B:INTEGER;
      PROCEDURE LEVEL3 (Parameters);
        VAR LC3A,LC3B:INTEGER;
        BEGIN      {LEVEL3}
          .
          .
          .
          LC3A:=1;
          LC2B:=0;
          LC1A:=LC1A+1;
          LCOA:=LCOB+1;
          .
          .
        END;      {LEVEL3}
      BEGIN      {LEVEL2A}
        .
        .
        LEVEL3 (Actual parameters);
        .
      END;
      PROCEDURE LEVEL2B (Parameters);
        VAR LC2AB,LC2BB:INTEGER;
        BEGIN      {LEVEL2B}
          .
          .
          END;      {LEVEL2B}
        BEGIN      {LEVEL1}
          .
          .
          LEVEL2A (Actual parameters);
          .
          .
          LEVEL2B (Actual parameters);
          .
        END;
        BEGIN      {MAIN}
          .
          .
          LEVEL1 (Actual parameters);
          .
        END.
      
```

Figure 12-5 A skeleton Pascal program to illustrate block structures.

the local storage. On exit, the created stack should be released by copying BP to SP and then restoring the BP (i.e., the pointer to the previous frame). This operation is performed by the LEAVE instruction.

To illustrate some of the principles of block structures, a possible implementation of the skeleton program in Fig. 12-5 is shown in Fig. 12-7. Figure 12-8 shows how the stack changes during the program execution. Notice that while local variables are on the stack, the variables declared in the main program normally have fixed locations assigned at compilation time and are accessible through the DS register.

Since the stack frame activated by a procedure contains the pointers to the current frame and those frames associated with the enclosing procedures, the cur-

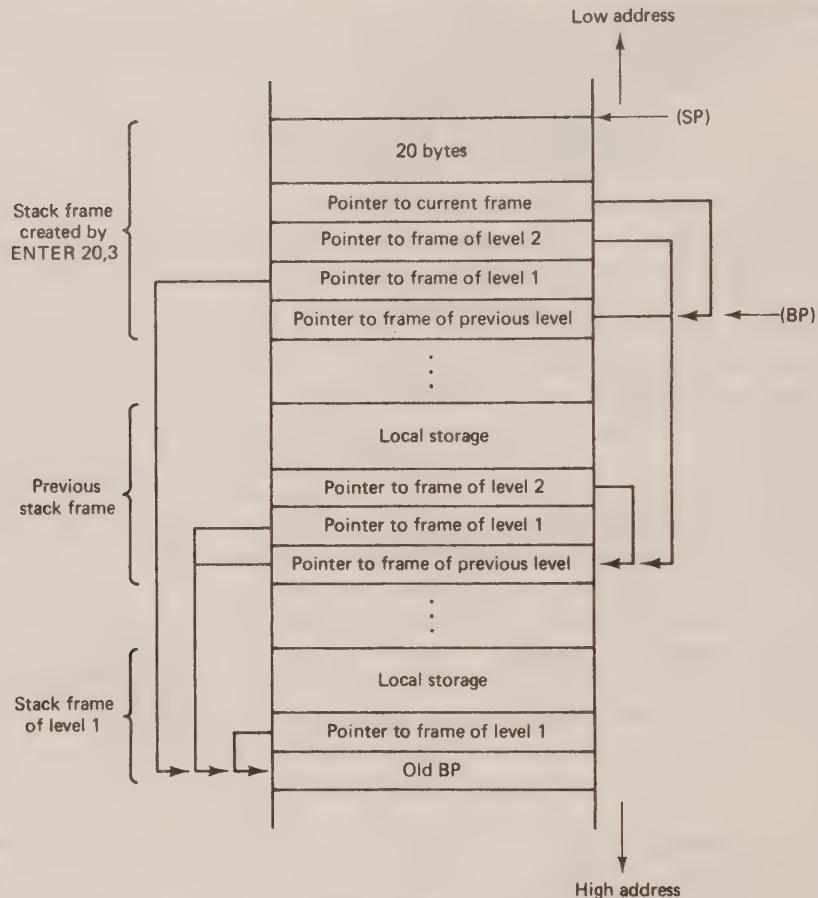


Figure 12-6 Stack frame caused by an `ENTER` instruction.

rent procedure can access their variables directly through those frame pointers. For example, in procedure LEVEL3, the following Pascal statements:

```
LC3A:=1;
LC2B:=0;
LC1A:=LC1A+1;
```

could be translated into

| | |
|-----|----------------------|
| MOV | WORD PTR [BP-10],1 |
| MOV | BX,[BP-4] |
| MOV | SS:WORD PTR [BX-6],0 |
| MOV | SI,[BP-2] |
| INC | SS:WORD PTR [BX-6] |

On the other hand, variables `LC0A` and `LC0B` are declared in the main program. They are static variables and have fixed locations. Therefore, statement

```
LC0A:=LC0B+1;
```

```

        .
LEVEL3    PROC    NEAR
ENTER    4,3          ;PROVIDE SPACE FOR LC3A AND LC3B
        .
        LEAVE
        RET    *
LEVEL3    ENDP
LEVEL2A   PROC    NEAR
ENTER    4,2          ;PROVIDE SPACE FOR LC2A AND LC2B
        .
        CALL NEAR PTR LEVEL3
        .
        LEAVE
        RET    *
LEVEL2A   ENDP
LEVEL2B   PROC    NEAR
ENTER    4,2
        .
        LEAVE
        RET    *
LEVEL2B   ENDP
LEVEL1    PROC    NEAR
ENTER    4,1
        .
        CALL NEAR PTR LEVEL2A
        .
        CALL NEAR PTR LEVEL2B
        .
        LEAVE
        RET    *
LEVEL1    ENDP
MAIN      .
        CALL NEAR PTR LEVEL1
        .

```

*Might have an operand specifying the number of bytes used for passing parameters.

Figure 12-7 An example of implementing blocks using ENTER and LEAVE instructions.

would be implemented as

```

MOV     AX,LC0B
INC     AX
MOV     LC0A,AX

```

12-2-2 Internal Architecture

The 80186 integrates a clock generator, a DMA controller, a programmable interrupt controller, a timer unit consisting of three timers, and a chip select unit into the processor IC. It is fabricated into a quad-in-line package with 68 pins. Figure 12-9 shows a block diagram of the 80186. Since these supporting and peripheral devices are commonly required, the 80186 can reduce the chip count and simplify the design of a typical system.

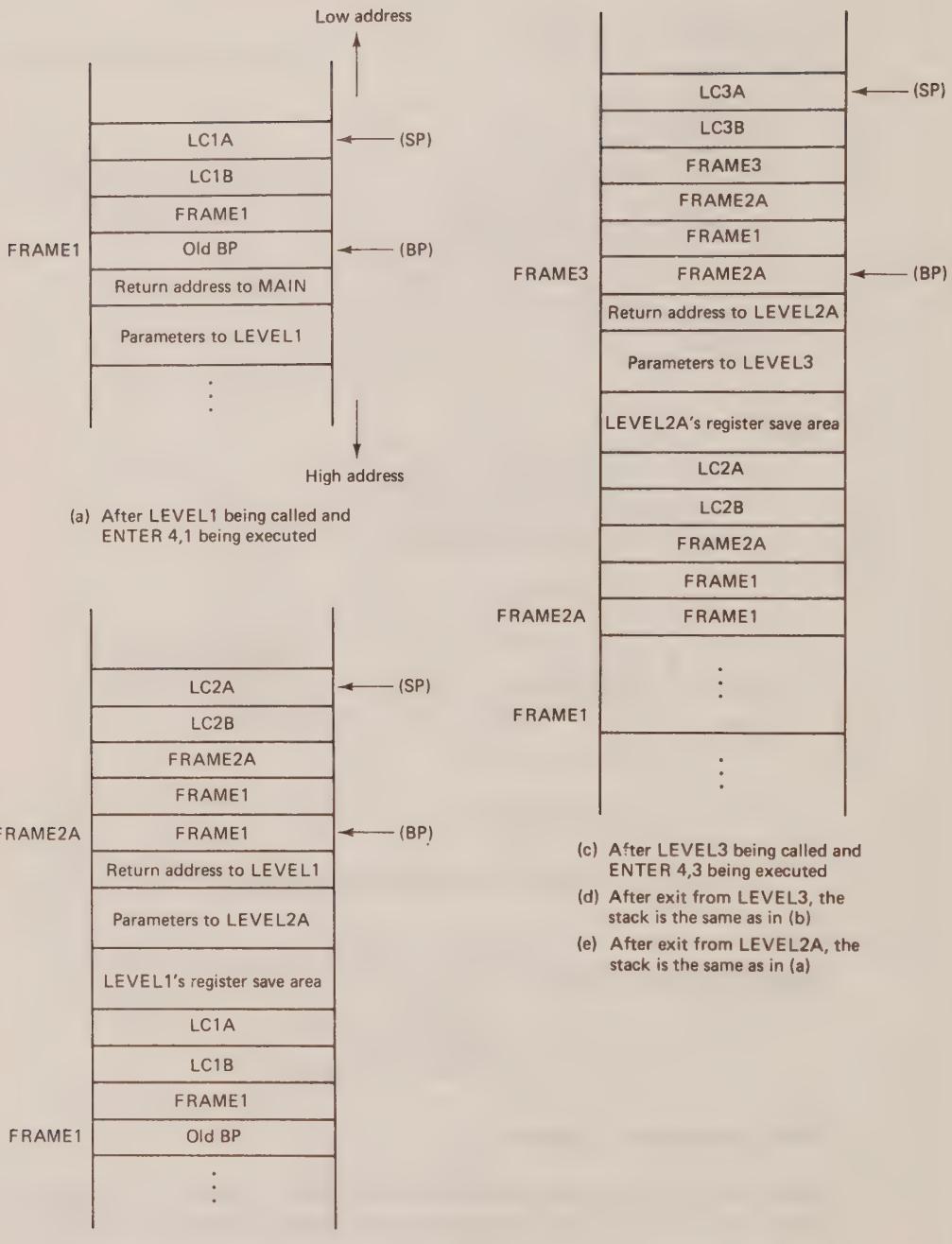
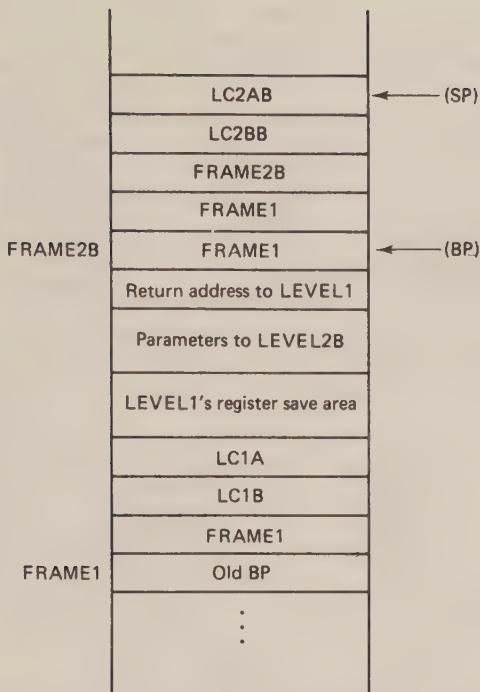


Figure 12-8 Contents of the stack while nested blocks are executed.



- (f) After LEVEL2B being called and ENTER 4,2 being executed
- (g) After exit from LEVEL2B, the stack is the same as in (a)

Figure 12-8 (continued)

With the integrated clock generator, the 80186 requires only a crystal to provide the clock signal. The maximum internal clock rate for the 80186/80188 is 6 or 8 MHz, depending on the model. However, the crystal's frequency should be two times the internal clock rate (i.e., a maximum of 16 MHz for the 80186/80188 and 12 MHz for the 80186-6/80188-6) because the input frequency is internally divided by two to generate the clock signal.

The integrated peripherals are programmable through their registers, which are in a 256-byte control block as shown in Fig. 12-10. This block can be either in the memory or the I/O space. Its base address, which must be divisible by 256, can be reassigned through the relocation register. Each register, including the relocation register, is accessed as a memory location (or I/O port) whose address is equal to its fixed offset plus the base address. The lower 12 bits of the relocation register multiplied by 256 is the base address of the control block. Bit 12 indicates whether the control block is assigned to memory space (1) or to I/O space (0). Bit 13 is not used. Bit 14 specifies whether the interrupt controller is to be in the master mode (0) or the iRMX compatible mode (1), in which the internal controller is a slave. Bit 15 is used to cause a trap (1) or no trap (0) whenever an ESC

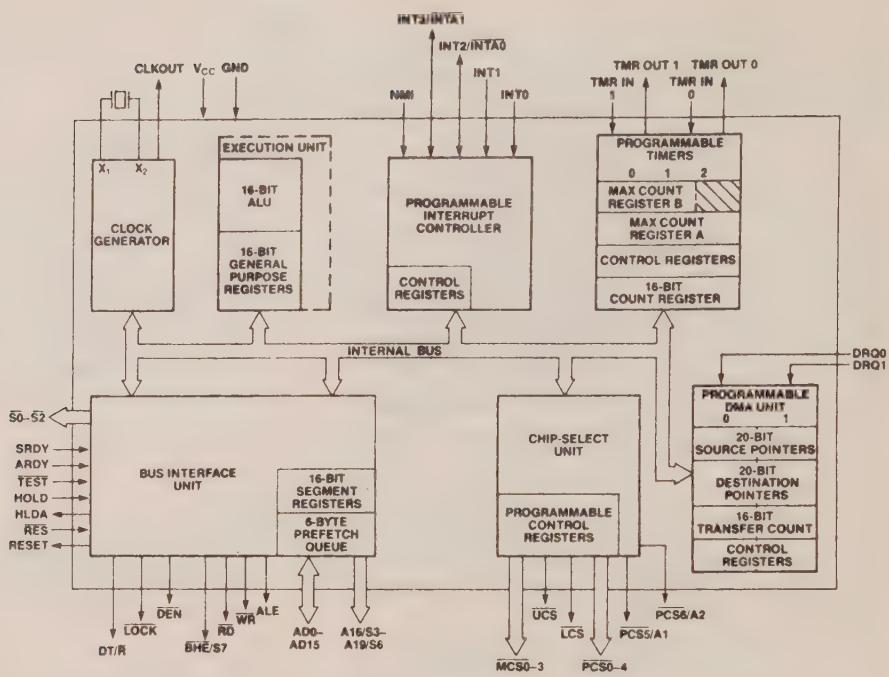


Figure 12-9 Block diagram of the 80186. (Reprinted by permission of Intel Corporation. Copyright 1982.)

instruction is executed. Upon reset of the 80186, the relocation register is set to 20FF. This means that the control block is located from FF00 through FFFF in the I/O space.

The following instruction sequence illustrates how the control block can be reassigned, for example, to memory locations AFF00 through AFFF.

| | |
|-----|---------|
| MOV | DX,0FFE |
| MOV | AX,1AFF |
| OUT | DX,AX |

Then, all internal peripheral registers will be accessible as memory operands.

The DMA controller provides two independent channels, each with a DMA request pin. For each channel, a transfer may be initiated by a request from the request pin, a timer out signal from timer 2, or a request from the controller itself. In the latter case, transfers take place continuously, one after another. During each DMA transfer, the 80186 first reads a byte or word from the source and then writes the data into the destination. This eliminates the need for DMA acknowledge pins but limits the maximum transfer rate to 1M words per second (assuming no odd word address is involved).

Figure 12-11 shows the layout of the DMA control block. Each channel has a control register, a counter register, a destination pointer, and a source pointer. The counter register stores the number of bytes or words to be transferred and is

| Relocation Register | OFFSET FEH |
|--------------------------------|---------------|
| | |
| DMA Descriptors Channel 1 | DAH DOH |
| | |
| DMA Descriptors Channel 0 | CAH COH |
| | |
| Chip-Select Control Registers | A8H A0H |
| | |
| Timer 2 Control Registers | 66H 60H |
| Timer 1 Control Registers | 5EH 58H |
| Timer 0 Control Registers | 56H 50H |
| | |
| Interrupt Controller Registers | 3EH 20H |

Figure 12-10 Control register map.
(Reprinted by permission of Intel Corporation. Copyright 1982.)

decremented by 1 after each DMA cycle. Each pointer, which has 20 bits, can independently specify either an I/O or memory location. Therefore, the transfer direction can be memory to memory, memory to I/O, I/O to memory, or I/O to I/O. To address an I/O port, the upper four bits of the pointers should be programmed to 0. After each transfer, each pointer may be incremented by 1 or 2, decremented by 1 or 2, or remain unchanged. This allows a transfer to use increasing addresses, decreasing addresses, or cause the contents of a memory location or I/O port to be duplicated.

The definitions of the bits in the control register are:

BYTE/WORD—0 for byte transfers and 1 for word transfers.

START/STOP—Channel is enabled when it is set to 1. It can be reset to 0 to stop DMA transfer if the CHG/NOCHG bit is 1. Also, it will be reset to 0 to indicate the end of DMA transfer if the TC bit is 1.

CHG/NOCHG—If this bit is set when the control register is written, the START/STOP bit will be programmed by the write. Otherwise, the START/STOP bit will not be affected by the write.

TDRQ—This bit is used to enable (1) or disable (0) timer 2 for generating internal DMA requests.

P—Specifies channel priority, 0 for low priority and 1 for high priority. If both channels have the same priority, DMA grants will alternate when both channels have requests pending.

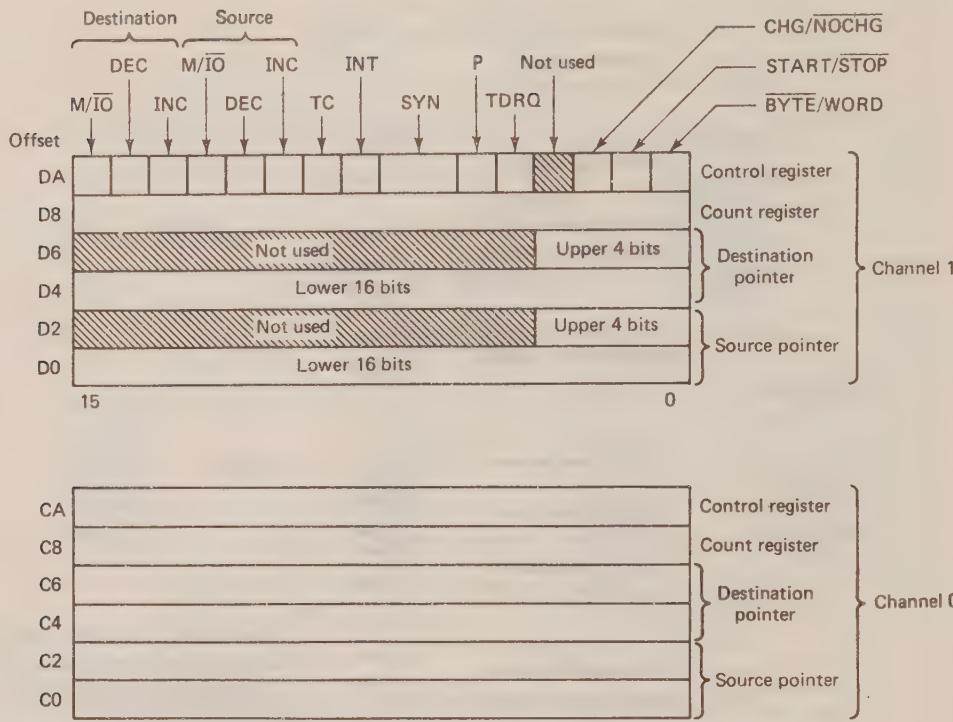


Figure 12-11 The 80186 DMA control block.

SYN—These two bits specify synchronization modes. If they are 00, DMA cycles are not synchronized. In other words, a DMA cycle is initiated either at the maximum rate or by a request from timer 2, depending on the TDRQ bit. If they are 01, the DMA is source synchronized and the DRQ pin should be connected to the source. If they are 10, the DRQ pin should be connected to the destination and the DMA is destination synchronized.

INT—If this bit is set, the DMA controller will generate an interrupt request when the transfer count reaches 0. Changing the INT bit will also cause the corresponding interrupt enable bit in the interrupt controller to be altered accordingly.

TC—If it is set, the DMA will terminate and the START/STOP bit will be cleared when terminal count reaches 0. Therefore, START/STOP may be examined as a status bit. If the TC bit is cleared, the DMA will continue and the count register will be decremented even after terminal count reaches 0.

Source: M/IO, DEC, INC—The M/IO bit indicates whether the source pointer addresses the memory (1) or I/O space (0). The DEC and INC bits determine how the source pointer is to be updated. If they are 01, depending on the BYTE/WORD bit, the source pointer will be incremented by 1 or 2 after

each DMA cycle. If they are 10, the pointer will be decremented by 1 or 2. For 00 or 11, the pointer will remain unchanged.

Destination: M/IO, DEC, INC—These bits have the same definitions as they do for the source.

Since a DMA controller in an 80188 can perform only byte transfers, the BYTE/WORD bit in the DMA control word is ignored.

The 80186's timer unit consists of three programmable timers. Each timer may generate an interrupt request when its terminal count is reached. Timers 0 and 1 have a total of four pins (TMR IN0, TMR IN1, TMR OUT0, and TMR OUT1) to accept external control signals and to output terminal count signals. The third timer has no external pins, and its counter is driven by the internal CPU clock at one fourth of the CPU clock rate. It is typically used for initiating DMA cycles or as a prescaler to the other two timers.

A diagram of the timer control block is given in Fig. 12-12. Except for timer 2, which does not have a max count B register, each timer has a control register, a max count A register, a max count B register, and a count register. Max count registers A and B are used to store two maximum counts. They can be alternately selected to provide the maximum count, and one can be changed while the other is being used.

Each count causes the count register to be incremented by 1. This register is reset to 0 whenever the designated maximum count is reached. A timer can be programmed by writing to its control register. It can be programmed so that when

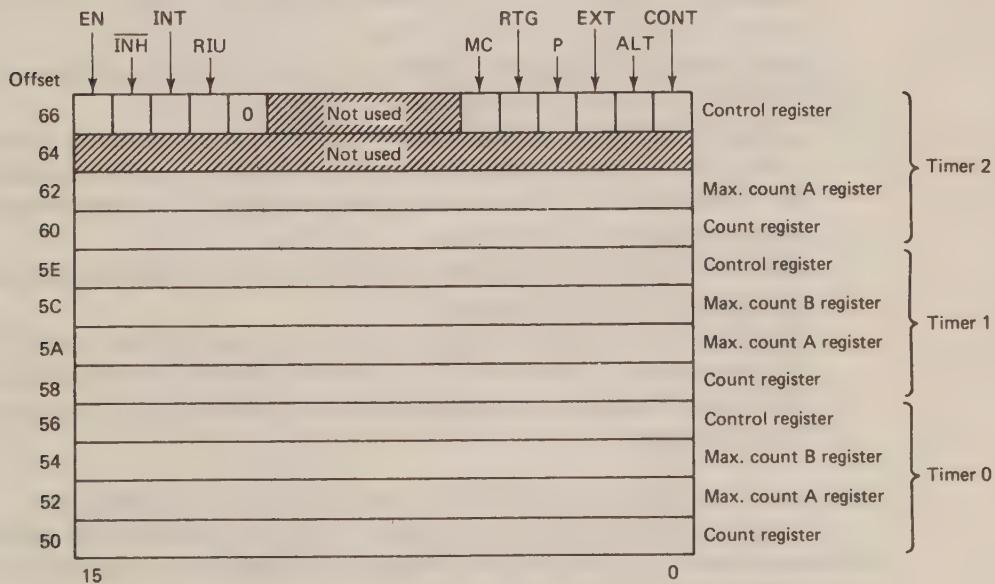


Figure 12-12 The 80186 timer control block.

the terminal count is reached it will halt, be restarted by the TMR IN pin, or continue. The control register's bit definitions are:

CONT—If this bit is 1, the timer will continue to run after the count register is reset to 0 on the terminal count. If both the CONT and ALT bits are 0, the timer will stop on the terminal count. If **CONT** = 0 and **ALT** = 1, the timer will continue until the terminal count in max count register B is reached.

ALT—This bit serves two purposes, to select a max count register and to specify the output on the TMR OUT pin. When this bit is cleared, max count register A is always used and a negative pulse of one clock cycle will be sent to the TMR OUT pin after the maximum count is reached. Otherwise, on a terminal count the other max count register will be selected, and the TMR OUT pin will always indicate which max count register is currently being used (1 for A and 0 for B).

EXT—This bit specifies whether the clock signal is from the internal CPU clock (**EXT** = 0) or from the TMR IN pin (**EXT** = 1). When **EXT** = 0, counting is enabled according to the RTG bit and TMR IN pin.

P—This bit is meaningful only if **EXT** = 0. When **P** = 1, the timer will count at a rate prescaled by timer 2—i.e., the terminal count signal from timer 2 is used as the timer clock. Otherwise, the timer will count at one fourth of the internal CPU clock rate.

RTG—This bit is meaningful only if **EXT** = 0. When **RTG** = 1, a low to high transition on the TMR IN pin will clear the count register and start counting. When **RTG** = 0, the timer will stop if TMR IN is low and will continue counting if TMR IN is high.

MC—This bit is a status bit which is set to 1 each time a maximum count is reached. It needs to be cleared by the program.

RIU—This status bit indicates whether register A or register B is being used as the current max count register; 0 indicates register A and 1 indicates register B. When the **ALT** bit is a 0, max count register A is always selected and, thus, the **RIU** bit is cleared.

INT—If this bit is set, the timer will generate an interrupt request each time the maximum count is reached.

INH—If this bit is set when writing to the control register, the EN bit will be programmed by the write. If it is cleared, the EN bit will not be altered.

EN—When this bit is set, the timer is enabled; otherwise, the timer is inhibited from counting.

For timer 2, the **ALT**, **EXT**, **P**, **RTG**, and **RIU** bits are hardwired to 0.

As an example, let us consider how to use the timer unit to measure the execution time of a given program sequence to within one hundredth of a second for an 8-MHz 80186. Let us assume that:

1. The 80186 peripheral control block starts at its reset location, FF00.

2. The TMR IN pin of timer 1 is connected to +5 V.
3. Execution time, measured in hundredths of a second, does not exceed a count of $2^{16} - 1$ and is to be stored in the variable TIME_ELAPSE.
4. If the count exceeds $2^{16} - 1$, the variable OVERFLOW is set to 1.

Figure 12-13 shows the necessary instructions to be placed before and after the program sequence whose execution time is to be determined. It is seen that timer 1 is enabled to count up by one whenever timer 2 reaches its terminal count. Since timer 2 can only count at one fourth of the CPU clock rate (2 MHz = 8 MHz/4), its maximum count is set to 0.01 s divided by 500 ns, which is 20,000. After the given program sequence is completed, both timers 1 and 2 are disabled and the measured time is obtained from the count register of timer 1.

The integrated interrupt controller can accept interrupt requests from internal timers and DMA channels as well as external interrupt requests. When bit 14 of the relocation register is 1, this controller operates in a special iRMX 86 compatible mode; otherwise, it acts as the master interrupt controller.

In the special iRMX 86 mode, the internal controller acts as a slave to the external master controller so that the 80186 becomes compatible with the interrupt structure of the iRMX 86 operating system. Under this mode, the INT0 and INT2 pins provide the same functions as the INTR and INTA pins do in the 8086. The

```

MOV DX,OFF5AH      ;SEND MAX COUNT FFFFH TO MAX COUNT
MOV AX,0FFFFH      ;A REGISTER OF TIMER 1
OUT DX,AX
MOV DX,OFF58H      ;SET COUNT REGISTER OF TIMER 1 TO 0
XOR AX,AX
OUT DX,AX
MOV DX,OFF5EH      ;SEND CONTROL WORD TO ENABLE TIMER 1
MOV AX,OC008H      ;TO COUNT AT AN INTERNAL RATE PRESCALED
OUT DX,AX
MOV DX,OFF62H      ;SEND MAX COUNT 20000 TO MAX COUNT A REGISTER
MOV AX,20000        ;OF TIMER 2 (20000 X 500 NS = 0.01 SEC)
OUT DX,AX
MOV DX,OFF60H      ;SET COUNT REGISTER OF TIMER 2 TO 0
XOR AX,AX
OUT DX,AX
MOV DX,OFF66H      ;SEND CONTROL WORD TO TIMER 2 TO START
MOV AX,OC001H      ;COUNTING AND TO CONTINUE ON A
OUT DX,AX          ;TERMINAL COUNT
.
.
.
MOV DX,OFF66H      ;PROGRAM SEQUENCE WHOSE EXECUTION
MOV AX,4000H        ;TIME IS TO BE MEASURED
OUT DX,AX
MOV DX,OFF5EH      ;DISABLE TIMER 2
IN AX,DX
AND AX,20H
JZ NOTOVER
MOV OVERFLOW,1      ;IF MC=1, SET OVERFLOW TO 1
JMP SHORT NEXT
NOTOVER: MOV OVERFLOW,0  ;IF MC=0, SET OVERFLOW TO 0
NEXT:   MOV DX,OFF58H  ;READ FINAL COUNT
IN AX,DX
MOV TIME_ELAPSE,AX
MOV DX,OFF5EH      ;DISABLE TIMER 1
MOV AX,4000H
OUT DX,AX

```

Figure 12-13 Example of using the timer unit to measure program execution time

INT3 pin sends interrupt requests caused by the internal peripherals to the external master 8259A, and the INT1 pin is used to accept a slave select signal. For a detailed description of the control block under the iRMX 86 mode, the reader should refer to Reference 2.

In the non-iRMX 86 (i.e., the master) mode, the controller acts as the master interrupt controller and its operation is very similar to that of the 8259A. There are three modes which can be selected by programming the INT0 and INT1 control registers. The major difference between these modes is the functions of pins INT0, INT1, INT2/INTA0, and INT3/INTA1. Internal interrupts caused by the timers and the DMA controller channels are serviced in the same way for all modes. The three master modes are:

Fully Nested Mode—All four pins are used as interrupt request pins (INT0, INT1, INT2, and INT3) which are directly connected to interrupt sources. The interrupt types for these inputs are generated internally.

Cascade Mode—The four pins are divided into two pairs: INT0 and INT2/INTA0, and INT1 and INT3/INTA1. Each pair can be individually programmed to be in the cascade mode or the fully nested mode. When a pair is in the cascade mode, it serves as a request/acknowledge pair which can be connected to an 8259A or cascaded 8259As. When attached to cascaded 8259As, up to 64 interrupt lines can be serviced by the pair. Unlike the fully nested mode, interrupt types are generated externally.

Special Fully Nested Mode—This mode has the same external connection as a cascade mode. The only difference is that, unlike the cascade mode, the 80186 interrupt controller will allow additional interrupts from an external 8259A even though the in-service bit corresponding to the request pin connected to that 8259A is set. (See Sec. 8-3-2 for a discussion of the special fully nested mode.)

The control block of the interrupt controller under a non-iRMX 86 mode is illustrated in Fig. 12-14. The interrupt controller handles interrupt requests from the INT3-INT0 pins as well as from internal peripheral devices. Each interrupt source, INT0, INT1, INT2, INT3, DMA channel 0, DMA channel 1, and the timer unit has an associated control register. The least significant 3 bits of a control register are used to specify a priority from 0 to 7, with 0 being the highest priority level. Note that all three timers share the same control register and, hence, the same priority level. However, among them timer 0 has the highest priority and timer 2 has the lowest priority. The next bit is the mask bit; when it is 1 the interrupt is disabled. The LTM bit determines whether the request signal is level-triggered (LTM = 1) or edge-triggered (LTM = 0). In addition, the INT0 and INT1 control registers have a cascade mode (C) bit and a special fully nested mode (SFNM) bit. If the C and SFNM bits are 10, the corresponding pair of request/acknowledge pins is configured in the cascade mode. To specify the special fully nested mode, both bits should be set to 1.

The interrupt types for the internal devices are generated by the controller.

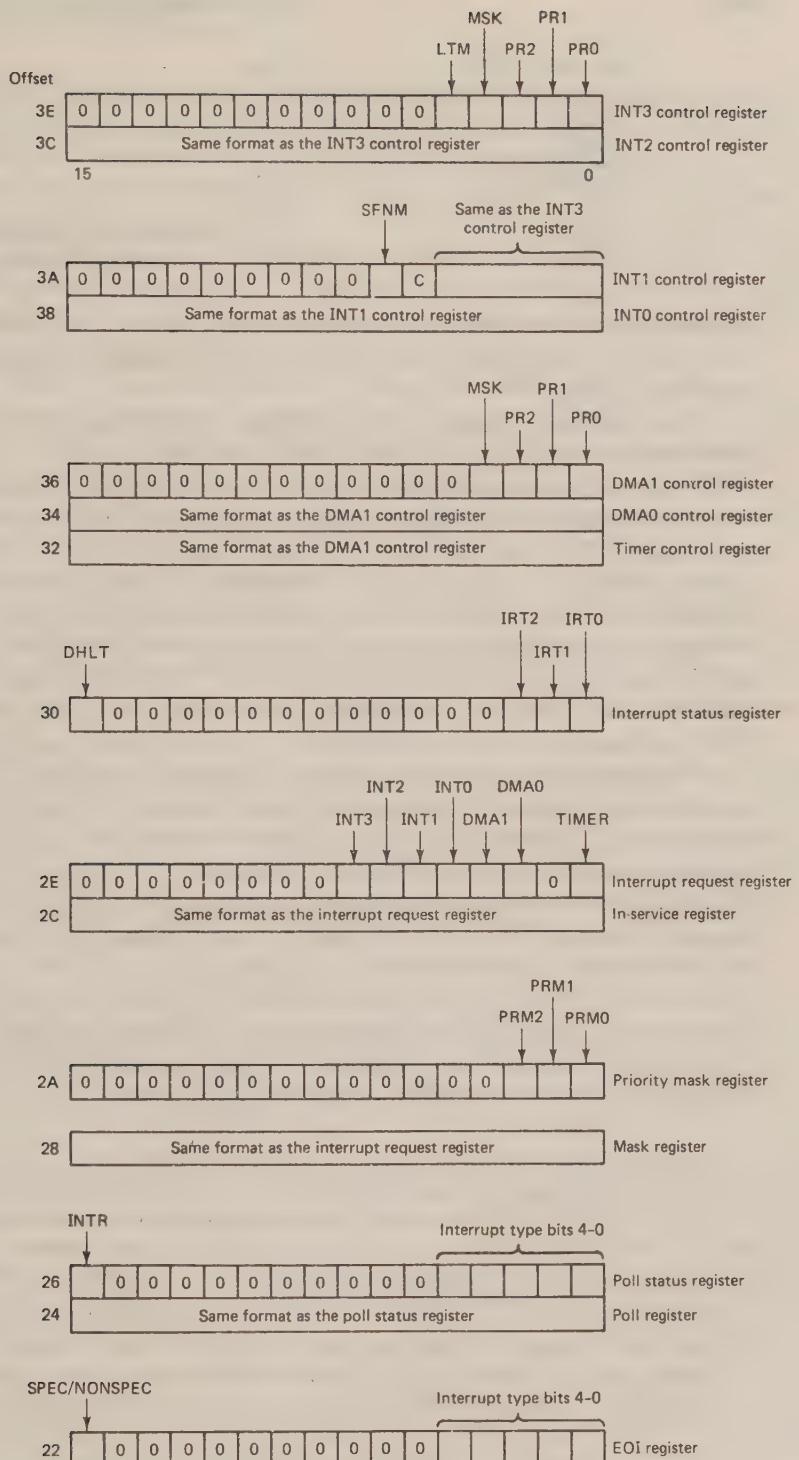


Figure 12-14 The 80186 interrupt control block (non-iRMX 86 mode).

| Interrupt Source | Type |
|------------------|------|
| Timer 0 | 8 |
| Timer 1 | 18 |
| Timer 2 | 19 |
| DMA Channel 0 | 10 |
| DMA Channel 1 | 11 |
| INT0 | 12 |
| INT1 | 13 |
| INT2 | 14 |
| INT3 | 15 |

Figure 12-15 Interrupt types generated by the 80186's interrupt controller.

In the fully nested mode, the controller also generates interrupt types for INT0, INT1, INT2, and INT3. These interrupt types cannot be changed and they are listed in Fig. 12-15.

The interrupt status register indicates which timer is causing an interrupt (INT2-INT0), and whether DMA transfers are halted (DHLT = 1) or not (DHLT = 0). The priority mask register indicates the priority of the current interrupt being serviced. Both the poll status and poll registers show whether or not an interrupt is pending (INTR) and the interrupt type of the pending request with the highest priority. Reading the poll register will cause the appropriate bit in the in-service register to be set just as if an interrupt request had been acknowledged, and this will affect the interrupt request and priority mask registers. However, reading the poll status register will *not* cause this activity. The EOI register is for the user to issue an end of interrupt command. Refer to Sec. 8-3-1 for the meaning of the EOI command and the interrupt request, in-service, and mask registers.

To simplify memory and peripheral interfaces, the 80186 provides chip select signals which eliminate the need for these interfaces to have their own address decoder logic. Thirteen chip select signals are provided: six for selecting memory modules and seven for selecting I/O peripherals. The range of each select signal is programmable through the memory and peripheral select control block whose layout is shown in Fig. 12-16. In addition, ready signals for terminating bus cycles can be generated internally with a programmed delay of 0 to 3 wait states.

Of the six pins for designating memory chip select signals, one is the upper memory chip select (UCS), one is the lower memory chip select (LCS), and the other four are the mid-range memory chip selects (MCS3-MCS0). The beginning address of the memory block selected by UCS is defined by the UMCS register

| Offset | 15 | M6 | M5 | M4 | M3 | M2 | M1 | M0 | EX | MS | 1 | 1 | 1 | R2 | R1 | R0 | MPCS register |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|---|---|----|----|----|---------------|
| A8 | 1 | | | | | | | | | | | | | | | | MMCS register |
| A6 | A19 | A18 | A17 | A16 | A15 | A14 | A13 | 1 | 1 | 1 | 1 | 1 | 1 | R2 | R1 | R0 | PACS register |
| A4 | A19 | A18 | A17 | A16 | A15 | A14 | A13 | A12 | A11 | A10 | 1 | 1 | 1 | R2 | R1 | R0 | LMCS register |
| A2 | 0 | 0 | A17 | A16 | A15 | A14 | A13 | A12 | A11 | A10 | 1 | 1 | 1 | R2 | R1 | R0 | UMCS register |
| A0 | 1 | 1 | A17 | A16 | A15 | A14 | A13 | A12 | A11 | A10 | 1 | 1 | 1 | R2 | R1 | R0 | |
| | | | | | | | | | | | | | | | | 0 | |

Figure 12-16 The 80186 memory and peripheral select control block.

while the ending address of this memory block is always FFFF. Bits 15-6 of the UMCS register specify address bits A19-A10 of the 20-bit lower limit with bits A9-A0 being 0. Note that bits 15-14 of the UMCS register are always set to 1 so that the smallest possible lower limit is C0000. The size of the upper memory range must be 2^n K bytes with $n = 0, 1, \dots, 8$. Therefore, the 2's complement of bits 14-6 of the UMCS, which corresponds to the block size in K bytes, must have one and only one bit equal to 1. Any other combination may cause an undefined operation. For example, if bits 13-6 of this register are 11100000, then the beginning address is F8000 and the block size is 32K bytes.

The lower three bits of the UMCS register specify how to generate an internal ready signal if the upper memory block is accessed. Bits R1 and R0 specify the number of wait states (0 to 3) to be inserted. Bit R2 indicates whether external RDY (i.e., ARDY or SRDY) also is used ($R2 = 0$) or is ignored ($R2 = 1$). For example, if R2 through R0 are 000, the actual number of wait states is determined by the external ready signals. The R2-R0 bits in the other four chip select control registers have the same meanings.

For the lower memory block, the beginning address is always 0 and bits 15-6 of the LMCS register specify bits A19-A10 of the upper address. Bits A9-A0 of the upper limit are always 1. Once again the block size can only be 2^n K bytes, with $n = 0, 1, \dots, 8$, and a valid setting of bits 13-6 of the LMCS register must represent $2^n - 1$.

The mid-range memory block is divided equally into four areas. One of the MCS3-MCS0 signals will be active if the memory address is within the mid-range block. MCS0 corresponds to the lowest quarter and MCS3 to the highest quarter of the defined mid-range memory block. The size of the entire block is defined by bits 14-8 of the MPCS register. One and only one of these seven bits must be set to one. If $Mi = 1$ and the other six bits are zero, the selected block is $2^i \times 8$ K bytes. Therefore, the smallest valid block size is 8K and the largest size is 512K. The MS and EX bits of the MPCS register are used for peripheral select signals.

The beginning address of the mid-range memory block is defined by the MMCS register. Bits 15-9 of this register correspond to bits A19-A13 of the block's beginning address, with A12-A0 being 0. However, the beginning address must be a multiple of the selected block size and the mid-range block may not overlap with the areas defined for UCS and LCS. For example, if the MPCS register contains 8838 and the MMCS register contains 31F8, then the entire block starts at address 30000 with the total size equal to 64K. In addition, MCS0, MCS1, MCS2, and MCS3 corresponds to four 16K memory areas: 30000-33FFF, 34000-37FFF, 38000-3BFFF, and 3C000-3FFFF, respectively. The lower three bits of the MMCS register can be programmed to provide an internal READY signal for the mid-range block.

The 80186 provides seven peripheral select signals, PCS6-PCS0, which fall within seven contiguous blocks of 128 bytes above the base address specified in the PACS register. Bits 15-6 specify bits A19-A10 of the base address and bits A9-A0 are 0—i.e., a base address must be a multiple of 1K. PCS0 is for the lowest 128 bytes (base address to base address + 127) and PCS6 is for the highest 128 bytes (base address + 768 to base address + 895). If the specified address is to

be in the I/O space, the MS bit of the MPCS register and bits 15-12 of the PACS register should be cleared. When MS = 1, the specified base address is in the memory space. $\overline{PCS5}$ and $\overline{PCS6}$ can also be programmed to provide latched address bits A1 and A2 instead of select signals. This is done by clearing the EX bit of the MPCS register. The A1 and A2 signals are useful in selecting internal registers of a peripheral interface device. The READY signal associated with PCS3-PCS0 is controlled by the R2-R0 bits of the PACS register while the READY signal associated with PCS6-PCS4 is controlled by the lower three bits of the MPCS register.

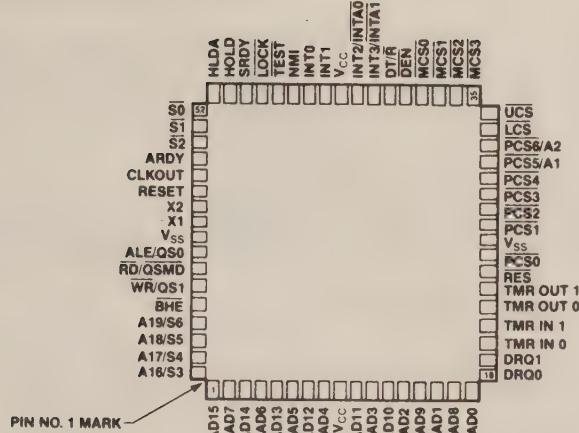
12-2-3 Bus Structure

In addition to the new pins designated for address selection and internal devices, the 80186 retains most of the 8086's pins for address, data, and bus control. The pin definitions of the 80186 are summarized in Fig. 12-17 along with its pin diagram. Due to this upward compatibility, bus interface and control devices such as 8282, 8286, 8288, and 8289, which are designed for the 8086/8088 family, can support the 80186 as well.

The major differences between the 80186's bus control pins and those of the 8086 are that the 80186 does not have an MN/MX strapping pin and it concurrently provides control signals for both the minimum mode (\overline{RD} , \overline{WR} , ALE, \overline{DEN} and $\overline{DT/R}$) and the maximum mode ($\overline{S0}$, $\overline{S1}$, $\overline{S2}$, and \overline{LOCK}). As a result, the 80186 can support two buses with one bus controller (8288). A local bus can be accessed by signals directly from the 80186 while a system bus is being accessed through an 8288. However, the queue status bits QS0 and QS1 share the same pins with the ALE and \overline{WR} signals. To output QS0 and QS1 (as required in a coprocessor configuration), the \overline{RD} pin must be grounded. In addition, the 80186 does not provide M/IO (this information can be obtained from the $\overline{S2}$ pin), and status bits S5-S3 are always 0.

An example of an 80186-based configuration is shown in Fig. 12-18. Note that the 80186 is connected to two buses. The local bus is to access the reset ROM and local RAM while the system bus, which is MULTIBUS compatible, is to access shared memory and I/O devices. Since local devices are accessed only by the 80186, they can be selected by signals directed from the 80186, thus eliminating address decoding logic. If local memory and local devices are not accessed (\overline{UCS} , \overline{LCS} , $\overline{PCS1}$, and $\overline{PCS0}$ are high), then the system bus is enabled. This figure also illustrates how the 80186's internal peripheral support devices, such as its clock generator and chip select logic, can reduce the system chip count.

The 8087, numerical coprocessor designed for the 8086/8088 family, can be used to support the 80186/80188 as well. However, for such an application an integrated bus controller (82188) is required. This device synchronizes the queue status signals QS0 and QS1 and converts the HOLD and HLDA signals of the 80186 to the 8087's $\overline{RQ}/\overline{GT}$ signals. A typical 80186/8087 configuration is illustrated in Fig. 12-19. The IOP (8089) can also be used in an 80186/80188 system if the IOP is configured in the remote mode.



(a) Pin diagram

| Symbol | In/Out | Description |
|-------------------|--------|---|
| V _{cc} / | --- | +5 V power supply. |
| V _{ss} / | --- | Ground. |
| <u>RES</u> | I | System reset is used to initialize the 80186. During the reset, the 80186 clears IP, DS, SS, ES and the instruction queue, and sets PSW to F002, CS to FFFF, relocation register to 20FF, and UMCS to FFFB. |
| AD15-ADO | I/O-3 | Address/data pins provide address/data during a bus cycle (like the 8086). |
| A19/S6-A16/S3 | 0-3 | Address/status pins provide address/status. Unlike the 8086, S6=0 for a processor cycle and S6=1 for a DMA cycle, and S5, S4, and S3 are 0. |
| <u>BHE/S7</u> | 0-3 | Bus high enable/status bit 7 is used to enable the high byte of the data bus (like the 8086). S7 is equivalent to <u>BHE</u> . |
| DT/ <u>R</u> | 0-3 | Data transmit/receive indicates the direction of data flow on the data bus (like a minimum mode 8086). |
| <u>DEN</u> | 0-3 | Data enable provides output enable for data bus transceivers (like a minimum mode 8086). |
| <u>S0, S1, S2</u> | 0-3 | Indicates the type of current bus cycle (like a maximum mode 8086). |

Figure 12-17 Pin definitions for the 80186. (Part (a) is reprinted by permission of Intel Corporation. Copyright 1982.)

| | | |
|--|------|---|
| ALE/QSO | 0 | Indicates an address is available on the address pins when the RD/QSMD pin is not grounded (like a minimum mode 8086). Unlike the 8086, if the RD/QSMD pin is grounded, the queue status bit 0 is sent instead. |
| WR/QS1 | 0-3 | Indicates a write command (like the advanced write of the 8288) when the RD/QSMD pin is not grounded. Unlike the 8086, if the RD/QSMD pin is grounded, the queue status bit 1 is sent instead. |
| <u>RD/QSMD</u> | 0-3 | Indicates a read command when this pin is not grounded (like a minimum mode 8086). Unlike the 8086, this pin can be grounded so that the 80186 outputs QS0 and QS1 instead of ALE and WR. |
| LOCK | 0-3 | Used to prevent other bus masters from accessing the system bus (like a maximum mode 8086). |
| TEST | I | Examined by the WAIT instruction (like the 8086). |
| NMI | I | Non-maskable interrupt (like the 8086). |
| HOLD, HLDA | I, O | Hold request and hold acknowledge are used by other bus masters for gaining control of the system bus (like a minimum mode 8086). |
| RESET | 0 | Indicates that the 80186 is being reset and can be used as a system reset signal. |
| X1, X2 | I | Provides an external connection for the internal clock generator's crystal oscillator. |
| CLKOUT | 0 | Clock output provides a system clock whose frequency is one half of the frequency of the X2-X1 inputs. |
| INT0, INT1 | I | Maskable interrupt requests. |
| INT2/ <u>INTAO</u> INT3/ <u>INTA1</u> | I/O | Maskable interrupt requests or interrupt acknowledge signals for requests INT0 and INT1. |

Figure 12-17 (continued)

| | | |
|-----------------------------------|---|--|
| ARDY | I | Asynchronous ready is used by memory or a device to acknowledge CPU's read or write request. This signal is internally synchronized. |
| SRDY | I | Synchronous ready. Can also be used as a ready signal but must be synchronized externally. If ARDY is used and SRDY is not, then SRDY should be grounded and vice versa. |
| TMR IN0, TMR IN1 | I | Timer inputs for clock or control signals. |
| TMR OUT0, TMR OUT1 | O | Timer outputs indicate terminal count or which max count register is used. |
| DRQ0, DRQ1 | I | DMA requests for two internal DMA channels. |
| <u>UCS</u> | O | Upper memory chip select. |
| <u>LCS</u> | O | Lower memory chip select. |
| <u>MCS3-MCS0</u> | O | Mid-range memory chip select. |
| <u>PCS4-PCS0</u> | O | Peripheral chip select signals. |
| <u>PCS5/A1,</u> <u>PCS6/A2</u> | O | Provides peripheral chip select signals or latched address signals A1 and A2. |

(b) Pin definitions

Figure 12-17 (continued)

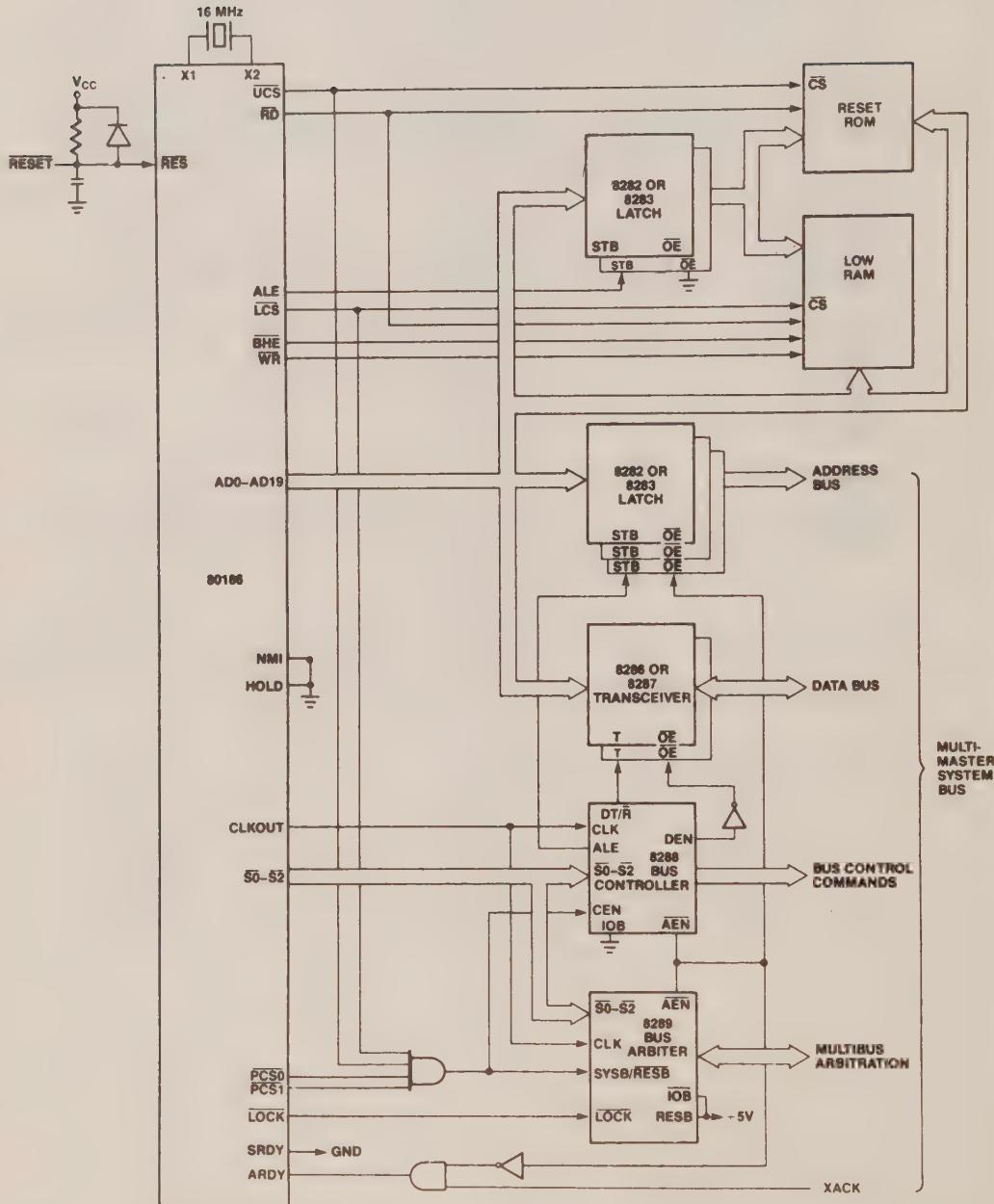


Figure 12-18 80186 in a multimaster system. (Reprinted by permission of Intel Corporation. Copyright 1982.)

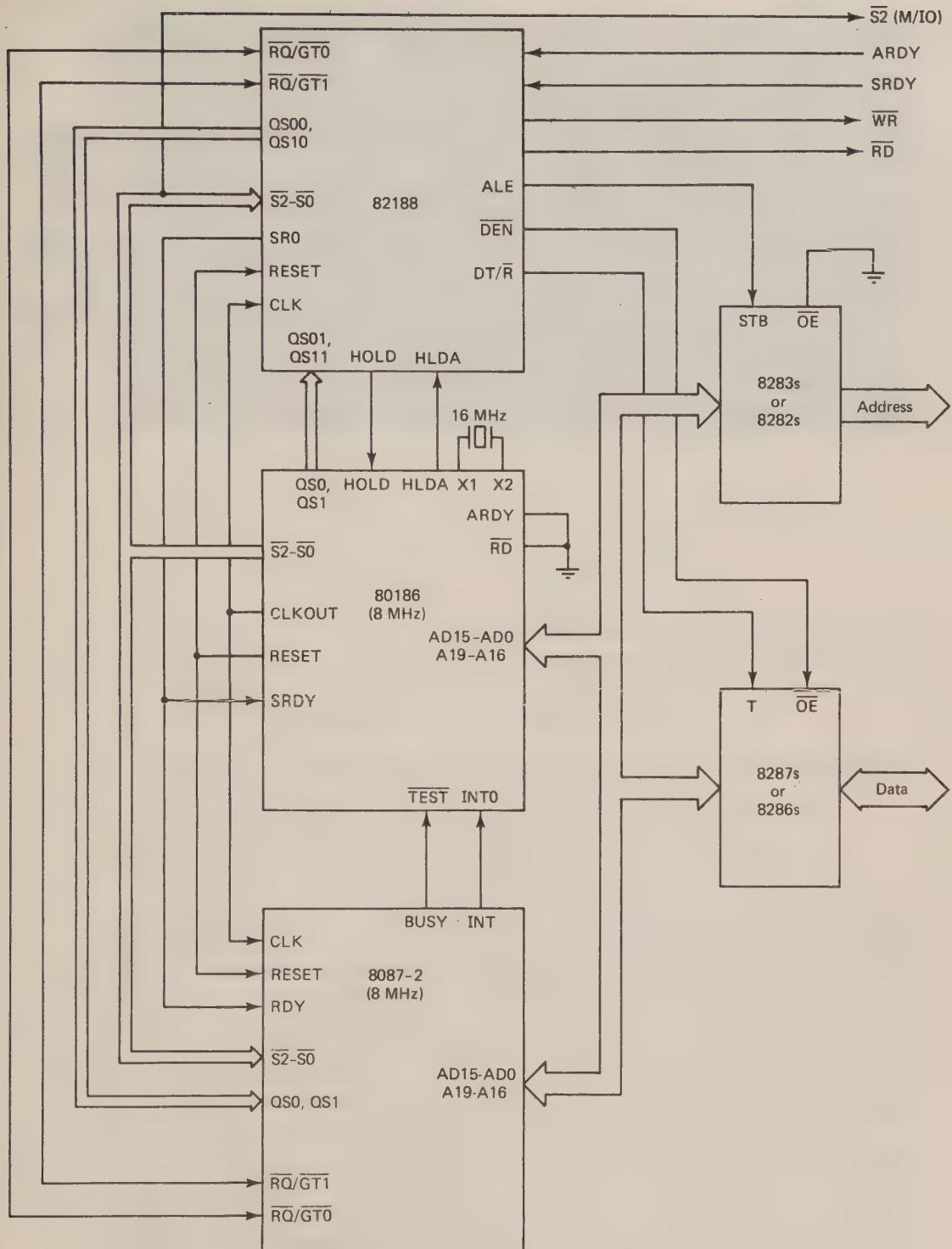


Figure 12-19 An 80186/8087 configuration.

BIBLIOGRAPHY

1. *iAPX 86/30, iAPX 88/30 Operating System Processors* (Santa Clara, Calif.: Intel Corporation, 1982).
2. *iAPX 86/88, 186/188 User's Manual* (Santa Clara, Calif.: Intel Corporation, 1983).

The 80286/80287

The enhancements to the 8086 that have been included in the 80286 are for improving the multiprogramming, or multitasking, capability of the 8086 family. This was primarily done by adding the necessary memory management and task switching logic in such a way that there is adequate protection:

1. Between the application tasks and the system and other more privileged tasks.
2. For separating the tasks from each other.
3. Between the code and data modules.
4. Against unwanted accessing or changing of either code or data.

The primary key to multitasking is the ability to break the various entities to be processed into parts in such a way that the parts can easily function together and adequate protection is maintained. Each task is designed to accomplish a well-defined job, but may incorporate several subtasks. Modularity is important in that it allows the programmer to view the entire system as a collection of modules that make up the tasks and subtasks. So that operating system modules that include critical code and data tables can be protected from applications programs, it also is necessary to be able to assign modules to different protection levels. A typical division of a system into tasks, levels, and modules is shown in Fig. 13-1.

A second key to multitasking is the ability to change quickly and smoothly from one task to another. At any given time, one of the tasks, the task containing the module that is currently executing, is called the *current task*. As time progresses, it is necessary to change the current task—i.e., to set aside logically the segments that comprise the current task and make the segments in another task the current

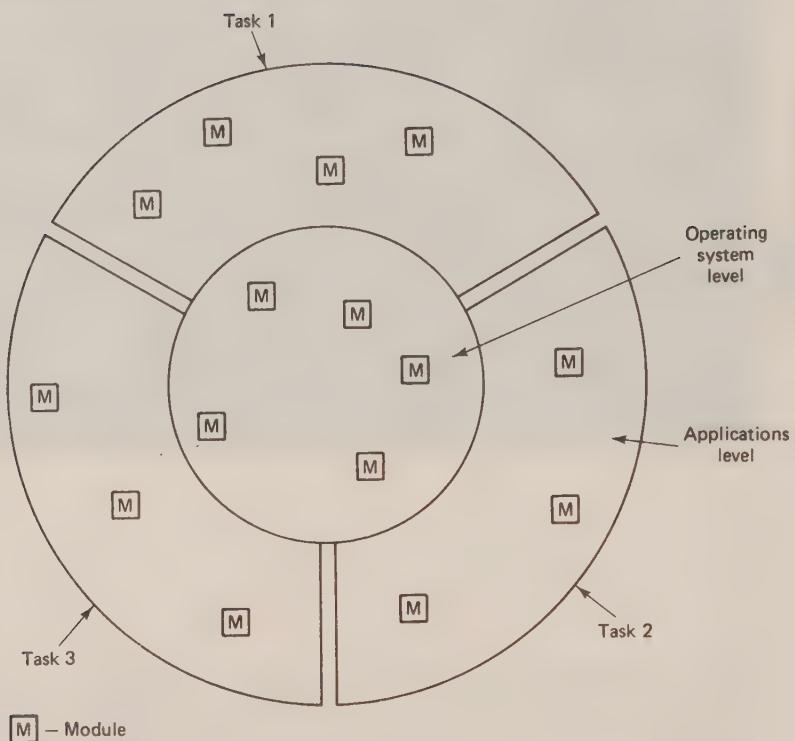
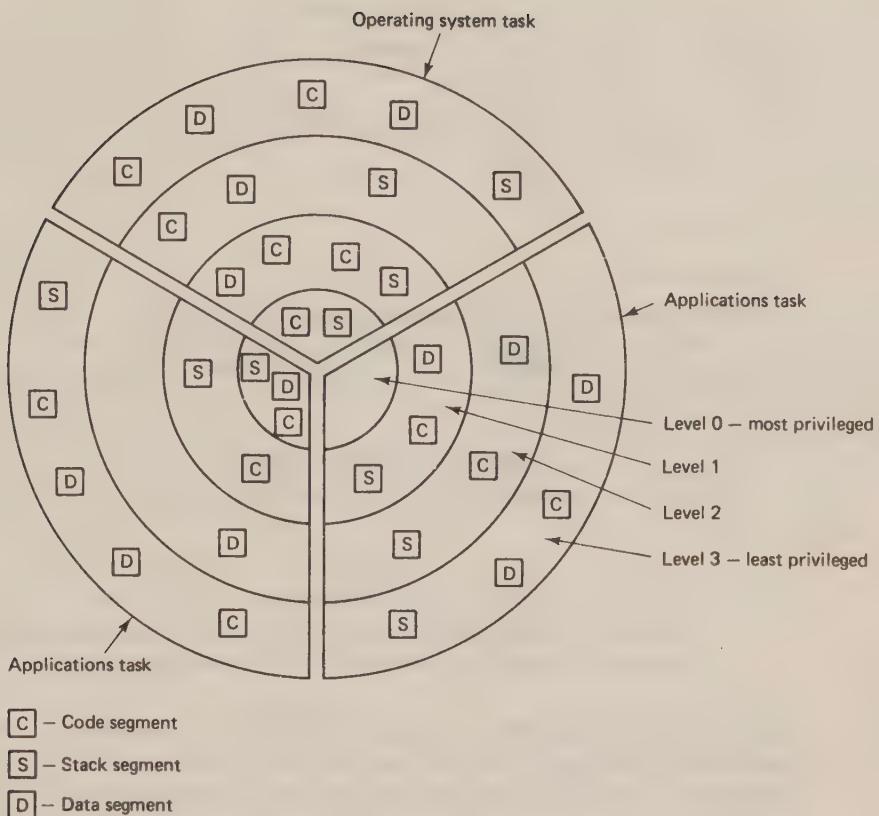


Figure 13-1 Breakdown of a system into tasks, levels, and modules.

task. This is called *task switching*, and a good multitasking system should provide a simple means of accomplishing such transitions.

The 80286 can be put into one of two modes, the real address mode or the protected virtual address mode (PVAM). When in the real address mode, the 80286 uses the same instructions as the 80186 and essentially operates as a fast 8086. Therefore, this mode will be of little interest to us. On the other hand, in its PVAM the 80286 has its multitasking capability.

In the 8086, Intel satisfied the need for modularity by allowing programs to be divided into segments and using the segment registers to bias references to these segments automatically. The 80286 PVAM extends this use of segments in such a way that the segments of several tasks can reside in memory at the same time and can be dynamically loaded as the need arises. In order to provide protection between segments within a task, segments are assigned to one of four privilege levels. Thus, the various tasks and segments are protected from unwanted accesses by type, segment, level, and task protection mechanisms. Figure 13-2 shows a typical set of currently active 80286 tasks and how they are broken down into levels and segments. Level 0 has the most privilege and level 3 the least privilege. Level 0 would normally include only the very critical modules in the operating system. Level 1 modules would consist chiefly of the less critical operating system services,



Note: There may be additional segments that may be shared by the tasks.

Figure 13-2 Division of an 80286 system into tasks, levels, and segments.

while level 3 would be for applications programs and level 2 most often would be used for custom extensions to these programs. In addition to the segments shown, there would be some segments, such as system tables and driver routines, that could be shared by some or all of the tasks. As with the 8086, the 80286 works with three basic types of segments, code segments, stack segments, and data segments. Notice the symbols used in Fig. 13-2 to represent these segments. These symbols will be used in other figures in this chapter.

The 80286 is a quad-in-line chip with seventeen pins on each side. The increased pin count is mainly due to the facts that there are 24 address pins instead of 20 (thus permitting over sixteen million bytes of memory) and the address and data signals are not multiplexed as they are on the 8086. Another major difference is the way in which the 80286 communicates with a coprocessor. The pin assignments and coprocessor arrangement will be discussed in detail in Secs. 13-6 and 13-7.

The internal architecture of the 80286 is shown in Fig. 13-3. Notice that it includes two queues, a code queue as well as the instruction queue, and a separate

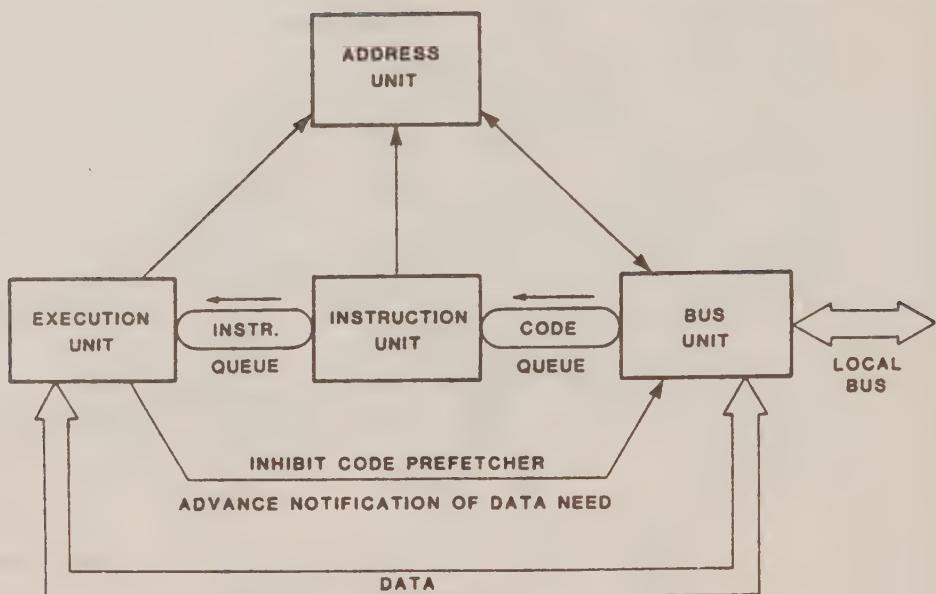


Figure 13-3 80286 internal architecture. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

set of logic for computing addresses. The code queue feeds into the instruction unit, which can supply the address unit with advance address information. The instruction queue feeds into the execution unit, which includes the CPU registers and ALU. The inclusion of a separate unit for calculating addresses and the ability to compute addresses in advance significantly improve the performance of the 80286 over that of the 8086, and are particularly useful when using the memory management features of the PVAM.

Figure 13-4 shows the registers in the 80286. The registers that are not shaded are the same as in the 8086 and are the ones that are used when operating in the real address mode. Except for bits 12, 13, and 14, the FLAGS register is the same as the 8086's PSW. Bit 0 of the machine status word determines which mode the 80286 is in and is discussed in Sec. 13-5. After a reset this bit is 0, indicating that the 80286 is in real address mode. If PVAM operation is desired, this bit is set during system initialization. Three new bits have been defined in the PSW. These bits and the other shaded registers are needed to implement the PVAM's multitasking capabilities and will be defined as they are introduced.

The 80286's instruction set is summarized in Fig. 13-5. There are sixteen new instructions that are in addition to the instructions for the 80186. These instructions are needed for the multitasking features of the 80286 and most of them are for loading and storing the extra registers. While in the PVAM, the intersegment branch instructions find extended usage and some of the 8086 and 80186 instructions have restrictions that are not applicable in real address mode.

The first three sections of this chapter start by considering single-task, single-

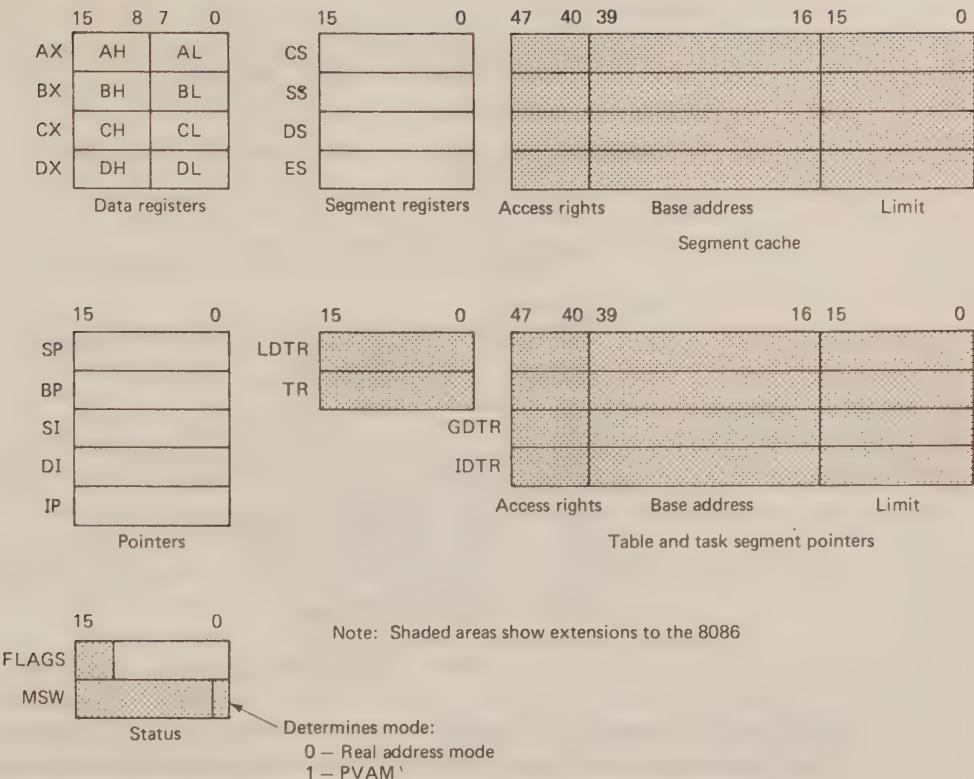


Figure 13-4 80286's register set.

level programs and work up to multiple task systems. Section 13-4 considers the interrupt system and Sec. 13-5 briefly discusses how a system of tasks can be built and how the system can be initialized for PVAM operation. The last two sections discuss the hardware aspects of the 80286—i.e., how it can be interfaced to a system bus and a coprocessor.

13-1 SINGLE-LEVEL TASKS

Let us begin with the simplest case, a task that occupies a single level. Although such tasks would access operating system services through interrupts and interlevel transfers, we will delay discussions of these possibilities until later. Since the 8086 does not have privilege levels, single-level tasks are most similar to 8086 programs, but there are important differences and these differences are our principal concern here.

As we have seen, in the 8086 memory is accessed by computing an effective address and adding it to sixteen times the contents of the appropriate segment register. Because an effective address is 16 bits wide, each segment is 64K bytes

8086/8088 Instructions

Instructions with restricted usage while in the PVAM:
HLT, IN, OUT, CLI, STI, and LOCK

80186/80188 Instructions

Instructions with restricted usage while in the PVAM:
INS and OUTS

80286 Instructions (for implementing the PVAM)

Existing instructions with extended usage:
CALL, JMP, RET, IRET, and software interrupts

New instructions:

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

Figure 13-5 80286 instruction set.

long with the contents of the segment register determining the base of the segment. The only limitation on base addresses of segments is that they be divisible by 16 and, therefore, segments can overlap. However, if two segments are overlapped there is no protection to prevent, for example, the data in one segment from accidentally being extended and overwriting code in the other segment.

For the 80286, each segment register has an extension, referred to as its *cache*. The definitions of the parts of a segment register are given in Fig. 13-6. As with the 8086, an address is computed by adding the effective address produced by the instruction to a base that is contained in the segment register. Unlike the 8086, the base is a full 24 bits in the segment register's cache and can, therefore, be any memory address. A data memory access is diagrammed in Fig. 13-7. Also 16 bits of the cache are reserved to specify the length, or *limit*, of the segment. Instead of all segments being 64K bytes long, they can have any length up to 64K bytes. If a limit is exceeded, a general protection interrupt will occur. (General protection interrupts as well as the other 80286 special interrupts are discussed in Sec. 13-4.)

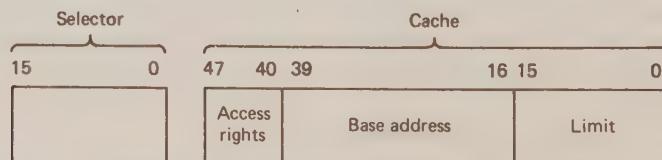


Figure 13-6 Components of a segment register.

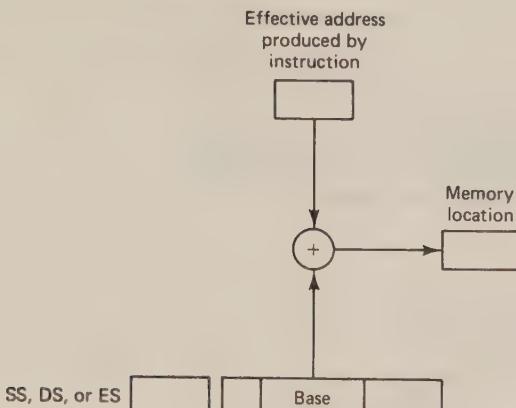


Figure 13-7 80286 memory reference.

For nonstack segments (CS, DS, or ES), a protection violation occurs when the effective address is greater than the limit. For a stack (SS), there is a violation when the contents of the SP become less than or equal to the limit. Also, when accessing a stack using the BP register or any other valid addressing mode, a violation will occur when the offset is less than or equal to the limit.

When a change is made from one segment to another, the cache is loaded from an 8-byte section of memory called a *segment descriptor*. Descriptors used to fill the CS cache are, clearly enough, referred to as *code segment descriptors* and those loaded into SS, DS, and ES caches are *data segment descriptors*. The breakdown of a descriptor is shown in Fig. 13-8(a). The access rights byte in the descriptor, defined in Fig. 13-8(b), is used to fill the access rights byte in the cache.

Each task is associated with a *local descriptor table* (LDT) that is made up of consecutive 8-byte descriptors. There also is a similar *global descriptor table* (GDT) that is used by all tasks. Each segment in a task is associated with a descriptor in one of these tables. In a dynamic environment, at any given time a segment may or may not be in memory and its presence is indicated by the P bit being 1. If P = 0 when an attempt is made to load a descriptor into a cache, a not present interrupt will occur. The resulting interrupt routine presumably would load the needed segment into memory and when the return is made the faulting instruction would be reexecuted. The privilege level of a segment is, by definition, determined by the descriptor privilege level (DPL) bits of the access rights byte. Bit 4 is 1 for segment descriptors. There are other types of descriptors and for them this bit is 0. They will be considered later. When bit 4 is 1, bit 3 is 0 for a data segment and 1 for a code segment. For a data segment, bit 2 indicates whether the segment is a nonstack or stack segment and bit 1 determines whether or not it can be written into. For a code segment, bit 2 defines its type (discussed later) and bit 1 determines whether or not it can only be executed or may also be read.

The access bit (bit 0), which is automatically set each time a descriptor is loaded into a segment register's cache, indicates whether or not the segment has been used since the last time this bit was cleared. Normally, all access bits are cleared periodically by the operating system, and then the operating system can

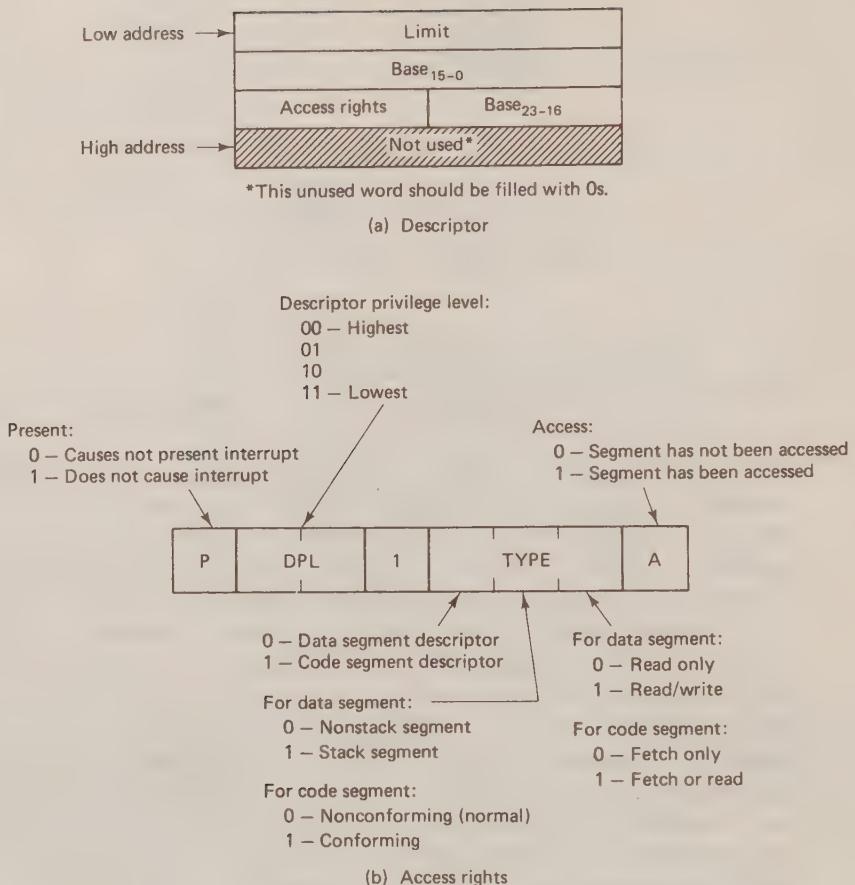


Figure 13-8 Definition of segment descriptors.

examine the access bits to determine which segments have been used recently. This information is important to swapping algorithms, which must decide which segments to swap out when more memory space is required by the current task. Also, this bit can be used to determine whether or not a segment descriptor has been accessed since it was last written to mass storage. If it has not, there is no reason to rewrite the segment to mass storage when it is swapped out. It can simply be overwritten by the segment that is being swapped in.

Loading a descriptor into a cache cannot be done directly. It is done automatically when the selector part of the segment register is changed by an instruction in the currently executing program. Figure 13-9 shows the sequence of events that results when the selector in SS, DS, or ES is changed by a data transfer instruction. As shown in Fig. 13-10, the selector consists of three parts. The index part is used to "select" a descriptor from the current task's LDT or from the GDT. If the table index (TI) bit is 1, the address of the descriptor is calculated by adding eight times the index to the base of the current LDT. [This base was put into the 80286's local

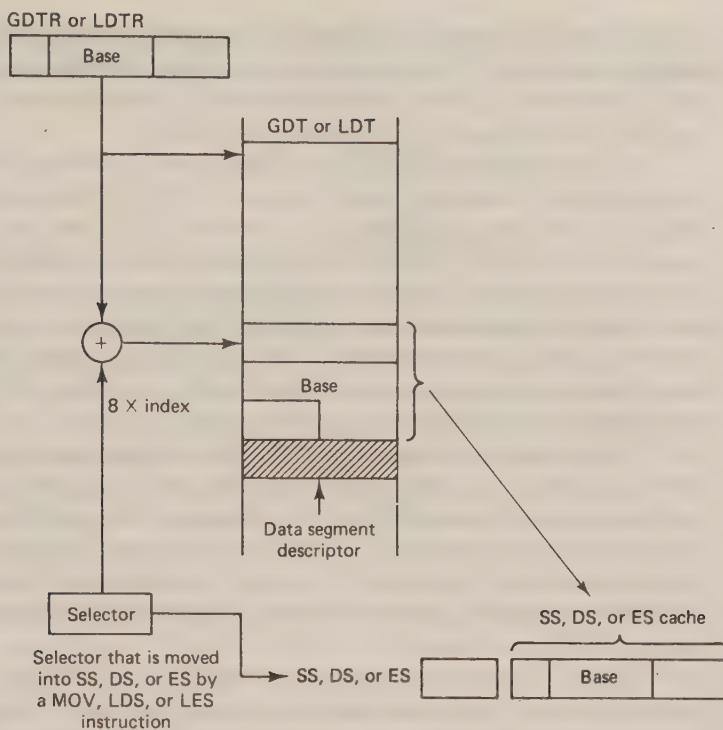


Figure 13-9 Loading the SS, DS, or ES cache.

descriptor table register (LDTR) when the switch was made to the current task. Task switching is discussed in Sec. 13-3.] Changing the selector of a segment register automatically causes the register's cache to be loaded from the new segment's descriptor in the LDT. Any subsequent data references that use the segment register will use the newly loaded base in the cache.

For example, consider the instructions

```

MOV AX,DSEG
MOV DS,AX
.
.
.
MOV CX,X2

```

where DSEG is the name of a segment and X2 is a variable name within that segment. The first instruction causes a selector whose index points to DSEG's

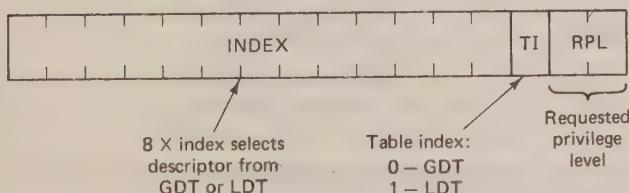


Figure 13-10 Definition of a selector.

descriptor to be put in AX. Then the second instruction loads the selector into DS and this causes the DS register's cache to be loaded from this descriptor. The latter MOV instruction uses the displacement of X2 and the new contents of the cache to bring the contents of X2 into the CX register.

We have seen that descriptors also can be put in the GDT. The GDT's base is in the global descriptor table register (GDTR). If the selector's TI bit is 0, the base is taken from the GDTR instead of the LDTR and the descriptor is selected from the GDT. The GDTR is not changed by task switches, but is filled only while the system is being initialized. This means that the descriptors in the GDT are available to all tasks and provide a common pool of code and data. The set of all addresses that are accessible by a task is called its *virtual address space*. For each task, this set is determined by the descriptors in the GDT in addition to the descriptors in its LDT. This is illustrated by Fig. 13-11.

The requested privilege level (RPL) bits in the selector are used for protection purposes. From a hardware standpoint the RPL of a DS or ES selector must be less than or equal to the DPL of the descriptor being selected. For an SS selector, the two must be equal. In fact, Intel's linking and building software always sets the corresponding RPLs and DPLs equal to each other. Due to this, the RPL will be of little interest to us except when it is changed by an ARPL instruction (discussed later).

The CS register's selector may be changed by a JMP, CALL, RET, or IRET instruction, and is always reloaded by an interrupt of any type. Since SHORT and NEAR branches are to points within the current segment, they modify only the

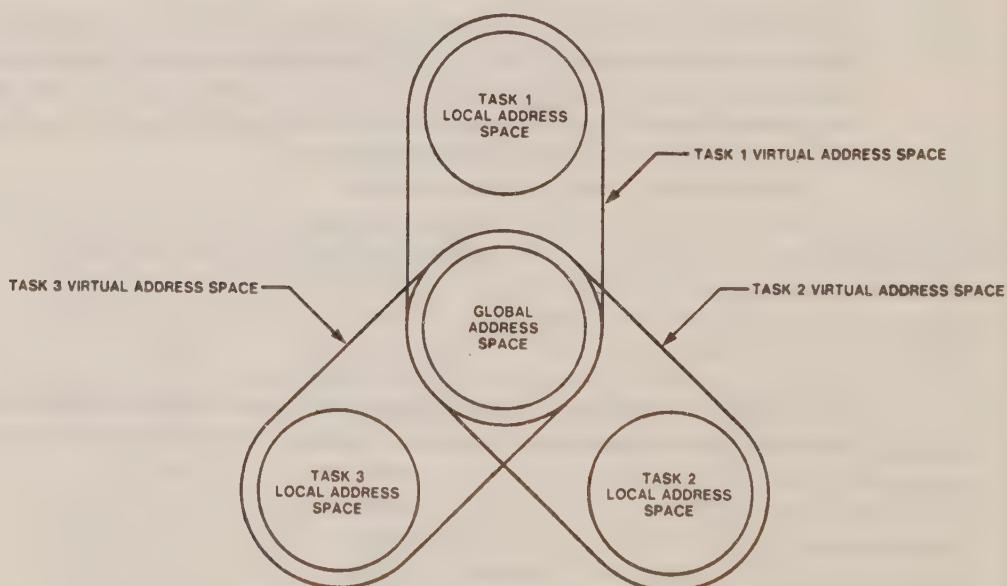


Figure 13-11 Relationship among the virtual address spaces. (Reprinted by permission of Intel Corporation. Copyright 1984. Courtesy of Intel Corporation.)

offset and are therefore similar to such branches in an 8086 program (except that the base is a 24-bit address taken from the CS cache). On the other hand, the branches

JMP FAR PTR *Label*

and

CALL FAR PTR *Label*

do replace the CS register's selector with the last two bytes of the instruction. Therefore, they cause the CS register's cache to be filled and the next instruction to be selected as shown in Fig. 13-12(a). In addition, a CALL will push the contents of the CS's selector and the IP (i.e., the return address) onto the stack.

A return from a FAR procedure due to a RET instruction will cause the IP and CS's selector to be filled from the stack. Thus, the contents of the CS's selector will become the same as it was just prior to the call. The restoration of the original selector will cause the calling segment's descriptor to be put in the CS cache, and the calling program will pick up where it left off. The return sequence is shown in Fig. 13-12(b).

The *current privilege level* (CPL) is the privilege level of the currently executing code segment. How the CPL is set depends on the type, *nonconforming* or *conforming*, of this segment. This type is determined by bit 2 of the code segment's descriptor at the time the segment is branched to. For a nonconforming segment (bit 2 = 0), the CPL will be the same as the DPL of the segment's descriptor. Also, for a

JMP FAR PTR *Label*

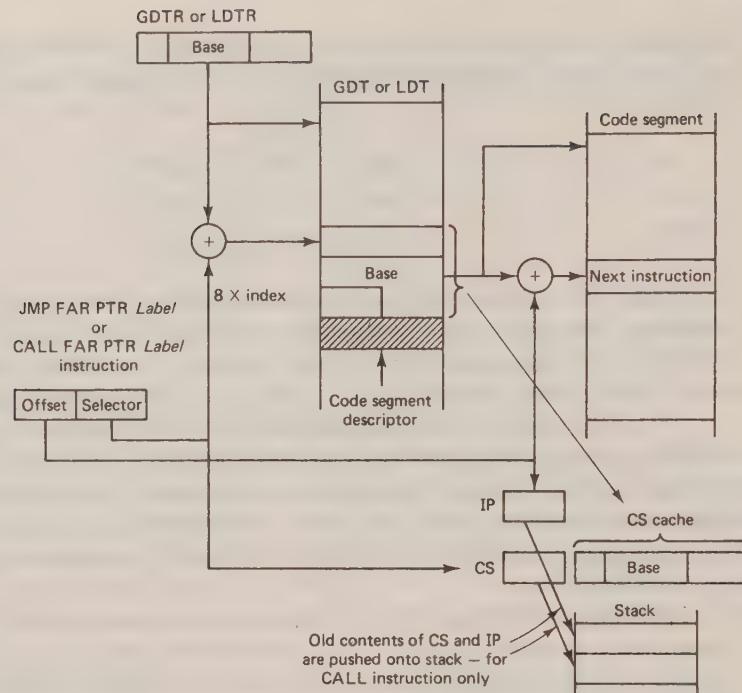
or

CALL FAR PTR *Label*

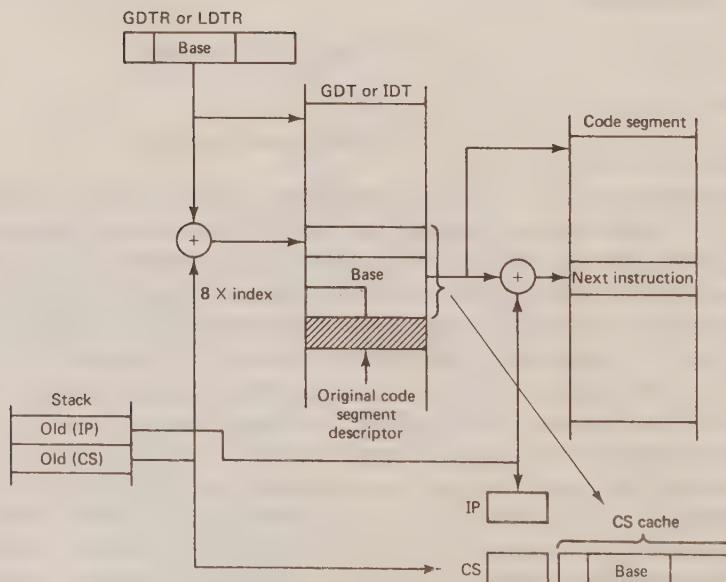
branch, the RPL from the instruction's selector must be less than or equal to the CPL at the time of the call, and the DPL in the called segment's descriptor must be the same as this CPL. For a conforming segment, the new CPL will be the same as the old CPL regardless of the DPL field in the called segment's descriptor; however, for the foregoing FAR branches this DPL must be less than or equal to the CPL immediately prior to the branch—i.e., the old CPL.

In either the nonconforming or conforming case, the net result is that the old CPL and the new CPL will be the same. This means that for the above FAR branches, there will be no level change. Clearly, a return from a call that stays within the same level will be to the same level. Also, the new CPL will be put in the RPL field of the CS register's selector, not the RPL portion of the selector in the instruction. Actually, as with data descriptor selectors, Intel's linking and building software always sets the RPL's instruction operands equal to the DPL of the descriptor, and these RPLs are not important to the programmer.

In addition, when changing data segments certain privilege level requirements related to the CPL must be met. Whenever the SS register's selector is changed,



(a) Determination of next instruction



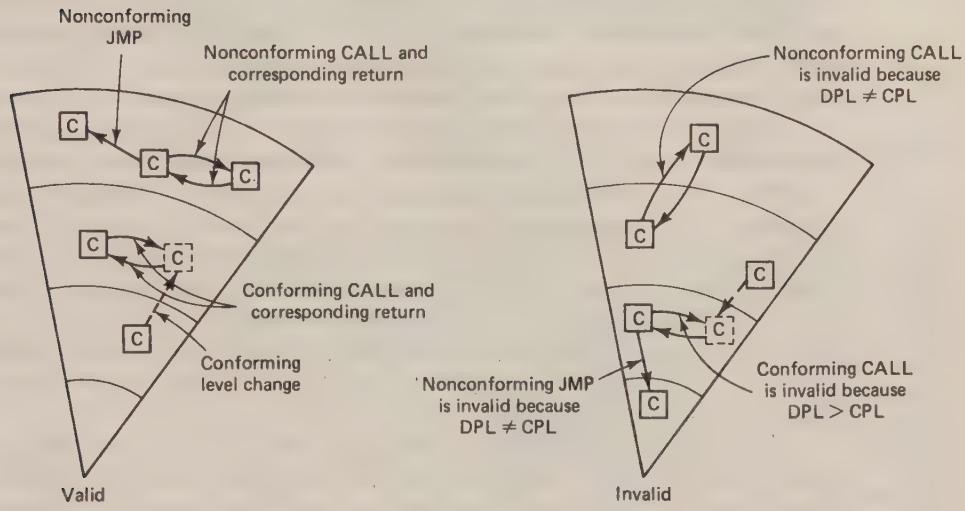
(b) Obtaining the return address

Figure 13-12 Determination of the next instruction after `JMP FAR` and `CALL FAR` instructions and a `RET` instruction after `FAR CALL`.

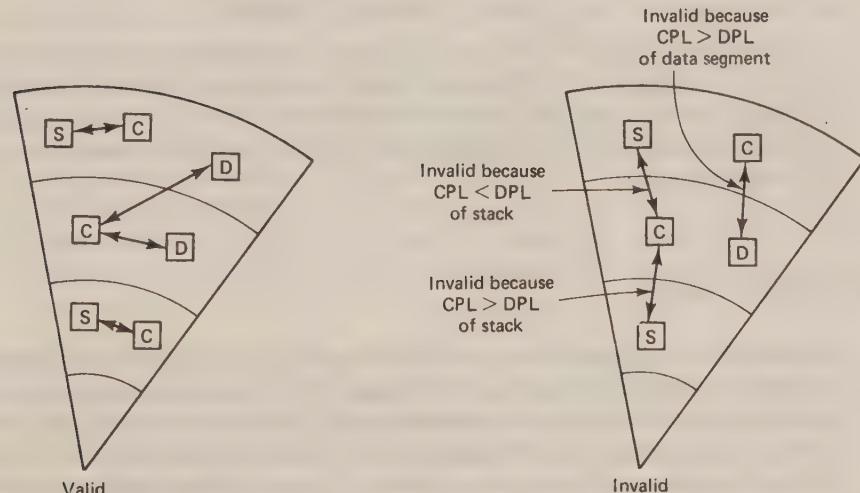
the RPL of the new selector, the CPL, and the DPL of the new segment must all be equal. For a change in the DS or ES selector, the DPL of the new segment must be greater than or equal to the maximum of the CPL and the RPL of the new selector. This maximum is called the *effective privilege level (EPL)*.

Figure 13-13(a) shows some valid and invalid transfers caused by

JMP FAR PTR Label



(a) JMPs and CALLs



(b) Data segment changes

Figure 13-13 Examples of valid and invalid segment changes.

and

CALL FAR PTR Label

instructions, and Fig. 13-13(b) shows some valid and invalid data segment changes. If any one of the above privilege level requirements is not met at the time of a segment change, a general protection interrupt will occur.

The reason for having privilege levels is to limit the access of code at the less privileged levels to the resources at the more privileged levels. By restricting stack references to the same privilege level as the code, accidental corruption of the stacks is prevented. By not allowing code to reference data at a more privileged level, an applications task is prevented from changing entries in systems tables.

The purpose of restricting normal branches to the same level is to prevent indiscriminate branching between levels, particularly from the less privileged levels to levels 0 and 1. Obviously, it is sometimes necessary to branch from one level to another, but this requires the special forms of the JMP and CALL instructions that are discussed in the next two sections. Although a transfer to a conforming segment does not change the CPL, it does allow the called segment to take on the level of the calling segment. This is useful for some routines, such as I/O drivers, that must be used by programs at all levels, but should not raise the privilege level and, thereby, permit these routines to include data references to levels more privileged than the calling program's CPL.

By the use of *Label* in the foregoing JMP and CALL instructions it was implied that the offsets and selectors are contained in the instructions (i.e., the branches are direct). Indirect branches for which the offsets and selectors are in doublewords in memory also can be made, provided that the doubleword is accessible from the current level. For example,

CALL DWORD PTR [BX] [DI]

would cause the offset to be obtained from the location pointed to by (BX) + (DI) and the selector to be obtained from this location plus two. These comments also apply to the branches discussed below that use *Gatename* and *Taskname* operands, even though the examples assume the offsets and selectors are included in the instructions.

13-2 MULTILEVEL TASKS

We have seen that an ordinary call that is made using a selector that points directly to a code segment descriptor cannot change the CPL and, hence, can branch only to a code segment at the same privilege level. Yet there must be a way of changing privilege levels while maintaining protection against unwanted level switches. This is done by forcing all calls that alter the CPL to be made indirectly through descriptors referred to as *call gates*. The format of a call gate and the definition of its access rights byte are given in Fig. 13-14.

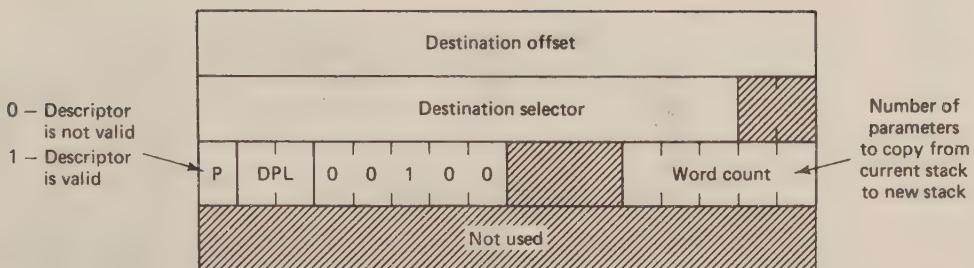


Figure 13-14 Definition of a call gate descriptor.

The form of an assembler instruction that makes a call through a call gate is:

CALL Gatename

where *Gatename* is any legal identifier or pointer to a memory doubleword. In either case, the offset portion of the instruction or doubleword is ignored and the selector portion points to the call gate to be used. The offset needed to complete the call is taken from the first word in the descriptor. The entire sequence that determines the address of the next instruction is diagrammed in Fig. 13-15. From this figure, it is seen that the selector in the instruction points to the descriptor (which may be in the GDT or current LDT depending on the TI bit in the selector), the selector in the call gate descriptor points to a code segment descriptor (which may also be in the GDT or current LDT), and the base in the code segment descriptor is added to the offset in the call gate to produce the address of the next instruction.

The privilege requirements enforced by a call through a gate are that the gate be of equal or lesser privilege (DPL of gate \geq CPL) and the called code segment be of equal or greater privilege (DPL of code segment descriptor \leq CPL). While the former places an upper limit on the level from which the call is made, the latter places a lower limit on this level. Figure 13-16 shows that only calls from levels 1 and 2 can be made to a code segment at level 1 through a call gate at level 2. Some valid and invalid calls are illustrated in Fig. 13-17. Also, the gate DPL must be numerically greater than or equal to the CALL instruction selector's RPL, but, as explained earlier, this is of little consequence because the selector's RPL is normally automatically set equal to the called segment's DPL. It also is possible to issue a

JMP Gatename

instruction, but it cannot branch to a different level. Although the gate's DPL can be numerically greater than the CPL, the branch must be made to a code segment for which the DPL is equal to the CPL.

The 80286's PVAM design philosophy is predicated on the assumptions that each task will have a stack at each level which has a code segment that is used by the task, each code segment will use the stack at its level, and parameters will be passed between code segments via the stacks. This guarantees that the stack ref-

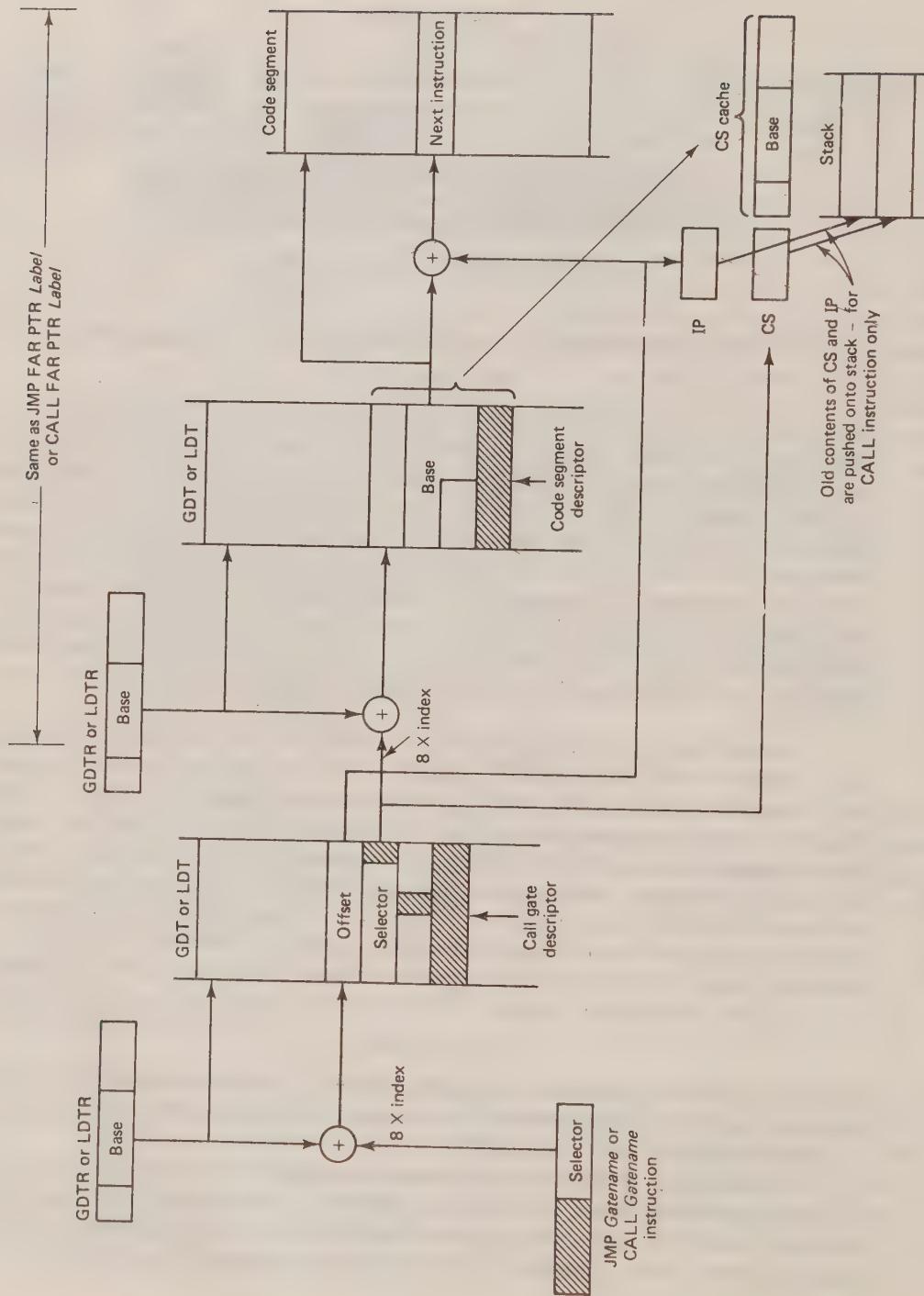


Figure 13-15 Determination of the next instruction after a **CALL Gatename** or **JMP Gatename** with no task switch.

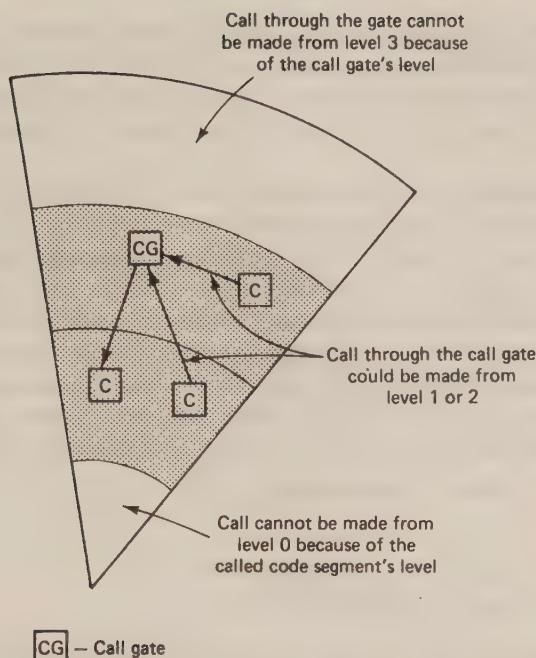


Figure 13-16 Accessibility of a code segment as a function of its gate.

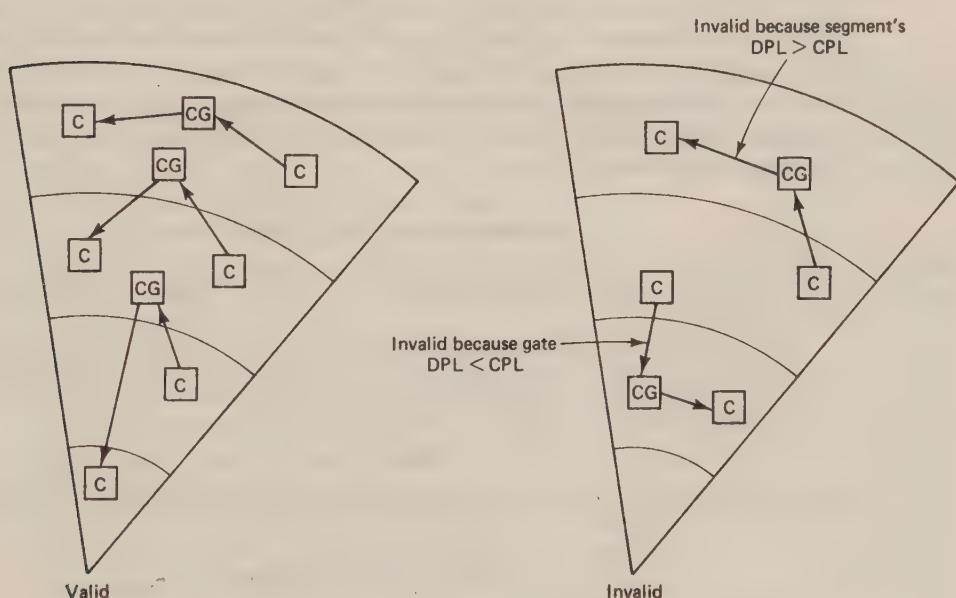


Figure 13-17 Examples of valid and invalid calls through gates.

erencing will be relatively straightforward and parameter passing will be consistent and permit the use of the ENTER and LEAVE instructions. To realize these assumptions, the stack is automatically switched whenever there is a level change and the parameters that are pushed onto the stack before the call are automatically copied to the new stack.

In Sec. 13-3, we will find that each task is associated with a task state segment, and for each of the task's level 0, 1, and 2 stacks there is a selector in this segment that points to it. Also, for each of these stacks, the task state segment contains a value for SP. What a stack switch entails is retrieving the selector for the stack at the level of the called code segment (the new stack) from the task state segment and putting it into SS, retrieving the corresponding value for SP, and then pushing the old SS and SP onto the new stack. (An SS and SP for level 3 are not needed because the level protection is such that an interlevel call can never be made to level 3.) The stacks for both the calling level and called level after the sequence

PUSH *1st parameter*

PUSH *nth parameter*
CALL *Gatename*

is executed are shown in Fig. 13-18. The last item pushed onto the stack is the return address—i.e., the contents of CS and IP.

The number of parameters that are copied is determined by the word count (WC) field in the call gate descriptor (see Fig. 13-14) and should equal the number of parameters pushed just before the call. This field is set to a specific value at the time the gate is defined. (Gate and other specifications are made when the task is built.) In addition to the number of parameters being included in the call gate, the segment being called must have a word-count specification in its opening procedure statement as follows:

*Procedure name PROC FAR WC(*n*)*

where *n* is the number of parameters. If the two counts do not agree, a linking error will occur.

If a call is made through a call gate but there is no level change, there is no stack switch, no parameters are copied, and the WC field in the gate is ignored. The only thing pushed onto the stack by the

CALL *Gatename*

instruction is the return address. In other words, the stack activity is the same as for a

CALL FAR PTR *Label*

instruction.

There is no special return instruction for returning to a different level. All returns are made using the RET instruction. For interlevel returns it is common

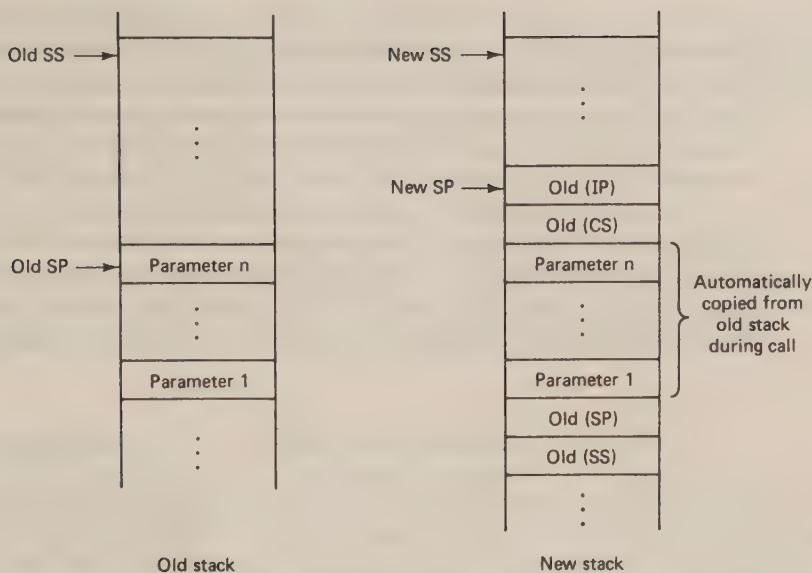


Figure 13-18 Appearance of the calling and called stack after a call to a different level.

to use

RET *m*

where *m* is twice the number of parameters (words) pushed onto the calling segment's stack just before the call. This causes the parameters, which are no longer needed, to be removed from this stack. Notice that the called segment's stack does not need to be adjusted because it is reinitialized each time it is called by reloading the SS and SP from the task state segment.

All returns must be such that the RPL of the selector popped from the stack is greater than or equal to the CPL. When returning to a nonconforming segment, the segment's DPL must be equal to the return selector's RPL, but for a conforming segment its DPL may be less than this RPL. In general, what is important to the programmer is that calls through gates must be to the same level or inward to higher privilege levels, and the corresponding returns must be to the same level or outward to lower privilege levels. The nonstack data segment protection rules are the same regardless of whether the CPL is changed. For nonstack data, the CPL at the time of the data reference is always used. If there is any type of privilege level violation, a general protection interrupt will occur, and if a call or return causes any stack to go outside its limits, a stack segment interrupt will take place.

Despite all of the checking that is done, there is one subtle way a level violation can go undetected. To demonstrate how this can happen, suppose that a level 3 code segment pushes a selector that points to a level 2 data segment onto the stack and then calls a level 1 code segment. The level 1 code segment could then access

the level 2 data segment without a protection fault. But this is usually not desirable, because when the level 1 code accesses a parameter it normally assumes it is working with data that is accessible to level 3 and treats it as such. To guard against this, the 80286 instruction set has included the adjust requested privilege level (ARPL) instruction. ARPL is a valid instruction only while operating in the PVAM and is formally defined, along with four other strictly PVAM instructions, in Fig. 13-19.

The ARPL instruction causes the number represented by bits 1 and 0 of its first operand to be replaced with the maximum of the numbers in bits 1 and 0 of its two operands. Also it sets ZF to 1 if the bits in the first operand are changed and, otherwise, clears ZF. Before ARPL is used it is assumed that the selector for the data segment in question is put in the first operand and the CPL of the calling

| Name | Mnemonic and Format | Description |
|--------------------|---------------------|---|
| Adjust RPL | ARPL DST,REG | Compares the lower 2 bits of the two operands and replaces the lower 2 bits of DST with the maximum. ZF ← 1 if lower bits of DST change; ZF ← 0 otherwise. |
| Load access rights | LAR REG,SRC | If descriptor pointed to by (SRC) can be accessed, then (REG) ← Access rights of descriptor pointed to by (SRC) (REG) ← 0. ZF ← 1 if loading is performed; ZF ← 0 otherwise. |
| Load segment limit | LSL REG,SRC | Same as LAR, except REG is loaded with the first word of the descriptor. |
| Verify read | VERR SRC | If descriptor pointed to by (SRC) is accessible from the current level and is a readable code descriptor then ZF ← 1; otherwise ZF ← 0. |
| Verify write | VERW SRC | Same as VERR except ZF ← 1 if descriptor is for a writable data segment. |

Notes: When in the real address mode, these instructions cause undefined opcode interrupts. These instructions will not cause an interrupt due to level protection.

Flags: Only ZF is affected.

Addressing modes: All operands are words. SRC and DST can use any addressing modes except immediate. REG can be AX, CX, DX, BX, SP, BP, SI, or DI.

Figure 13-19 PVAM adjustment and checking instructions.

program is in the second operand. After ARPL is executed, the data segment selector is normally restored to its original memory location or is put in DS or ES.

For example, consider the following portions of a level 0 procedure for which the passed parameter is assumed to be a data segment selector:

| | | | | |
|----------|------|-------------|-------------------------------|--|
| LVL0PROC | PROC | FAR | WC(1) | |
| | PUSH | BP | | |
| | MOV | BP,SP | | |
| | MOV | AX,[BP + 4] | ;LOADS THE CALLING CS AND | |
| | ARPL | [BP + 6],AX | ;ADJUSTS PASSED SELECTOR | |
| | MOV | DS,[BP + 6] | ;GENERAL PROTECTION INTERRUPT | |
| | | | ;OCCURS IF SEGMENT'S DPL < | |
| | | | ;SELECTOR'S RPL | |
| | POP | BP | | |
| | RET | 2 | | |
| LVL0PROC | ENDP | | | |

The third instruction puts the calling program's CS selector into AX. The fourth instruction compares the CPL field of this selector with the RPL field of the passed selector and replaces the lower two bits of this field with the maximum of the two values. Then the fifth instruction attempts to put the (possibly modified) data selector in DS.

Now, suppose that LVL0PROC is called from level 3 and passes the selector for LVL2DSEG, a level 2 data segment. The ARPL instruction will replace the 2 in the RPL of LVL2DSEG's selector with 3 before an attempt is made to put the selector in DS. When the fifth instruction is executed, a general protection interrupt will be initiated because the RPL being loaded into DS will be greater than the DPL in LVL2DSEG's descriptor (which was not changed by the ARPL instruction). An alternative to using an interrupt routine to process protection errors is to put a branch on zero (JZ) instruction just after the ARPL instruction and use a sequence in LVL0PROC to take the required actions in the event that the RPL is changed.

On the other hand, if LVL0PROC is called from any level other than 3, ARPL will not alter the RPL field of the selector, the RPL will remain the same as the DPL for LVL2DSEG, and there will be no protection violation.

It should be evident that the ARPL adjustment is not always needed. A level 3 code segment would never need to perform this type of adjustment. Also, a special-purpose code segment that is always called by a limited set of known code segments may not have to include ARPL instructions. However, if the selectors being passed are not *always* known to be of the proper level, ARPL instructions should precede transfers into the DS and ES registers.

The other four PVAM only instructions are for advance checking of access rights bits and limits so that protection and other access related interrupts will not be given a chance to occur. If possible, LAR loads the access rights of the descriptor, whose selector is designated by the second operand, into the register designated by the first operand. If, for any reason, the descriptor cannot be loaded (e.g., the

selector's index is too large and points outside the descriptor table's limits), the ZF flag is cleared; otherwise it is set. An attempt to load an invalid descriptor would not cause an interrupt. The instructions

| | |
|--|---------|
| LAR | AX,DSEG |
| JNZ | LOADERR |
| : } Instructions for examining access rights | |

would try to load the access rights of the descriptor pointed to by DSEG into AX and would branch to LOADERR if the load failed. After the access rights have been loaded they could be examined for potential problems (e.g., the P bit being 0) before an attempt is made to use the DSEG descriptor. LSL is similarly needed for inputting and examining the limit portion of a descriptor.

VERR and VERW are for verifying that a code segment is readable or a data segment is writable, respectively. This is done by checking bit 1 of the descriptor pointed to by the selector in the instruction's operand, and setting ZF to 1 if this bit is 1 and the descriptor is accessible from the CPL. Otherwise, ZF is cleared. Therefore, the sequence

| | |
|-----------------------------------|---------|
| MOV | AX,CSEG |
| VERR | AX |
| JNZ | READERR |
| : } Instructions for reading CSEG | |

would inspect CSEG's descriptor to see if an interrupt would take place if an attempt is made to load the CSEG selector into a segment register or to subsequently read from the segment. If the selector is not a valid selector for a segment that is currently readable, a branch would be taken to READERR.

13-3 MULTIPLE TASKS

The preceding discussion has studied data referencing and branching among segments within a single task, but the 80286's capabilities are not fully utilized until it is placed in a multitasking environment. It is now time to consider how several tasks can work together and, in particular, how task switching is done.

Associated with each task is a *task state segment (TSS)*, which is used to store the current state of the task, and an LDT for referencing the local portion of the task's virtual address space. Although a TSS may contain other task related data (such as a coprocessor's state), its first 44 bytes must be used to store the information shown in Fig. 13-20. Assuming that tasks operate for a while and then are suspended while other tasks use the CPU, it is this information that is needed during a return switch to the task in order for the task to pick up where it left off. The contents of all of the CPU registers would be automatically stored in the TSS immediately

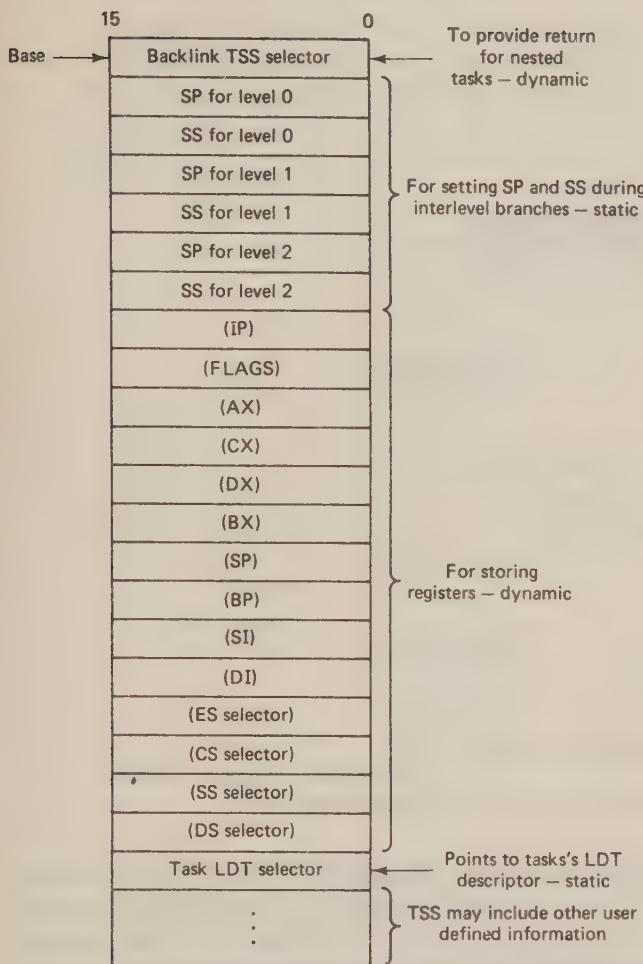
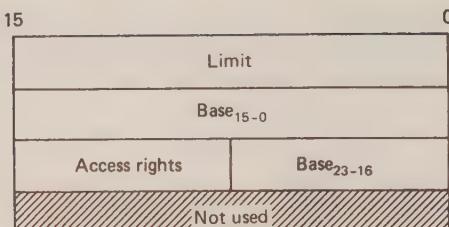


Figure 13-20 Definition of a task state segment.

before the task is suspended and then automatically restored when the task is reentered. The portion that holds the register contents is considered to be dynamic since they are automatically changed with each switch from the task. Only the selector portions of the segment registers need to be saved because the selectors can be used to reload the cache portions from the LDT of the task when the task is returned to.

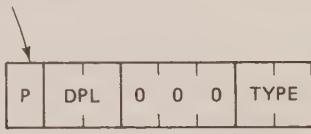
It was seen earlier that the SS and SP pointers for the different levels are required for changing stacks during interlevel calls. This part of the TSS is initialized when the task first enters the system and is static as long as the task remains in the system.

For each task's TSS and LDT, there must be system segment descriptors *in the GDT* that have the form shown in Fig. 13-21. A task's TSS and LDT descriptors indicate the base, length, level, and validity of the task's TSS and LDT, respectively. A task's TSS is pointed to by its TSS descriptor and this descriptor is selected by



(a) Descriptor

0 — Descriptor is not valid
1 — Descriptor is valid



Type:
00 — Unused
01 — Available TSS
10 — LDT descriptor
11 — Busy TSS

*The use of the P bit is left to the operating system designer.

(b) Access rights

Figure 13-21 Definition of system segment descriptors.

the selector causing the task switch. Its LDT descriptor is used to load the cache portion of the LDTR when a switch is made to the task. During the switch, the selector in the twenty-second word shown in the TSS definition is used to select the task's LDT descriptor from the GDT. The selector is then put in the selector portion of the LDTR. This activity is illustrated in Fig. 13-22.

The P bit could be used to indicate the presence (in memory) of the TSS or LDT, but since some operating systems are designed so that all of the TSS and/or LDT's are always present, this bit could be used for other validity checking. Regardless of what it is used for, if the descriptor is referenced while $P = 0$ then a not present interrupt will occur. As usual, the DPL gives the privilege level. For a TSS descriptor, the level is of the TSS, not of the task. A task is not assigned a level as a whole, but its segments are assigned levels through their descriptors. The significance of the level assigned to a TSS is discussed later. The DPL fields in both TSS and LDT descriptors normally are filled with zeros, thus giving them the most privilege possible.

TSS and LDT descriptors are differentiated by bit 0 of the access rights byte, which is 1 for a TSS descriptor and 0 for an LDT descriptor. Bit 1 of this byte is always 1 for an LDT descriptor, but for a TSS descriptor it is 0 if the corresponding task is not busy and is 1 if it is busy. This bit is always set when the task is entered

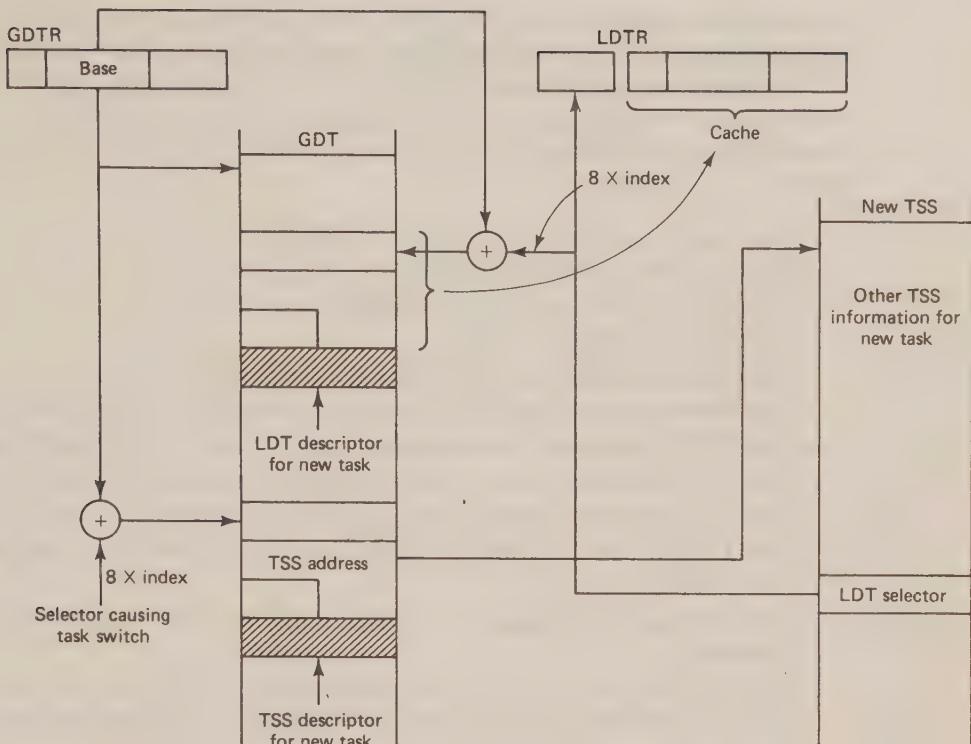


Figure 13-22 Selecting a TSS and loading the LDTR during a task switch.

and, except when exiting to a nested task, is cleared when it is exited. For nested tasks, which are considered later, the busy bit is cleared only when the nesting is complete and a switch is made to an unnested task. If a switch to a busy task is attempted, a general protection interrupt will occur.

A task switch can be initiated by a `JMP` or `CALL` instruction or by an interrupt. Interrupts are covered in Sec. 13-4 and a discussion of task switches due to interrupts is delayed until then. The difference between task switches resulting from `JMP` and `CALL` instructions is that `JMP` instructions cause unnested switches while `CALL` instructions cause nested switches. A task switch can be made directly by using

`JMP Taskname`

or

`CALL Taskname`

or indirectly by using

`JMP Gatename`

or

`CALL Gatename`

Taskname and *Gatename* are either valid identifiers or point to double-word memory operands. In either case, only the selector is used and the offset is ignored.

A direct switch is illustrated in Fig. 13-23. Note that a task switch involves a new register, the task register (TR). This register is for storing both the task's TSS descriptor and the selector that points to it. A direct task switch is summarized as follows:

1. The current contents of the segment register selectors and the working registers are put in the current TSS.
2. A TSS descriptor is chosen by adding 8 times the selector's index to the base portion of the GDTR.
3. This TSS descriptor points to the TSS of the new task and, for a nested switch, the selector part of the TR is put in the "backlink" word of the new TSS.
4. The TR is filled with the instruction's selector and the new task's descriptor.
5. An LDT descriptor is chosen from the GDT by adding 8 times the index of the "LDT selector" word in the new TSS to the base part of the GDTR.
6. The LDTR is loaded from this LDT descriptor and the "LDT selector" word in the TSS.
7. The "IP" and "CS selector" words are put in the IP and CS registers, respectively.
8. The "CS selector" word and the base part of the GDTR or the new contents of the LDTR are used to select a code segment descriptor from the GDT or new LDT.
9. The code segment descriptor is put in the CS's cache.
10. The base portion of the CS cache and the IP are added to produce the address of the next instruction and the registers are filled using the new TSS.
11. The task being switched to begins execution.

Unwarranted switches are protected against by requiring the RPL of the JMP or CALL instruction's selector and the CPL at the time the switch is initiated to be at least as privileged as the TSS descriptor's DPL. Normally, direct switches are conducted only by the operating system and this DPL is assigned level 0.

Task switches can be initiated by code segments with less privilege than the TSS descriptor's DPL by using JMP's and CALL's through task gate descriptors, which can be located in either the GDT or the current task's LDT. The format of such a descriptor is given in Fig. 13-24. Except for having to first go to the task gate to get the selector that points to the desired TSS descriptor, an indirect switch is the same as a direct switch and is shown in Fig. 13-25. For an indirect switch, the only protection is that the DPL of the task gate must be of equal or lesser privilege than either the RPL of the selector or the CPL. Several valid and invalid task switches are depicted in Fig. 13-26.

Neither the task initiating the switch nor the TSS descriptor exerts any restriction on the level of the code segment selected in the new task. The level of this code segment must obey only the rules of any code segment to which a branch

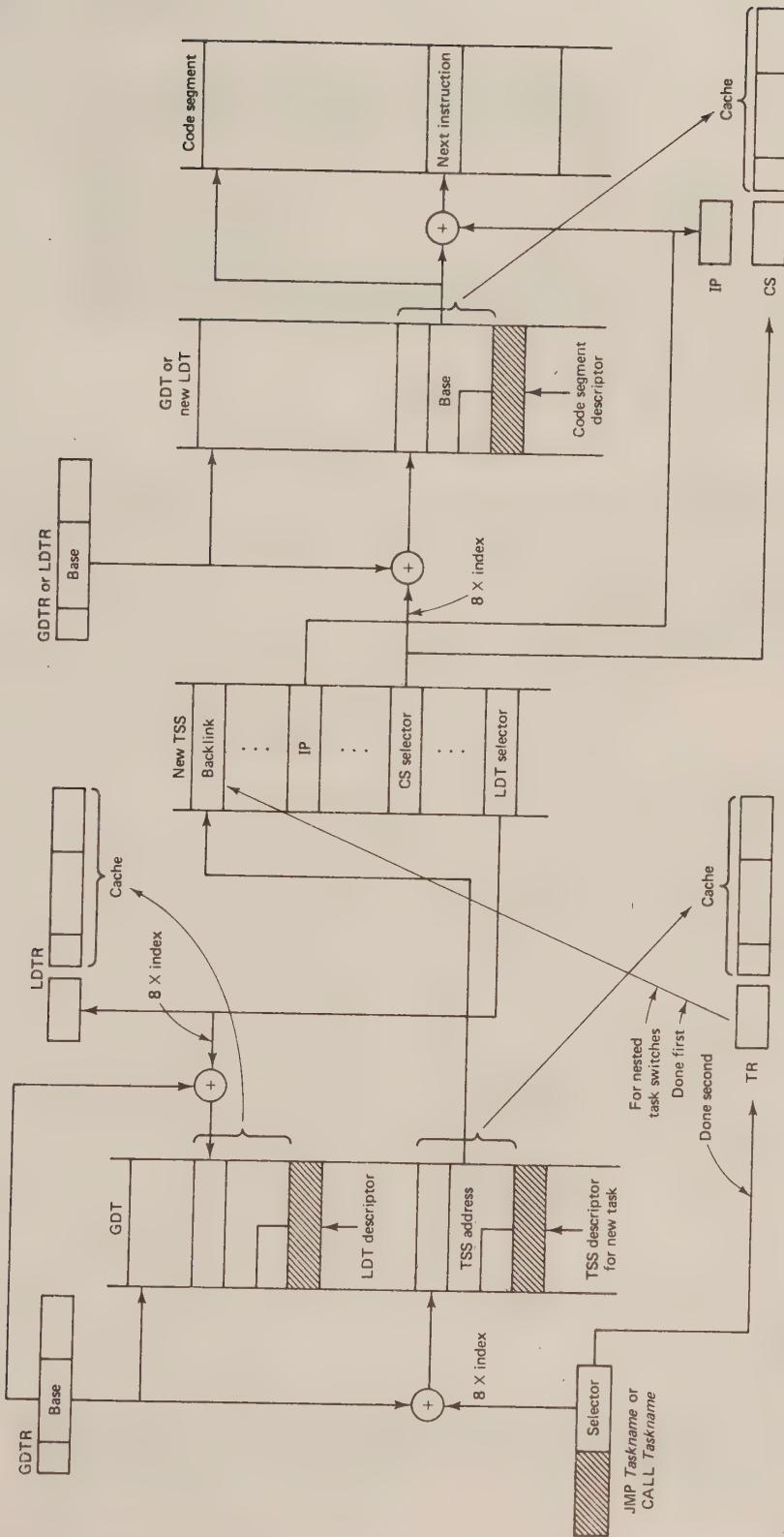
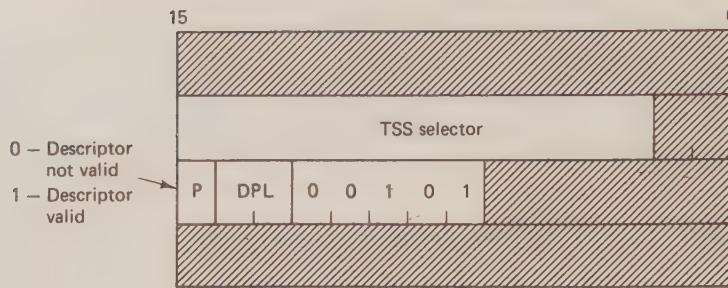


Figure 13-23 Determination of next instruction after a `JMP Taskname` or `CALL Taskname`.

- 577
- Notes:
1. Before initiating the switch, the current contents of the segment register selectors and the `IP`, `FLAGS`, `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI`, and `DI` are stored in the current `TSS`.
 2. `JMP` invokes an unnested task switch and `CALL` invokes a nested task switch.



Note: As with TSS and LDT descriptors, the use of the P bit is left to the operating system designer.

Figure 13-24 Definition of task gate descriptors.

is made. That is, the CPL of the current CS selector, which in this case is in the “CS selector” word in the new TSS, must be equal to the DPL of the code segment’s descriptor, if the segment is nonconforming, or greater than or equal to this DPL if the segment is conforming.

In addition to reloading the CS register’s selector and cache, the SS, DS, and ES registers must be reloaded as shown in Fig. 13-27. The protection rules are the

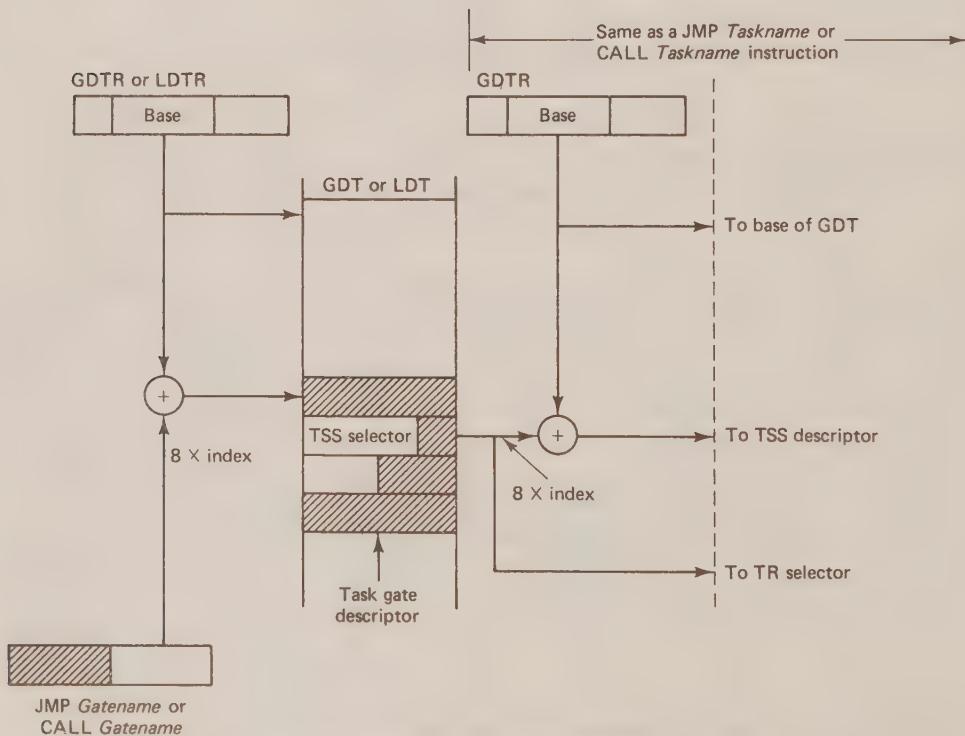
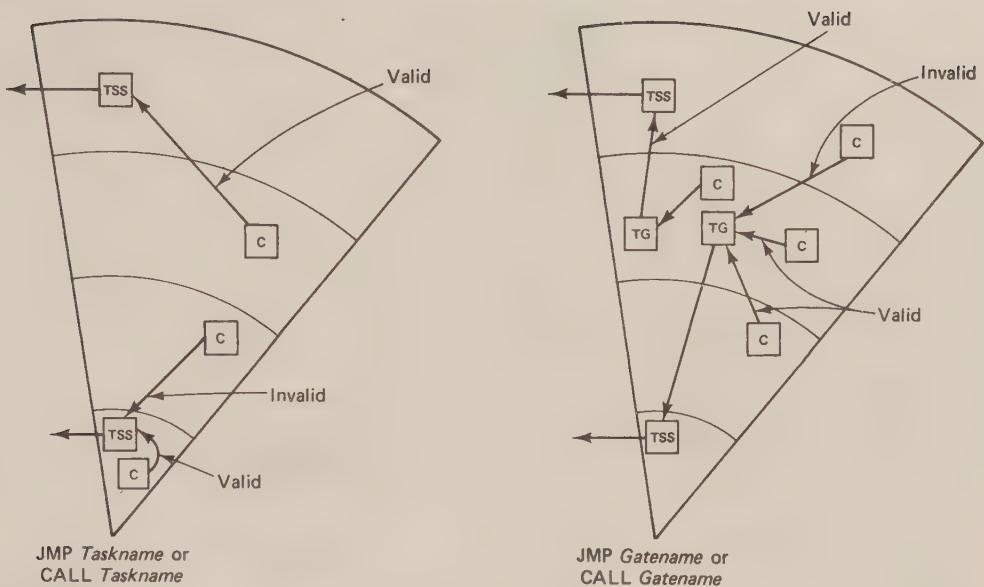


Figure 13-25 Determination of next instruction after a JMP Gatename or CALL Gatename with a task switch.



Note: Level of code segment being switched to does not depend on the CPL of the present task, the DPL of the TSS descriptor, or the DPL of the task gate, it depends only on the CS selector in the new TSS and the DPL in the corresponding descriptor.

Figure 13-26 Examples of valid and invalid task switches.

same as always, i.e., for DS and ES the DPL in the segment's descriptor must be greater than or equal to the CPL and for SS they must be equal.

If a CALL instruction is used, the switch is *nested*. What this means is that the new task is essentially being treated as subprogram of the calling task. The activity that takes place while calling a task is shown in Fig. 13-28. The additional actions caused by a nested switch are:

1. The NT flag (bit 14) in FLAGS register is set while the called task is executing and cleared when a return is made to the calling task.
2. As well as setting the busy bit in the TSS descriptor of called task, the busy bit of the calling task remains set. The busy bit in the called task is cleared when the return is made.
3. The TR's selector is put in the "backlink" word of the called task's TSS so that the return can be accomplished.
4. When NT = 1, the IRET instruction uses the "backlink" word to select the TSS descriptor of the calling task from the GDT, thereby causing a return switch.

A summary of the various types of branches that can be made while an 80286 is operating in its PVAM is given in Fig. 13-29. The purpose of the figure is to bring together in one place the protection checks, uses, and descriptions of all of the PVAM's types of branches.

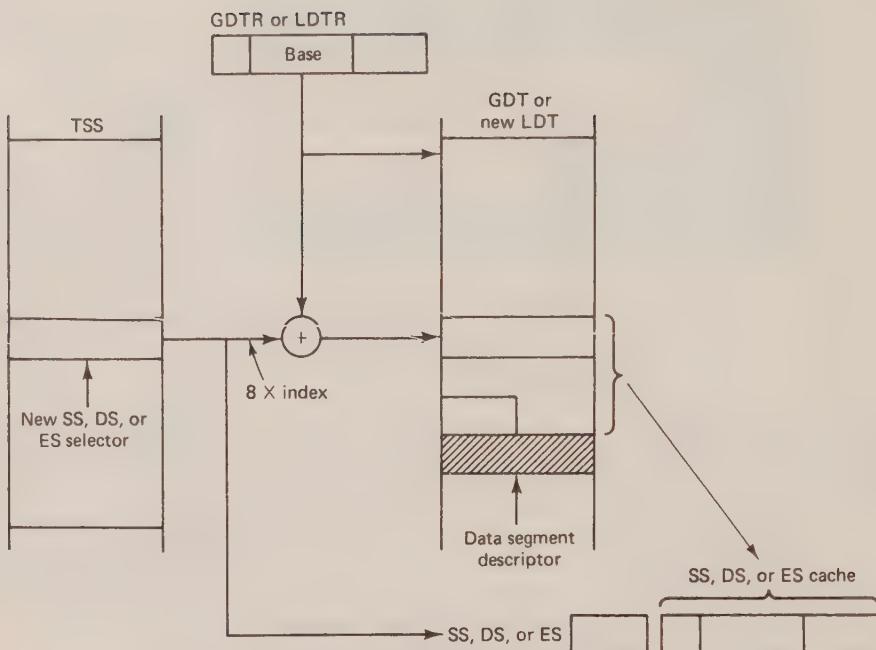


Figure 13-27 Loading the SS, DS, and ES after a task switch.

As a final task switching consideration, we need to examine multitasking in a coprocessor environment. Because a coprocessor, once it is given an instruction, runs independently from the CPU, there exists the problem of two tasks using the coprocessor at the same time. For example, suppose that task A initiates a coprocessor operation and then a switch is made to task B, and while task B is executing it also initiates a coprocessor operation. Although task B can wait until task A's operation is complete, it will not automatically store the coprocessor's state while it is using the coprocessor. Recall that the task switch only puts the current state of the CPU in task A's TSS, not the state of the coprocessor.

To alleviate this problem, all switches cause bit 3 in the MSW register, the task switch (TS) bit, to be set. Any time the coprocessor is to be used (i.e., a WAIT or ESC instruction is executed), this bit is checked. If it is 1, an interrupt will occur. The interrupt routine presumably would store the state of the coprocessor, replace the coprocessor's state with information corresponding to the current task, and clear the TS bit using a clear task switch flag (CLTS) instruction. CLTS is a *privileged instruction*, which means that it can be executed only from level 0. It is a 2-byte instruction with no operands and is defined in Fig. 13-30. The storing and replacing of the coprocessor's state could be accomplished using the save and load state instructions defined in Fig. 11-27. Appropriate storage areas could be obtained by extending the TSS by an amount sufficient to store the entire state of the coprocessor.

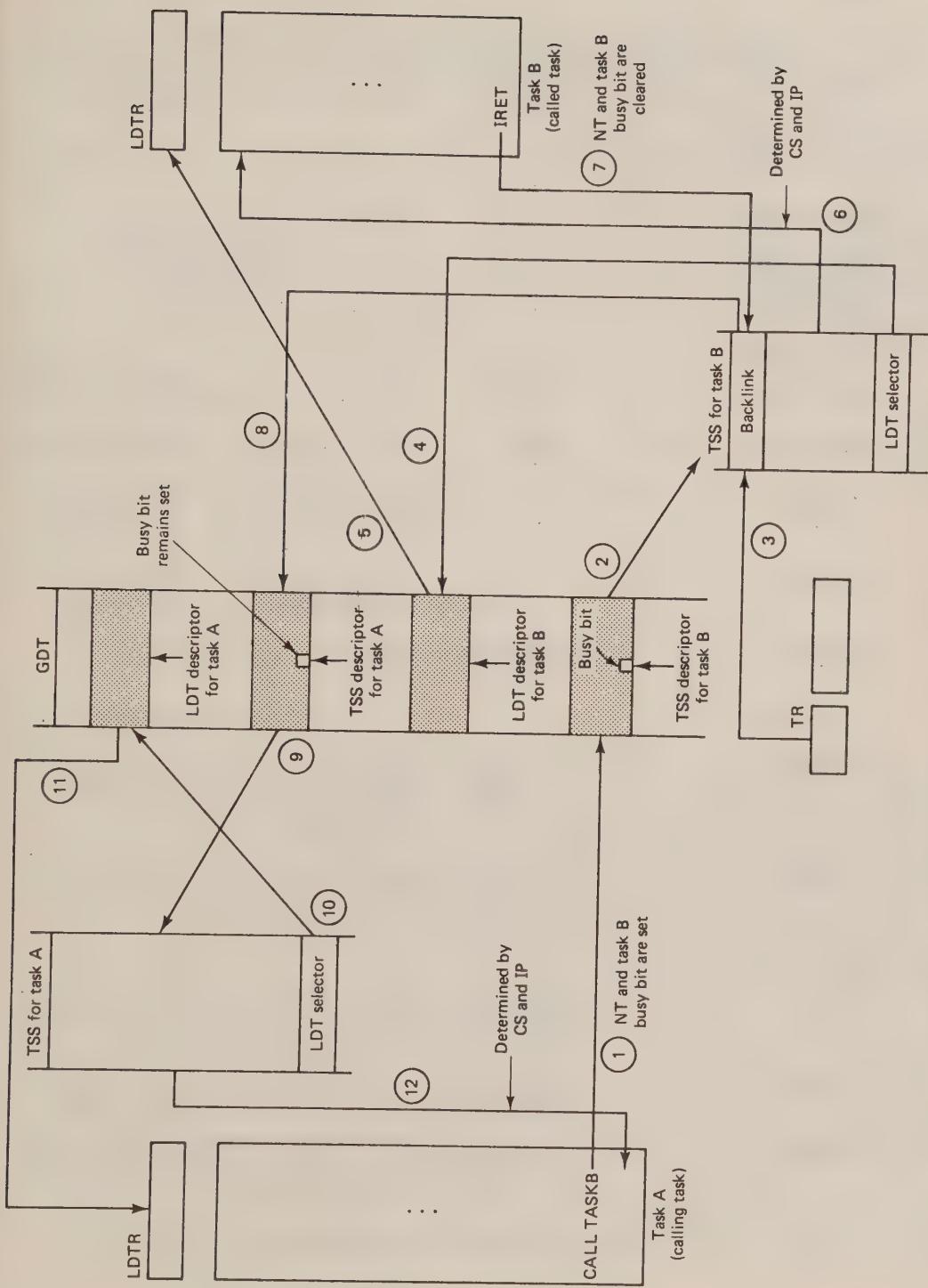


Figure 13-28 Activity while switching to and returning from a nested task.

The other instructions that are privileged when in the PVAM are the HLT, LGDT, LIDT, LMSW, LTR, and LLDT. Except for HLT, these instructions are for loading the PVAM-related registers (see Sec. 13-5 for their definitions).

| Branch Instructions | TSS DPL | Gate DPL | Privilege Level Checks | Target DPL* | Actions Taken | Purpose and Protection |
|------------------------------|----------|----------|--|-------------|--|---|
| Intrasegment Branches | | | | | | |
| JMP SHORT PTR <u>Label</u> | | | | | | |
| JMP NEAR PTR <u>Label</u> | | | | | | |
| CALL NEAR PTR <u>Label</u> | | | | | | |
| RET Constant | | | | | | |
| Intersegment Branches | | | | | | |
| JMP FAR PTR <u>Label</u> | | | DPL=CPL RPL≤CPL | | Selector <u>Label</u> points to descriptor which is put in CS cache | Normal branch to another segment--level protection |
| CALL FAR PTR <u>Label</u> | | | DPL=CPL RPL≤ CPL | | Same as JMP FAR PTR <u>Label</u> except CS:IP is pushed onto stack | Normal call to another procedure-level protection |
| JMP <u>Gatename</u> | DPL≥EPL | | DPL=CPL | | Selector <u>Gatename</u> points to call gate which points to descriptor which is put in CS cache | Indirect branch to another segment--gate and level protection |
| CALL <u>Gatename</u> | DPL≥EPL | | DPL≤CPL | | For no level change, same as JMP <u>Gatename</u> except CS:IP is pushed onto stack | Indirect call to a procedure--gate and level protection |
| RET Constant | | | Popped RPL>New CPL≥CPL Popped RPL=DPL | | CS:IP is popped from stack; CS selector points to descriptor which is put in CS cache | Normal return--level protection |
| Interlevel Branches | | | | | | |
| CALL <u>Gatename</u> | DPL≥EPL | | DPL≤CPL Stack DPL=Gate RPL =New CPL | | SS:SP, parameters, and CS:IP are pushed onto new stack; selector <u>Gatename</u> points to call gate which points to descriptor which is put in CS cache | Interlevel call to a procedure--gate and level protection |
| RET Constant | | | Popped RPL>New CPL≥CPL Popped RPL=DPL Stack DPL>New CPL=SS RPL Data DPLs≥New EPLs | | CS:IP is popped, parameters are skipped, and SS:SP is popped; CS selector points to descriptor which is put in CS cache | Normal return--level protection |
| Intertask Branches | | | | | | |
| JMP <u>Taskname</u> | DPL≥EPL | | DPL>New CPL Stack DPL>New CPL=SS RPL Data DPLs≥New EPLs | | Selector <u>Taskname</u> points to TSS descriptor which points to new TSS; task switch without nesting | Direct unnested task switch--task and level protection |
| CALL <u>Taskname</u> | DPL≥EPL | | DPL>New CPL Stack DPL>New CPL=SS RPL Data DPLs≥New EPLs | | Same as JMP <u>Taskname</u> except task switch is done with nesting | Direct nested task switch--task and level protection |
| JMP <u>Gatename</u> | DPL≥ EPL | | DPL>New CPL Stack DPL>New CPL=SS RPL Data DPLs≥New EPLs | | Selector <u>Gatename</u> points to task gate which points to TSS descriptor which points to TSS; task switch without nesting | Indirect unnested task switch--task, gate, and level protection |

Figure 13-29 Summary of branch instructions.

| | | | |
|---|---|--|--|
| CALL Gatename | DPL ≥ EPL Stack DPL=New CPL=SS RPL Data DPLs ≥ New EPLs | DPL=New CPL Same as JMP Gatename except task switch is done with nesting | Indirect nested task switch--task, gate, and level protection |
| IRET | DPL=New CPL Stack DPL=New CPL=SS RPL Data DPLs ≥ New EPLs | For nested task only; backlink selector points to new TSS; task switch without nesting | Return task switch from a nested task--task and level protection |
| RPL --requested privilege level in selector DPL --descriptor privilege level in descriptor CPL --current privilege level in CS register EPL --effective privilege level=Max(CPL,RPL) | | | Level protection--due to level relationships |
| *Although this column implies direct branches, in all cases the offset/selector pair could be in memory. | | | Gate protection--due to level and table accessibility of gate |
| | | | Task protection--due to level accessibility of TSS descriptor |

Figure 13-29 (continued)

| Name | Mnemonic and Format | Description |
|--|---------------------|-------------|
| Clear task switch flag | CLTS | (TS) ← 0 |
| Note: CLTS is a privileged instruction. | | |
| Flags: Only TS | | |
| Addressing modes: None | | |

Figure 13-30 Instruction for clearing the TS flag.

13-4 INTERRUPT SYSTEM

The interrupt system of the 80286 is similar to that of the 8086 except that it incorporates descriptors into the branching process in a way that is similar to the manner in which it executes CALL instructions. Instead of having a set of 2-word interrupt vectors starting at address 000000, the 80286 has an interrupt descriptor table (IDT) that is filled with 8-byte descriptors and can be anywhere in memory. The IDT is pointed to by the IDTR register (IDTR), which consists of an access rights byte, a 3-byte base, and a 2-byte limit just like the cache portion of a segment register. As with the 8086's set of interrupt vectors, the IDT can have up to 256 entries with each entry having a type that is determined by its position in the interrupt table. For example, type n is associated with the n th entry in the IDT and the address of its descriptor can be found by adding the base part of the IDTR to 8 times n . The size of the IDT in bytes is kept in the limit part of the IDTR and should be at least 8 times $r+1$, where r is the highest type being used. Like the GDTR, the IDTR is normally initialized only by the initialization code following a reset and does not change while the system is executing tasks.

In general, there are three fundamental classes of interrupts, software interrupts, external interrupts, and exceptions. *Exceptions* are interrupts that are caused

by such things as divide errors, protection violations, illegal operations, limits being exceeded and so on. Although the 8086's interrupt system included only two exceptions, the 80286 has been extended so that several of its interrupt types have been assigned to exceptions. How the 80286's interrupt types are assigned is given in Fig. 13-31. The fourth column in this figure indicates the instruction that is returned to once the interrupt routine is complete. Most exceptions will cause a return to the instruction that caused the fault, while software and external interrupts allow the current instruction to be completed and cause the return to be to the next instruction. By returning to the faulting instruction it is sometimes possible to recover from the problem without aborting the task. Whether or not possible recovery is likely is indicated in the fifth column of Fig. 13-31.

The divide and single step exceptions and the use of the TF flag are the same

| Type | Name | Class | Instruction Returned to | Recovery Worthwhile |
|-----------|---|---|-------------------------|---------------------|
| 0 | Divide error | Exception | Faulting | Maybe |
| 1 | Single step | Exception | Next | --- |
| 2 | Nonmaskable interrupt | External | Next | --- |
| 3 | Breakpoint (INT instruction) | Software | Next | --- |
| 4 | Overflow (INTO instruction) | Software | Next | --- |
| 5 | Bound check (BOUND instruction) | Software | Faulting | Yes |
| 6 | Undefined opcode (UD) | Exception | Faulting | Maybe |
| 7 | No math unit available (NM) | Exception | Unrelated | Yes |
| 8 | Double fault (DF) | Exception | Faulting | No |
| 9 | Math unit protection fault (MP) | Exception | Unrelated | No |
| 10 | Invalid task state segment (TS) | Exception | Faulting | Yes |
| 11 | Not present (NP) | Exception | Faulting | Yes |
| 12 | Stack fault (SS) | Exception | Faulting | Yes |
| 13 | General protection (GP) | Exception | Faulting | No |
| 14 | | Presently unused | | |
| 15 | | Presently unused | | |
| 16 | Math fault (MF) | Exception | Unrelated | Maybe |
| 17 | | Reserved for Intel software for runtime exception | | |
| 18 | : | Reserved for general use by Intel software | | |
| to | : | | | |
| 31 | : | | | |
| 32 | External maskable interrupts or Software interrupts | External or Software | Next | --- |
| to 255 | | | | |

Figure 13-31 Summary of interrupts.

as on the 8086. The additional 80286 exceptions are summarized in Fig. 13-32. Many of these exceptions are related to the multitasking features of the PVAM. Exception types 8, 10, 11, 12, and 13 have 1-word error codes associated with them. The error code is automatically pushed onto the top of the stack used by the interrupt routine just before the routine is entered, and provides the routine with information it may need while processing the exception.

An error code can be 0 or of one of the two formats given in Fig. 13-33, and the rightmost column in Fig. 13-32 gives the possible types of error codes for each exception. It is possible for an exception, such as a protection violation, to occur during an interrupt sequence. Sometimes, if the combination of interrupts is difficult to recover from, this will result in a double fault exception, but other times the exception will be processed and the original interrupt sequence will be continued.

| Type | Cause | Error Code Type |
|---------|---|----------------------------------|
| 6 (UD) | An invalid opcode or redundant prefixes occurred. | None |
| 7 (NM) | There is no NDP and the EM or TS bit in the MSW is 1, or a WAIT instruction is encountered and both the MP and TS bits in the MSW are 1. | None |
| 8 (DF) | An exception occurs during an interrupt sequence while processing another exception and the combination of exceptions normally precludes the possibility of recovery. | 0 |
| 9 (MP) | NDP has overrun the limit of a segment while trying to read or write the second or subsequent words of an operand. | None |
| 10 (TS) | Task switch is being made and the limit in the TSS descriptor is < 43, or the TSS contains an invalid selector or a selector that points to an invalid descriptor. | Selector index |
| 11 (NP) | Reference to a descriptor with P=0 is made, except when a TS or SS exception occurs (see TS and SS type exceptions). | Interrupt type or selector index |
| 12 (SS) | Stack underflow or overflow, or reference to a stack segment descriptor with P=0 but is otherwise valid. | 0 or selector index |
| 13 (GP) | Due to all other exceptions. Primarily: * Segment limit exceeded when referring to CS, DS, or ES segment. * Segment limit exceeded when referring to a descriptor table. * Branch to a data segment. * Write to a read only or code segment. * Read from an execute only segment. * Loading SS cache with read only descriptor (except when a task switch is being performed, then a TS exception occurs). * Loading CS cache from a data segment descriptor. * Switching to a busy task. * Violation of privilege level rules. * Invalid descriptor found in GDT, LDT, or IDT. | Interrupt type or selector index |
| 16 (MF) | There is an error input on the ERROR pin during a WAIT instruction or most ESC instructions, and the EM bit in the MSW is 0. | None |

Figure 13-32 Summary of PVAM exceptions.

In the latter case, the error code for the exception contains the type of the original interrupt in bits 15 through 3, bit 2 is 0, and bit 1 is 1. On the other hand, if an exception occurs during normal processing (not during an interrupt sequence) the error code contains the index of the selector causing the exception in bits 15 through 3, bit 2 contains the selector's table index, and bit 1 is 0. Bit 0 is 1 if the original interrupt was due to an external event; otherwise, it is 0. External events are external interrupts, single step interrupts, and coprocessor related interrupts. (For a more complete discussion of exceptions and error codes, refer to the Intel manual entitled *iAPX 286 Programmers Reference Manual*.)

The general protection (type 13) interrupt serves as a catchall exception for processing a variety of violations, some of which are listed in Fig. 13-32. Among other things, this exception will occur if an invalid descriptor is encountered. As summarized below, each of the three kinds of descriptor tables can contain only certain types of descriptors.

| Descriptor Type | GDT | LDT | IDT |
|-----------------|-----|-----|-----|
| Segment: | | | |
| Data | X | | X |
| Code | X | X | |
| System: | | | |
| TSS | X | | |
| LDT | X | | |
| Gate: | | | |
| Call | X | | |
| Task | X | X | X |
| Interrupt | | | X |
| Trap | | | X |

If a wrong type of descriptor is found in a table during processing, then there will be a general protection exception.

The possible entries in the IDT are interrupt gate, trap gate, and task gate descriptors. Task gates were defined in Fig. 13-24 and interrupt and trap gates are defined in Fig. 13-34. Unused descriptor positions in the IDT should have 0s in what would normally be the access rights byte. The only difference between an interrupt gate and a trap gate is that, if the interrupt type selects an interrupt gate (i.e., bit 0 of the access rights byte is 0), the IF bit in the FLAGS register is cleared during the interrupt sequence. If a trap gate is selected, the IF bit is left undisturbed.

As shown in Fig. 13-35, the primary purpose of the interrupt or trap gate is to provide the selector (and hence the code segment descriptor) and offset that ultimately determine the address of the first instruction in the interrupt routine. The action taken is similar to that caused by a

CALL Gatename

instruction in which the instruction selects a call gate. The differences are that the

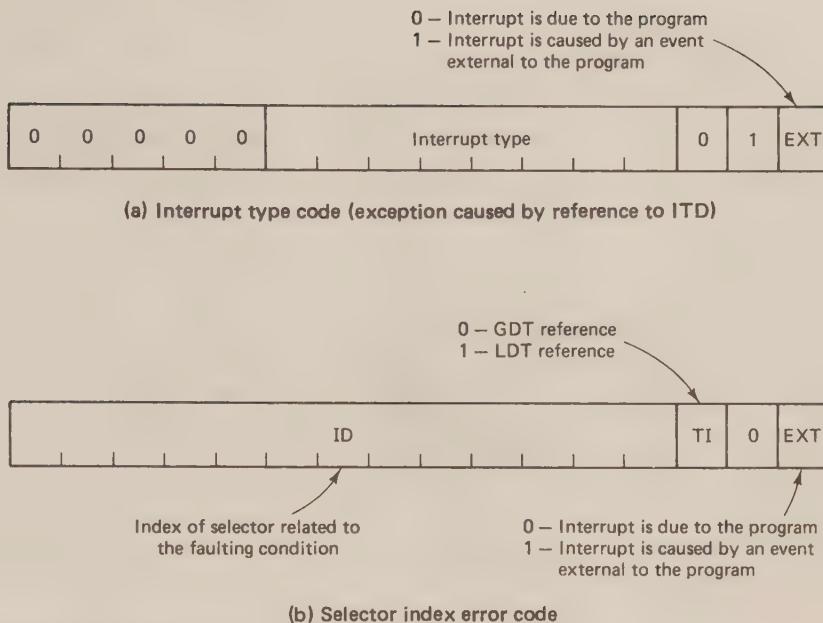
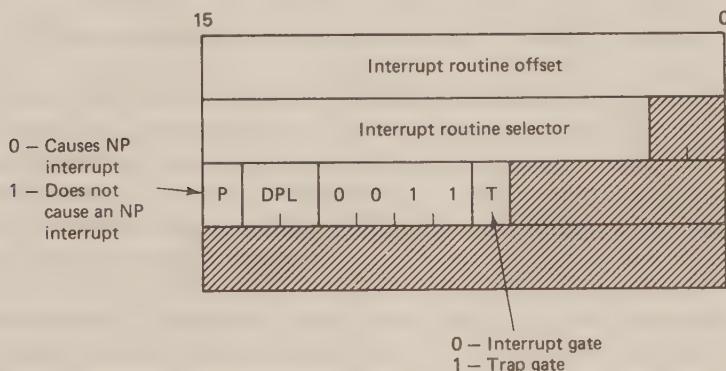


Figure 13-33 Error code formats.

selector and offset are gotten from the interrupt or trap gate instead of a call gate, no parameters are passed, and the FLAGS register as well as the return address is pushed onto the stack. Also, for an exception an error code may be pushed onto the stack. Figure 13-36 gives the interrupt routine's stack when it is entered. Two cases are shown, one for which the interrupt routine is at the same level as the interrupted program and one for which the levels are different.

Figure 13-37 shows what happens when the interrupt type points to a task



Note: An interrupt through an interrupt gate causes the IF flag to be cleared, but one through a trap gate does not.

Figure 13-34 Definition of interrupt and trap gate descriptors.

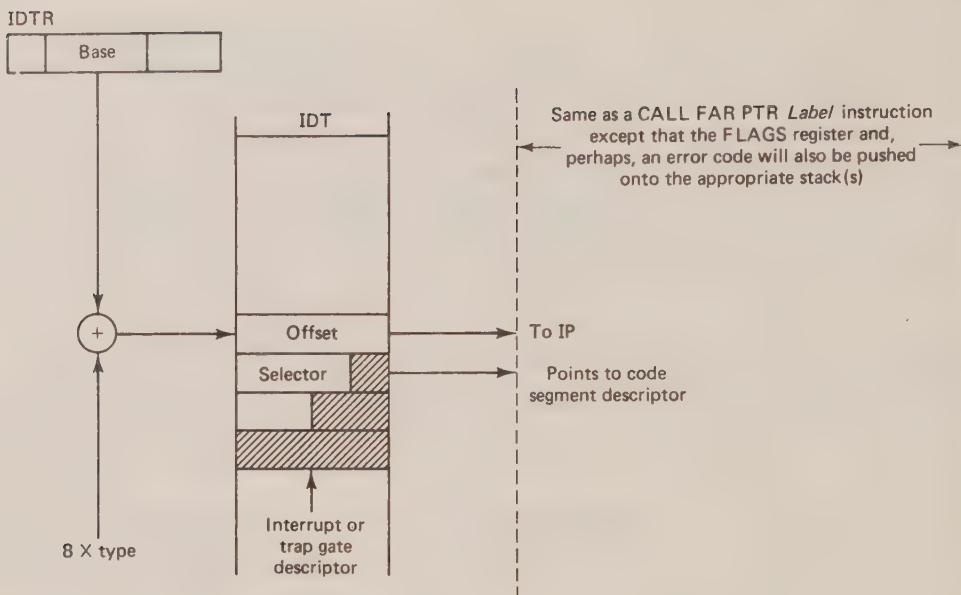


Figure 13-35 Determination of the interrupt routine's entry point when an interrupt or trap gate is used.

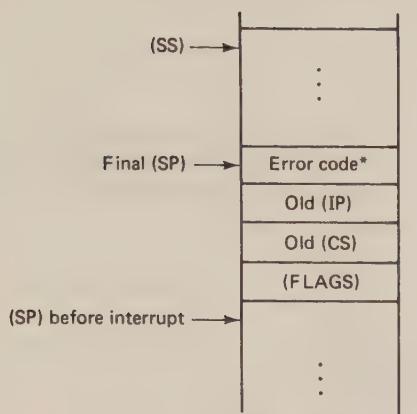
gate. The result is essentially the same as that of a

CALL Gatename

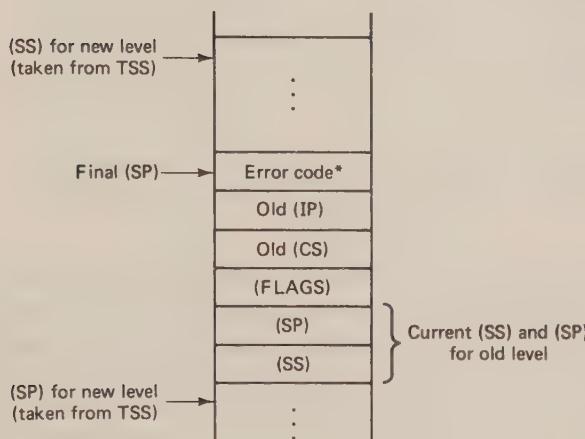
instruction in which *Gatename* selects a task gate—i.e., the NT flag is set and a nested task switch is performed. In addition, an error code may be pushed onto the interrupt routine's stack.

All returns from interrupt routines are made using IRET instructions. If NT = 0 (i.e., the routine was determined by an interrupt or trap gate), the return causes the (FLAGS) register and the return selector and offset to be popped from the stack. For NT = 1 (the task gate case), a return task switch is executed. If the interrupt involved an error code, care should be taken to remove the error code from the interrupt routine's stack before executing the return.

Just when a task gate should be used instead of an interrupt or trap gate is not easy to decide. The task switch resulting from a task gate would provide the interrupt routine with all of the flexibility and protection mechanisms of a separate task, but would increase the delay between the time of the interrupt and the time the interrupt routine is entered from 78 to 167 clock cycles. However, the difference is mostly offset by the fact that the task switch automatically saves the current register contents and initializes the segment registers. These actions normally require an additional 61 clock cycles, thus the increase is actually only from about 139 to 167 clock cycles. Putting the interrupt routine in a separate task also may make some of the data associated with the interrupted task inaccessible (e.g., the data



(a) Same privilege level



(b) Different privilege levels

Figure 13-36 Interrupt routine's stack after interrupt sequence.

*Error code is present only when a DF, TS, NF, SS, or GP interrupt occurs.

that can be referenced only through the interrupted task's LDT). Another consideration is that if a task switch is not performed, all of the protection rules for a branch within a task must be obeyed. Some exceptions, such as nested TSS-related faults, may require the use of a task gate (see the Intel manuals).

Figure 13-38 flowcharts the events of an interrupt sequence. The protection checks are similar to those made during the execution of a `CALL Gatename` instruction, but there are some differences. The DPL of the gate must be greater than or equal to the CPL of the interrupted task only if an INT instruction caused the interrupt. For interrupt and trap gates, the DPL of the interrupt routine's code segment descriptor may be more privileged than the interrupted program's CPL, even for nonconforming segments. However, this DPL must be numerically less than or equal to the CPL. As usual, if the code segment descriptor for the interrupt

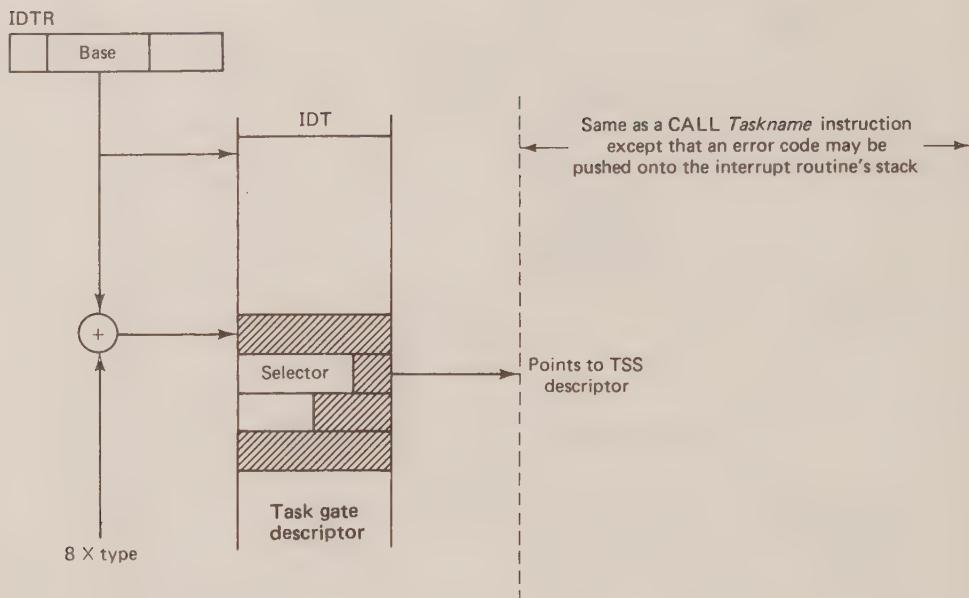


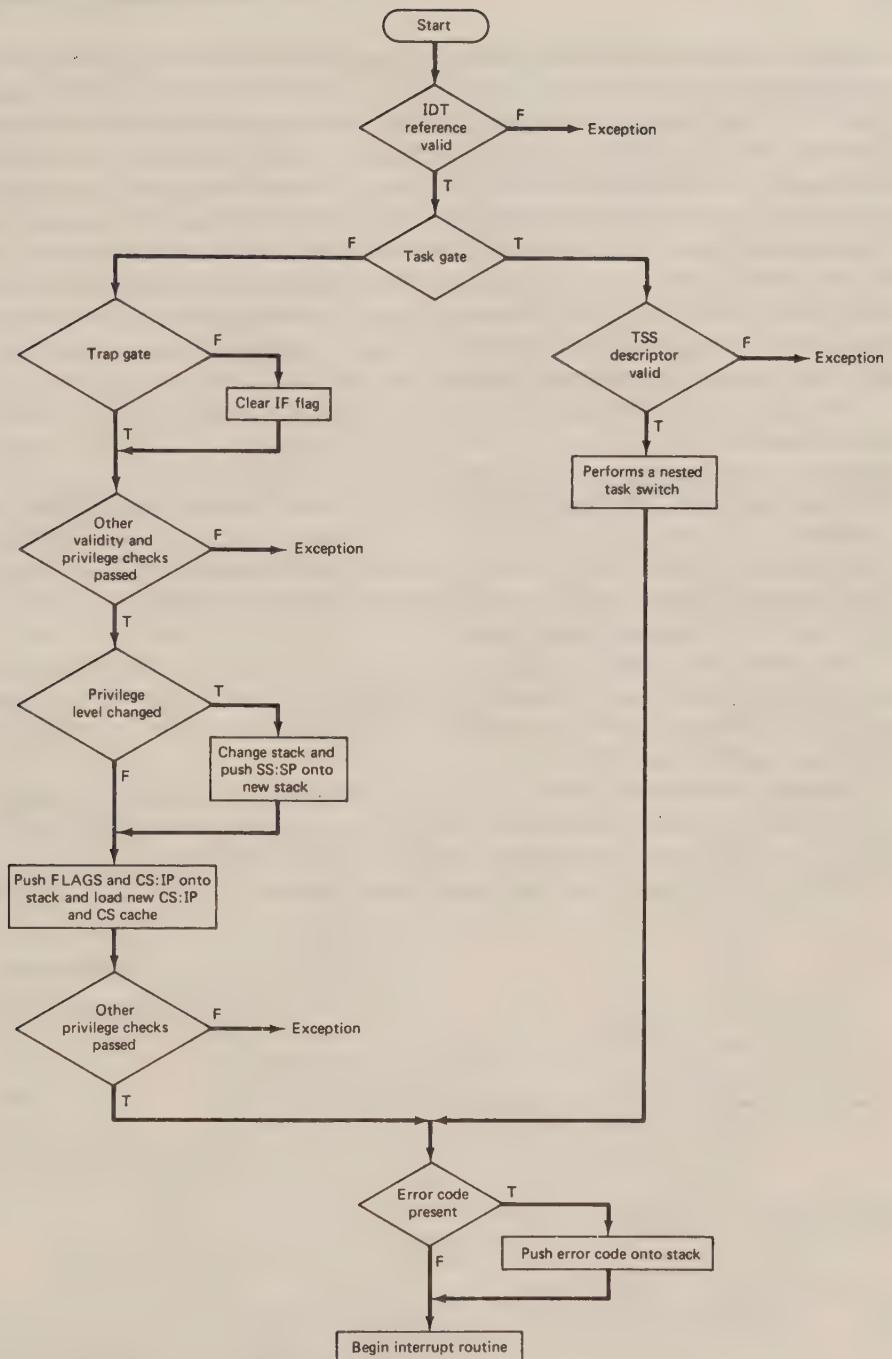
Figure 13-37 Determination of the interrupt routine's entry point when a task gate is used.

routine is conforming, then the final value for the CPL will equal the original value (i.e., the interrupt routine will take on the same level as the interrupted program).

As with the 8086, the STI and CLI instructions can be used to set and clear the IF bit, which enables external interrupts when 1 and disables them when 0. However, when in the PVAM these instructions can be executed only when the CPL is less than or equal to the IOPL bits, which are bits 12 and 13 of the FLAGS register. Such instructions are called *trusted instructions* and are used to restrict certain I/O activity to the levels designated by the operating system or during initialization. The other trusted instructions are IN, OUT, INS, OUTS, and LOCK. In addition, although the POPF instruction can be executed at any level, the IF bit is changed only when $CPL \leq IOPL$ and the IOPL field is changed only when $CPL = 0$. The IOPL can be changed by level 0 code by executing a PUSHF, changing the top of the stack so that the IOPL is set to the desired level, and then executing POPF.

13-5 SYSTEM AND TASK INITIALIZATION

This section will consider briefly how an 80286 is initialized to operate in its PVAM and tasks are introduced to the system as they need to be executed. In addition, the software Intel has made available to aid the development of multitasking 80286 systems will be mentioned.



Note: For single step exception, the TF is also cleared during the interrupt sequence.

Figure 13-38 Interrupt sequence.

Before outlining the procedure for bringing up an 80286 and putting it in its PVAM, we must define the 80286's reset state and MSW register. The definition of the MSW is given in Fig. 13-39 along with the contents put in the pertinent registers by a reset. Bit 0 of the MSW determines whether the 80286 is in the real address mode or PVAM and is cleared by a reset, thus causing the startup mode to always be the real address mode. Bits 2 and 1 are initially 00 and indicate whether an NDP is being used or the NDP instructions are to be emulated. Before any NDP instructions are executed these bits must be set to either 01 (an NDP is present) or 10 (the NDP is to be emulated). Bit 4, the task switch bit, was discussed in Sec. 13-3. The only bits in the MSW or FLAGS registers that are 1 following a reset are bits that are not currently being used. All flags are cleared to 0.

The IP, the selector portion of the CS, and pins A₂₃-A₂₀ dictate the address at which execution begins, and their values after a reset are FFF0, F000, and F, respectively. Because execution begins in the real address mode and A₂₃-A₂₀ are 1s, the address of the first instruction is FFFFF0 and the entire initialization code is normally put in the segment beginning at FF0000. Care must be exercised in writing the initialization code, however, because the first time the CS is changed A₂₃-A₂₀ drop to 0. Also, while making operand references using SS, DS, or ES, A₂₃-A₂₀ are 0, even during initialization.

The interrupt system is originally set up for real address mode operation which uses 256 double-word vectors beginning at 000000 (just like the 8086). Before changing to the PVAM, a PVAM compatible system must be created by establishing an IDT and loading the IDTR.

An important concept needed to create and modify the various tables included in the PVAM is aliasing. A descriptor that is used to point to the GDT or IDT, an LDT, or a TSS as a data segment so that it can be read from or written into by MOV instructions and other data manipulating code is called an *alias*. Because

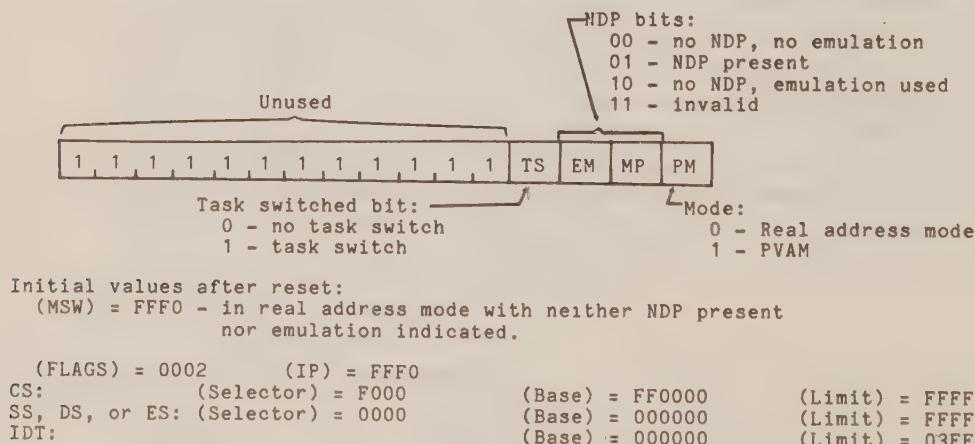


Figure 13-39 MSW definition and initial register contents.

it is these aliases that are needed by the operating system when it has to change a descriptor or TSS, they are the ones that are of primary interest to us. However, multiple descriptors that point to the same segment for the purpose of giving a segment more than one set of attributes are also called *aliases*. For example, a debugging program would need to refer to what normally is a code segment as a data segment. Therefore, the segment would have to be pointed to by a code segment descriptor and a data segment descriptor. Aliases also could be used for allowing one program read/write privileges when referencing a data segment, but restricting another program to only reading the segment.

The instructions needed by an initialization procedure or operating system to load and store the PVAM registers are defined in Fig. 13-40. A typical procedure which uses these instructions to bring up a multitasking 80286 is:

1. Branch to the beginning of the startup code by putting a JMP instruction at FFFFF0.
2. Load an image of the resident monitor, the operating system's LDT and TSS, and the beginning contents of the GDT and IDT. The GDT should include aliases for itself, the IDT, and, perhaps, the operating system's TSS; descrip-

| Name | Mnemonic and Format | Description |
|-------------|---------------------|----------------|
| Load GDTR* | LGDT SRC | (GDTR) ← (SRC) |
| Store GDTR | SGDT DST | (DST) ← (GDTR) |
| Load IDTR* | LIDT SRC | (IDTR) ← (SRC) |
| Store IDTR | SIDT DST | (DST) ← (IDTR) |
| Load MSW* | LMSW SRC | (MSW) ← (SRC) |
| Store MSW | SMSW DST | (DST) ← (MSW) |
| Load TR*+ | LTR SRC | (TR) ← (SRC) |
| Store TR+ | STR DST | (DST) ← (TR) |
| Load LDTR*+ | LLDT SRC | (LDTR) ← (SRC) |
| Store LDTR+ | SLDT DST | (DST) ← (LDTR) |

*These are privileged instructions.

+These instructions can be used only while operating in the PVAM.

Flags: None, except for LMSW which loads the flags into the MSW.

Addressing modes: For LGDT, SGDT, LIDT, and SIDT, SRC and DST must use memory addressing modes. For the remaining instructions, SRC and DST must use memory or register addressing modes.

Figure 13-40 Instructions for loading and storing the PVAM registers.

tors for the operating system's LDT and TSS; and any other gates and descriptors required for startup.

3. Load the GDTR and IDTR using the LGDT and LIDT instructions.
4. Switch to the PVAM mode using the LMSW instruction.
5. Clear out the CPU's code and instruction queues by executing a JMP SHORT instruction. This is necessary because these queues work differently in the PVAM than in the real address mode.
6. Set either the EM or MP bit in the MSW using the LMSW instruction.
7. Initialize the system's hardware.
8. Set the TR to point to a dummy TSS using the LTR instruction. This is necessary so that when a switch is made to the resident monitor the task switch will not destroy any useful information.
9. Switch to the resident monitor task using a JMP instruction. The old state is stored in the dummy TSS.

A set of code for implementing steps 4, 5, and 6 is:

| | | |
|-----------|-------|--------------------|
| SMSW | AX | |
| OR | AX,N | ;WHERE N IS 3 OR 5 |
| LMSW | AX | |
| JMP | SHORT | CONTINUE |
| CONTINUE: | . | |
| | . | |

After the foregoing procedure is complete the resident monitor would be responsible for introducing other system and applications tasks. Each time, a new task is to be brought into the system, the resident monitor would have to add its LDT and TSS descriptors to the GDT and create its LDT. One of the entries in the LDT would be an alias for the LDT (Intel suggests making the first descriptor this alias) and a descriptor for each of the task's segments. Some of the task's segments may be loaded at this time; others may be loaded by the "not present" interrupt routine as they are needed.

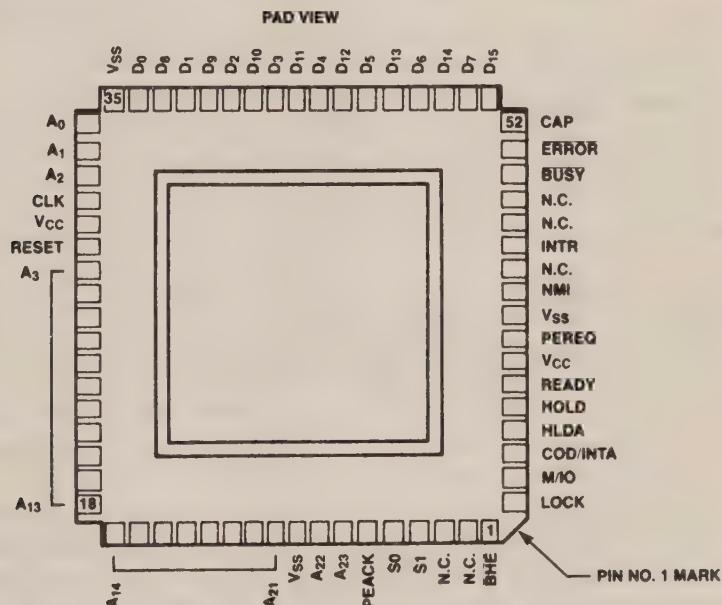
Putting together a multitasking system is much more involved than linking an applications program. Therefore, Intel has made available two programs for helping create such a system. These programs are the binder, BND286, and the builder, BLD286. The binder performs essentially the same functions as a linker, but the builder can be used to define (i.e., designate the attributes of) segments, gates, tasks, tables, and aliases and to allocate memory. The inputs to the builder are linked modules and the output can be a linked module with the newly defined segments, gates, tasks, and so on added, a load module, or a complete bootloadable system. In addition, a simulator, SIM286, is available for simulating a system during its development stage.

13-6 INTERFACING TO THE 80286

The 80286 is contained in a 68-pin quad-in-line package with both 5- and 8-MHz versions available. A pin diagram and the pin definitions of the 80286 are given in Fig. 13-41. Note that the address and data pins are not multiplexed as they are on the 8086 and 80186. The bus cycle status, which is referred to in Fig. 13-41, is given in Fig. 13-42.

Supporting devices are required to interface an 80286 to the system bus in a design that is similar to the one required for a maximum mode 8086. The 82284 clock generator, 82288 bus controller, and 82289 bus arbiter, which are the counterparts to the 8284A, 8288, and 8289, are available for the 80286 system design. The 8283 (or 8282) address latch chip, 8287 (or 8286) transceiver chip, and 8259A interrupt controller can be used in an 80286 system as well.

As an example of a bus interface for the 80286, Fig. 13-43 shows a configuration which has two buses, a local bus for the 80286 to access its private memory and I/O devices and a system bus shared by other bus masters. In the figure, the



NOTE: N.C. pads must not be connected.

(a) Pin diagram

Figure 13-41 Pin diagram and definitions for the 80286. (Part (a) is reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

| Symbol | In/Out 3-state | Description |
|----------------------------------|-------------------|--|
| CLK | I | System clock. Internally divided by 2 to generate processor clock (e.g., it is 16 MHz for an 8-MHz 80286). |
| D ₁₅ - D ₀ | I/O-3 | Data bus. |
| A ₁₉ - A ₀ | 0-3 | Address bus - only A ₁₅ - A ₀ are used for I/O addressing. In real address mode A ₁₉ - A ₀ are 0 except after a reset when they are 1 while fetching instructions, but are 0 when using SS, DS, or ES. |
| COD/INTA | 0-3 | Code/Interrupt Acknowledge - distinguishes between instruction fetches and memory operand references (see Fig. 13-42). |
| M/I _O | 0-3 | Memory/I _O select - distinguishes between memory and I/O transfers (see Fig. 13-42). |
| S ₁ , S ₀ | 0-3 | Bus cycle status - indicates beginning and type of bus cycle (see Fig. 13-42). |
| PEREQ, PEACK | I, 0-3 | Processor Extension Request, Acknowledge - PEREQ is used by the coprocessor to request the 80286 to transfer a data operand and PEACK is used by the 80286 to acknowledge that the requested operand is being transferred. |
| BUSY, ERROR | I, I | Processor Extension Busy, Error - BUSY is sent by the coprocessor to indicate it is busy. Low causes the 80286 to wait during a WAIT or most ESC instructions. ERROR is used by the coprocessor to cause the 80286 to perform a type 16 interrupt when a WAIT or most ESC instructions is executing. |
| CAP | --- | Substrate filter capacitor - must be connected to a 0.047 μ F $\pm 20\%$, 12 V capacitor. |
| RESET | I | For reset conditions see Fig. 13-39. |

Note: BHE, LOCK, READY, HOLD, HLDA, INTR, NMI, V_{ss} (ground) and V_{cc} (+5V supply voltage) serve the same purposes as on an 8086.

(b) Pin definitions

Figure 13-41 (continued)

crystal oscillator connected to the 82284 should have twice the desired frequency of the internal processor clock. The ARDYEN and SRDYEN respectively enable ARDY and SRDY to generate the ready signal to the 80286. The SRDYEN pin can be grounded because the local bus is dedicated to the 80286. On the other hand, the ARDYEN pin must be activated only when the 80286 is accessing the system bus. Most of the 82288 and 82289 signals serve the same purposes as they do in the 82288 and 82289, which were explained in Sec. 11-2-3. The following paragraphs describe those signals that are new.

The MB (multibus mode select) input to the 82288 determines the timing of command outputs. For the 82288 associated with the system bus, this input is strapped to +5 V to select the multibus timing mode. In the multibus mode, the command and data activation outputs are delayed to meet the required address and data setup times. This delay will result in at least one wait state being inserted

| 80286 Bus Cycle Status Definition | | | | |
|-----------------------------------|------|----|----|------------------------------------|
| COD/INTA | M/IO | S1 | S0 | Bus cycle initiated |
| 0 (LOW) | 0 | 0 | 0 | Interrupt acknowledge |
| 0 | 0 | 0 | 1 | Reserved |
| 0 | 0 | 1 | 0 | Reserved |
| 0 | 0 | 1 | 1 | None; not a status cycle |
| 0 | 1 | 0 | 0 | IF A1 = 1 then halt; else shutdown |
| 0 | 1 | 0 | 1 | Memory data read |
| 0 | 1 | 1 | 0 | Memory data write |
| 0 | 1 | 1 | 1 | None; not a status cycle |
| 1 (HIGH) | 0 | 0 | 0 | Reserved |
| 1 | 0 | 0 | 1 | I/O read |
| 1 | 0 | 1 | 0 | I/O write |
| 1 | 0 | 1 | 1 | None; not a status cycle |
| 1 | 1 | 0 | 0 | Reserved |
| 1 | 1 | 0 | 1 | Memory instruction read |
| 1 | 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 1 | None; not a status cycle |

Figure 13-42 Definition of bus cycle status. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

in the bus cycle. If extra delay is desirable to increase the setup time, the CMDLY (command delay) pin should be connected to a delay logic instead of being grounded as shown in Fig. 13-43. For the local bus interface, the MB pin is grounded to optimize timing and provide short bus cycles. The MB pin also selects the function of the CEN/AEN pin. When MB is grounded, the CEN/AEN input serves as the CEN (command enable) function. When MB is high, this input serves as the AEN (address enable) function and it should be controlled by the AEN output of the 82289.

The CENL (command enable latch) pin is for selecting the appropriate 82288 for the current bus cycle. Therefore, this pin should be controlled by the address decoder. Since the CENL input of the 82288 and the SYS/RESB input of the 82289 are latched internally, they will remain stable for the duration of the bus cycle. The COD/INTA pin is not used in this example. It can be fed to the address decoder to enable one of the two buses during interrupt acknowledge cycles.

An 80286 bus cycle consists of a send-status state, denoted T_s , plus one or more perform-command states, denoted T_c . Each state is one processor clock long—i.e., two system clocks long. During the T_s state, the processor sends out the bus cycle status $\bar{S}1$ and $\bar{S}0$. Upon detecting a change from the passive $\bar{S}1 = \bar{S}0 = 1$ state, the 82288 will output a pulse on its ALE (address latch enable) pin during the second phase of the state T_s . During the T_c state, the bus controller sends out a bus command, and a data transfer is performed. If the READY signal is not received in T_c , the 80286 enters the T_c state again. The T_c states may be repeated without limit until terminated by READY. Therefore, the T_c state also serves the purpose of a wait state to accommodate slow memory and I/O devices.

It should be pointed out that the 80286 uses pipelined address timing. The address, M/IO, and COD/INTA to be used in the current bus cycle actually were sent out during the second phase of T_c in the previous bus cycle. The overlapped address timing allows maximum address setup time and hence permits slower mem-

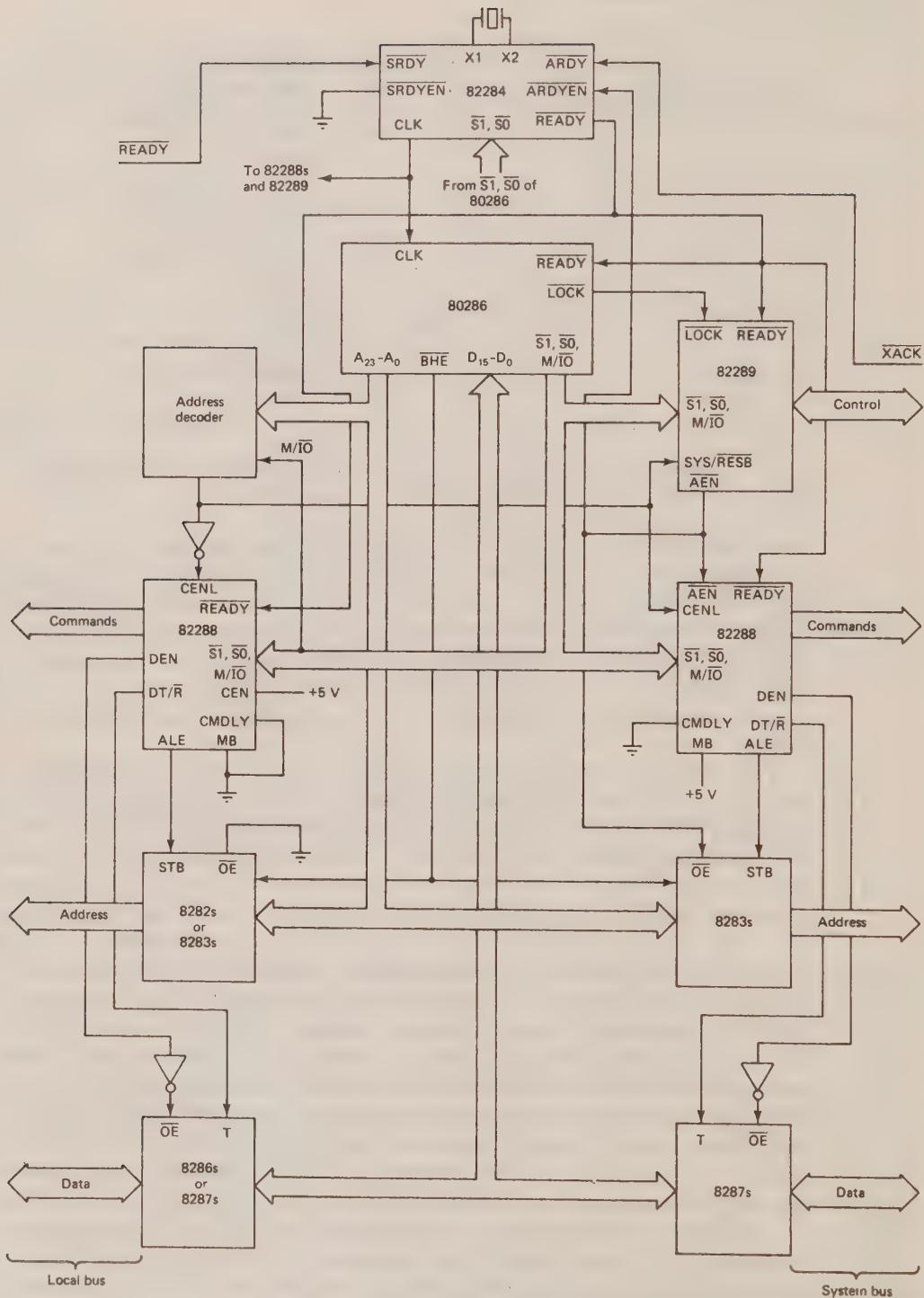


Figure 13-43 80286 in a dual-bus configuration.

ory devices to be used with no wait states. An example of a bus interface that utilizes this feature will be given later. With no wait states, a bus cycle requires only two states. Therefore, the maximum transfer rate for an 8-MHz CPU will be 4 megawords per second (assuming even addresses). Figure 13-44 shows a typical read bus cycle timing diagram.

In addition to the T_s and T_c states, there are two other possible bus states, the T_i (idle) state and the T_h (hold) state. As with the 8086, when no bus operations are required, the 80286 enters the idle state. To respond to a HOLD request, the 80286 outputs the HLDA signal and enters the T_h state. While in the T_h state, the processor puts all address, data, and status output pins in their high-impedance state.

Since both the 8086 and 80286 use a 16-bit data bus, a memory interface for the 80286 is very similar to that for the maximum mode 8086. This means that the BHE and A0 bus lines must be used in conjunction with the write command signal to determine whether only the low-order byte, only the high-order byte; or both

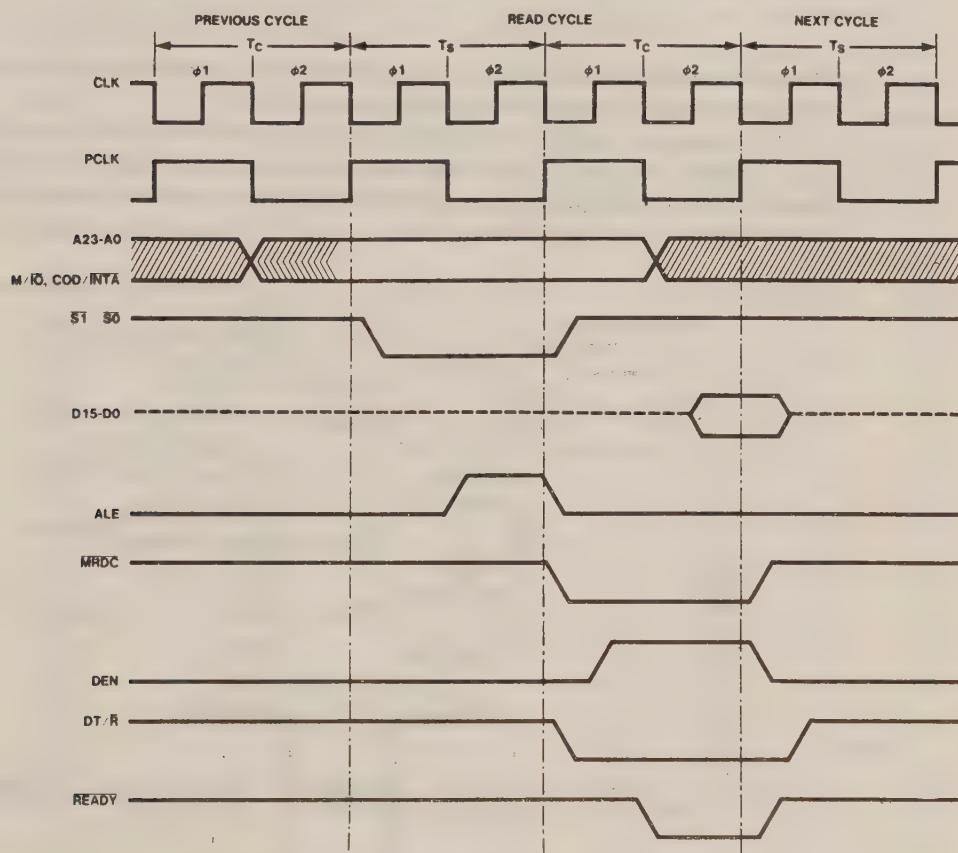


Figure 13-44 Read cycle timing of the 80286. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

bytes is to be written. However, the 80286 has more stringent timing requirements than the 8086. With no wait states, a bus cycle for the 8086 has 4 processor clock cycles, and for the 80286 it has only two.

If the memory interfacing method used for the maximum mode 8086 is implemented for the 80286, a simplified logic diagram can be as shown in Fig. 13-45. In this implementation, the address is latched using the ALE signal generated by the bus controller. Referring to the read cycle timing given in Fig. 13-44, the address will be output during ϕ_2 of the T_s state and the datum will be ready by the end of T_c if no wait states occur. Therefore, the maximum read access time allowed for memory devices without causing a wait state can be calculated as:

- 3 system clock cycles (ϕ_2 of T_s plus T_c) – delay of ALE from ϕ_2 (max)
- 8282 latch delay (max) – 8286 transceiver delay (max) – data setup time for the 80286 (min).

For an 8-MHz 80286, the maximum memory access time is

$$187.5 \text{ ns} - 15 \text{ ns} - 45 \text{ ns} - 30 \text{ ns} - 10 \text{ ns} = 87.5 \text{ ns}.$$

With this implementation, most memory devices, except for fast static RAM, are not fast enough to operate with zero wait states.

Because the address for the current bus cycle is output before the current cycle begins (see Fig. 13-44), the valid address can be latched into the address latches at an earlier point. An additional clock cycle can be gained for the access time by generating the address strobe as soon as the previous memory command

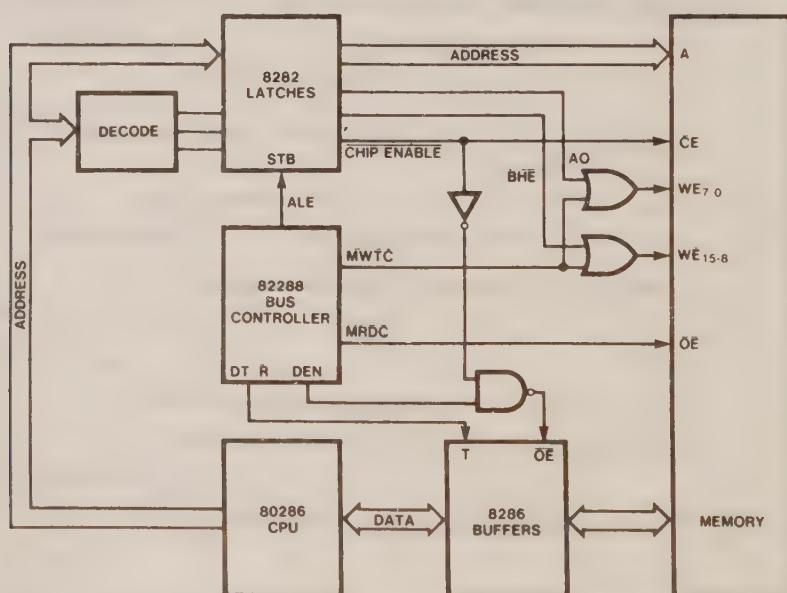


Figure 13-45 Memory interface using standard ALE signal. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

becomes inactive. An implementation of this scheme is shown in Fig. 13-46. In addition to the strobe logic, 8283 latches and 8287 transceivers are used because they are faster than their noninverting counterparts. For an 8-MHz 80286, the maximum read access time with no wait states is increased to:

$$\begin{aligned} 250 \text{ ns (4 system clock cycles)} &- 15 \text{ ns (maximum command inactive time)} \\ - 8 \text{ ns (STB logic delay)} &- 40 \text{ ns (maximum 8283 latch delay)} - 22 \text{ ns} \\ (\text{maximum 8287 transceiver delay}) &- 10 \text{ ns (minimum data setup time for} \\ \text{80286)} = 155 \text{ ns.} \end{aligned}$$

Therefore, slower memory devices can be used without sacrificing the data transfer rate. It should be noted that the additional access time comes with the reduction of address hold time. Although the reduced address hold time is sufficient for typical memory devices, this is not true for many I/O devices. Therefore, a separate set of address latches will be required for the I/O devices.

Two major items must be considered when interfacing I/O devices to an 80286. One is how to connect the data bus to 8-bit and 16-bit devices. Several such techniques were discussed in Sec. 9-7 and will not be repeated here. The other is device protection. For an 80286-based system, it might be desirable to use memory-mapped I/O. An advantage in doing this is that it offers full device protection under the 80286 PVAM. I/O-mapped devices are protected only by the I/O privilege level (IOPL) of the 80286 FLAGS register. Depending on its CPL, a task is either permitted to execute all I/O instructions (i.e., IN, OUT, INS, and OUTS) or is

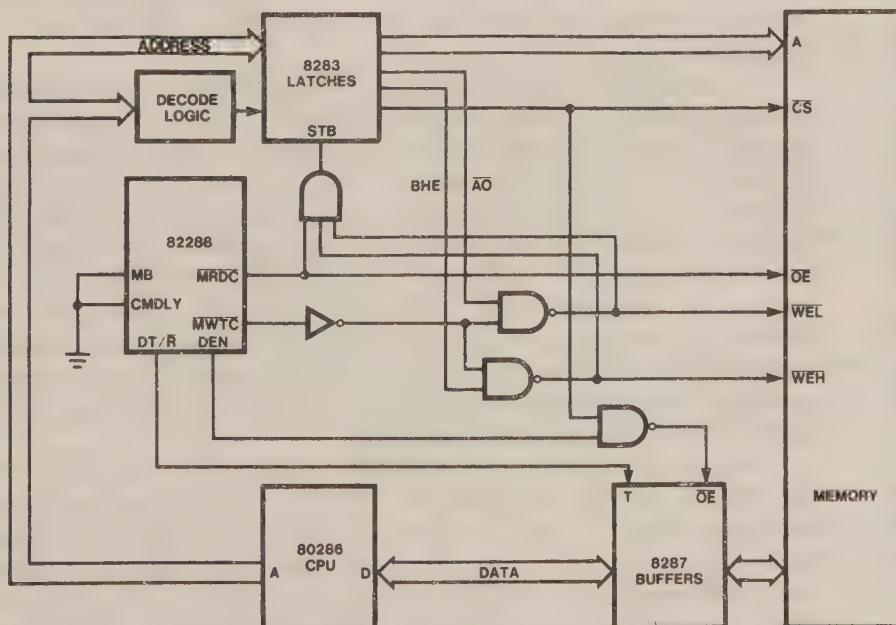


Figure 13-46 Memory interface using special strobe logic. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

denied from executing any of them. Therefore, protection of I/O devices for I/O-mapping is less flexible than for memory-mapping that can use the read only and data segment DPL protection mechanisms.

13-7 THE 80287

The 80287, which is the 8087's counterpart, is the numerical data coprocessor for the 80286. Both 5- and 8-MHz versions are available. The 80287 is upward software compatible with the 8087, although two processor control instructions are different. Namely, the 80287 has two new instructions, FSETPM (enter protected mode) and FSTSW AX (store control word directly into AX), and it ignores two of the 8087's instructions, FENI (enable interrupts) and FDISI (disable interrupts). Unlike the 8087 which indicates errors with interrupt requests handled through the 8259A, the 80287 signals errors through the 80286's ERROR pin, thus eliminating the need for the FENI and FDISI instructions. An active ERROR signal causes the 80286 to perform a type 16 exception interrupt. Just like its host, the 80287 can operate in either the real address mode or the PVAM. After reset, the 80287 is in the real address mode, and the FSETPM instruction is required to switch it to the protected mode. When operating in the protected mode, each instruction and data pointer represents a 16-bit segment selector and a 16-bit offset. (In the real address mode, each pointer represents a 20-bit physical address as in the 8087.) The new instruction FSTSW AX allows the 80287 to store its control word directly into the AX register of its host. (For the 8087, AX is not a valid operand in its FSTSW instruction.)

Communication between the 80287 and 80286 is quite different from that between the 8087 and 8086. The 80287 relies on its host to fetch or store memory operands. This is necessary to enforce the 80286's memory management and protection under the protected mode. To transfer a word to the memory, the 80287 outputs a request at the PEREQ pin, and then the 80286 reads the data word from the 80287, returns an acknowledge through the PEACK pin and writes the data word to the memory. In addition, to communicate with the 80287 the 80286 uses the four I/O addresses 00F8, 00FA, 00FC and 00FE. These addresses must be decoded, as specified in Fig. 13-47, to activate the select and command inputs of the 80287. The numerical processor select (NPS2 and NPS1) and command (CMD1 and CMD0) pins are used to indicate that the CPU is performing an ESC instruction and to direct the operation of the 80287.

Figure 13-48 shows how the 80287 is connected to the 80286. The 80287 has two clock input pins, CLK286 and CLK. The CLK286 pin must be connected to the system clock so that it can properly sample the CPU's bus status. The CLK input is used to generate the internal processor clock signals. Because the 80286 and 80287 need not be operated at the same clock rate, this input can be driven either by the same 82284 clock as the CPU or by a separate 8284A clock, depending on the CKM (clock mode) strapping pin. When the CKM pin is grounded, the

| I/O Address (Hexadecimal) | 80287 Select and Command Inputs | | | |
|------------------------------|---------------------------------|------|------|------|
| | NPS2 | NPS1 | CMD1 | CMD0 |
| 00F8 | 1 | 0 | 0 | 0 |
| 00FA | 1 | 0 | 0 | 1 |
| 00FC | 1 | 0 | 1 | 0 |
| 00FE | * Reserved For Future Use | | | |

NOTE:

These addresses are generated automatically by the 80286. Users should not attempt to reference these I/O addresses explicitly, at the risk of corrupting data within the 80287.

Figure 13-47 I/O address decoding for the 80287. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

80287 internally divides the CLK clock by three and the CLK pin should be connected to the system clock generator. The dedicated clock mode is selected by connecting the CKM pin to +5 V and is primarily used for an 8-MHz 80287. In this mode, the CLK clock input is used directly without being divided by three. Therefore, a separate clock generator such as 8284A should be used to provide compatible clock signals.

The NPRD (numerical processor read) and NPWR (numerical processor write) pins are connected directly to the IORC and IOWC outputs of the bus controller so that the 80287 can respond to I/O commands. The S1, S0, COD/INTA, READY, RESET, HLDA, PEACK, PREQ, ERROR, BUSY, and D₁₅-D₀ pins are connected to the corresponding pins of the CPU. The BUSY signal indicates that the 80287 is busy. During the execution of an ESC instruction, the 80286 will automatically check the BUSY input first and wait if this signal is low. Therefore, unlike an 8086/8087 system, for an 80286/80287 system there is no need to insert a WAIT instruction before each ESC instruction.

In Fig. 13-48, one of the low-order data lines D₇-D₀ is connected to a pull-up resistor. This ensures that if the 80287 is not present, at least one of the lines D₇-D₀ will be high when the 80286 reads the initialized 80287 status word. This permits the initialization code to automatically set the EM and MP bits in the MSW to 10 so that the 80287 instructions will be emulated.

The 8205 decoder is to recognize the I/O addresses 00F8 through 00FE which, as mentioned above, are reserved for the 80287. This decoder is enabled when M/I_O is low and COD/INTA is high, and it activates NPS1, CMD1, and CMD0 as specified in Fig. 13-47. During an I/O transfer between the 80286 and the 80287, it is necessary to disable the data transceivers connected to the system bus. This is accomplished by the NAND gate which is connected to the OE input of the transceiver chips.

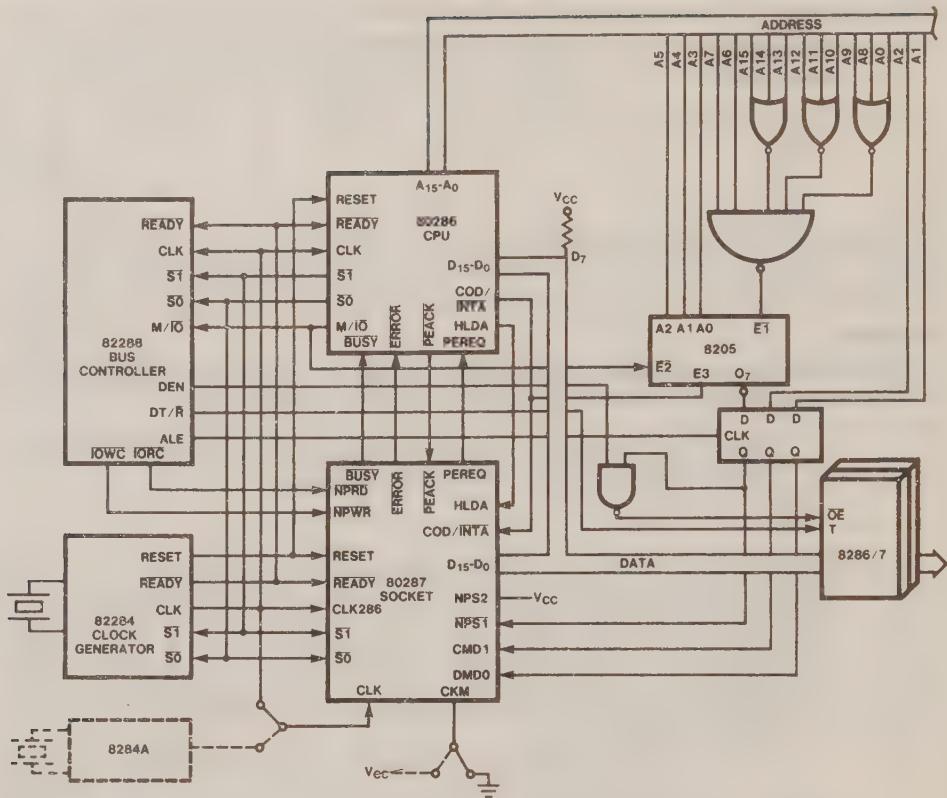


Figure 13-48 Connection of the 80286 and 80287. (Reprinted by permission of Intel Corporation. Copyright 1983. Courtesy of Intel Corporation.)

BIBLIOGRAPHY

1. *iAPX 286 Builder User's Guide* (Santa Clara, Calif.: Intel Corporation, 1984).
2. *iAPX 286 Hardware Reference Manual* (Santa Clara, Calif.: Intel Corporation, 1983).
3. *iAPX 286 Operating Systems Writers Guide* (Santa Clara, Calif.: Intel Corporation, 1983).
4. *iAPX 286 Programmer's Reference Manual* (Santa Clara, Calif.: Intel Corporation, 1984).
5. *iAPX 286 Simulator User's Guide* (Santa Clara, Calif.: Intel Corporation, 1982).
6. *iAPX 286 Utilities User's Guide* (Santa Clara, Calif.: Intel Corporation, 1984).

Appendix: 8086/8088 Instruction Set Summary

The first two columns in Table A-1 give the mnemonic and a brief description of each instruction. The third column indicates the number of clock cycles needed to execute the instruction. For the 8086, four clock cycles need to be added for each word transfer with an odd address. For the 8088, four clock cycles need to be added for each word transfer. There are two numbers of clock cycles for the conditional branch and loop instructions; the one before the "/" is for when a branch is taken and the other number is for when the branch is not taken.* The fourth column gives the number of bytes, or the range of numbers of bytes, for the instruction, and the fifth column indicates the possible flag settings using the following symbols:

- Not affected
- 0 Cleared to 0
- 1 Set to 1
- x Set or cleared according to the result
- u Undefined
- r Restored from previously saved value

The sixth column contains the page number where the definition of the instruction can be found.

*For the control transfer instructions, 8 clock cycles have been included for loading the queue.

TABLE A-1 8086/8088 INSTRUCTION SET SUMMARY

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|--|--------------|-----------------|---------------------|-----------|
| | | | | O D I T S Z A P C | |
| AAA | ASCII adjust for addition | 4 | 1 | u - - - u u x u x | 76 |
| AAD | ASCII adjust for division | 60 | 2 | u - - - x x u x u | 76 |
| AAM | ASCII adjust for multiplication | 83 | 2 | u - - - x x u x u | 76 |
| AAS | ASCII adjust for subtraction | 4 | 1 | u - - - u u x u x | 76 |
| ADC | Add with carry | | | x - - - x x x x x | 65 |
| | Register to register | 3 | 2 | | |
| | Memory to register | 9+EA | 2-4 | | |
| | Register to memory | 16+EA | 2-4 | | |
| | Immediate to register | 4 | 3-4 | | |
| | Immediate to memory | 17+EA | 3-6 | | |
| | Immediate to accumulator | 4 | 2-3 | | |
| ADD | Addition | | | x - - - x x x x x | 65 |
| | Register to register | 3 | 2 | | |
| | Memory to register | 9+EA | 2-4 | | |
| | Register to memory | 16+EA | 2-4 | | |
| | Immediate to register | 4 | 3-4 | | |
| | Immediate to memory | 17+EA | 3-6 | | |
| | Immediate to accumulator | 4 | 2-3 | | |
| AND | Logical AND | | | 0 - - - x x u x 0 | 93 |
| | Register to register | 3 | 2 | | |
| | Memory to register | 9+EA | 2-4 | | |
| | Register to memory | 16+EA | 2-4 | | |
| | Immediate to register | 4 | 3-4 | | |
| | Immediate to memory | 17+EA | 3-6 | | |
| | Immediate to accumulator | 4 | 2-3 | | |
| CALL | Call a procedure | | | - - - - - - - - - | 157 |
| | Intrasegment direct | 19 | 3 | | |
| | Intrasegment indirect through register | 16 | 2 | | |
| | Intrasegment indirect through memory | 21+EA | 2-4 | | |
| | Intersegment direct | 28 | 5 | | |
| | Intersegment indirect | 37+EA | 2-4 | | |
| CBW | Convert byte to word | 2 | 1 | - - - - - - - - - | 68 |
| CLC | Clear carry flag | 2 | 1 | - - - - - - - - - 0 | 92 |
| CLD | Clear direction flag | 2 | 1 | - - 0 - - - - - - | 92 |
| CLI | Clear interrupt flag | 2 | 1 | - - 0 - - - - - - | 92 |
| CMC | Complement carry flag | 2 | 1 | - - - - - - - - - x | 92 |
| CMP | Compare | | | x - - - x x x x x | 69 |
| | Register to register | 3 | 2 | | |
| | Memory to register | 9+EA | 2-4 | | |
| | Register to memory | 9+EA | 2-4 | | |
| | Immediate to register | 4 | 3-4 | | |
| | Immediate to memory | 10+EA | 3-6 | | |

TABLE A-1 (continued)

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|--------------------------------|-------------------|-----------------|-------------------|-----------|
| | | | | O D I T S Z A P C | |
| CMPS/ | Immediate to accumulator | 4 | 2-3 | | |
| CMPSB/ | Compare string/ | | 1 | x - - - x x x x x | 208 |
| CMPSW/ | Compare byte string/ | | | | |
| CMPSW | Compare word string | | | | |
| | Not repeated | 22 | | | |
| | Repeated | 9 + 22/rep | | | |
| CWD | Convert word to double word | 5 | 1 | - - - - - | 68 |
| DAA | Decimal adjust for addition | 4 | 1 | u - - - x x x x x | 74 |
| DAS | Decimal adjust for subtraction | 4 | 1 | u - - - x x x x x | 74 |
| DEC | Decrement by 1 | | | x - - - x x x x - | 69 |
| | 16-bit register | 2 | 1 | | |
| | 8-bit register | 3 | 2 | | |
| | Memory | 15 + EA | 2-4 | | |
| DIV | Unsigned division | | | u - - - u u u u u | 70 |
| | 8-bit register | 80-90 | 2 | | |
| | 16-bit register | 144-162 | 2 | | |
| | 8-bit memory | (86-96) + EA | 2-4 | | |
| | 16-bit memory | (150-168) + EA | 2-4 | | |
| ESC | Escape | | | - - - - - | 457 |
| | Register | 2 | 2 | | |
| | Memory | 8 + EA | 2-4 | | |
| HLT | Halt | 2 | 1 | - - - - - | 91 |
| IDIV | Integer division | | | u - - - u u u u u | 70 |
| | 8-bit register | 101-112 | 2 | | |
| | 16-bit register | 165-184 | 2 | | |
| | 8-bit memory | (107-118) + EA | 2-4 | | |
| | 16-bit memory | (171-190) + EA | 2-4 | | |
| IMUL | Integer multiplication | | | x - - - u u u u x | 70 |
| | 8-bit register | 80-98 | 2 | | |
| | 16-bit register | 128-154 | 2 | | |
| | 8-bit memory | (86-104) + EA | 2-4 | | |
| | 16-bit memory | (134-160) + EA | 2-4 | | |
| IN | Input from I/O port | | | - - - - - | 232 |
| | Fixed port | 10 | 2 | | |
| | Variable port | 8 | 1 | | |
| INC | Increment by 1 | | | x - - - x x x x - | 69 |
| | 16-bit register | 2 | 1 | | |
| | 8-bit register | 3 | 2 | | |
| | Memory | 15 + EA | 2-4 | | |
| INT | Interrupt | | | - - 0 0 - - - - - | 172 |

TABLE A-1 (continued)

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|--|--------------|-----------------|-------------------|-----------|
| | | | | O D I T S Z A P C | |
| INTO | Type = 3 | 52 | 1 | ----00----- | 172 |
| | Type ≠ 3 | 51 | 2 | | |
| | Interrupt if overflow | | 1 | | |
| | Interrupt is taken | 53 | | | |
| IRET | Interrupt is not taken | 4 | | rrrrrrrrrrrrrr | 172 |
| | Return from interrupt | 24 | 1 | | |
| | JA/ | 16/4 | 2 | | |
| | JNB/ | | | | |
| JNBE | Jump if not below or equal | | | ----- | 80 |
| JAE/ | Jump if above or equal/ | 16/4 | 2 | | |
| JNC | Jump if not carry | | | | |
| JB/ | Jump if below/ | 16/4 | 2 | | |
| JNAE/ | Jump if not above or equal/ | | | ----- | 80 |
| JC | jump if carry | | | | |
| JBE/ | Jump if below or equal/ | 16/4 | 2 | | |
| JNA | Jump if not above | | | | |
| JCXZ | Jump if CX is zero | 18/6 | 2 | ----- | 88 |
| JE/ | Jump if equal/ | 16/4 | 2 | | |
| JZ | Jump if zero | | | | |
| JG/ | Jump if greater/ | 16/4 | 2 | | |
| JNLE | Jump if not less or equal | | | ----- | 80 |
| JGE/ | Jump if greater or equal/ | 16/4 | 2 | | |
| JNL | Jump if not less | | | | |
| JL/ | Jump if less/ | 16/4 | 2 | | |
| JNGE | Jump if not greater or equal | | | ----- | 80 |
| JLE/ | Jump if less or equal/ | 16/4 | 2 | | |
| JNG | Jump if not greater | | | | |
| JMP | Jump | | | | |
| | Intrasegment direct short | 15 | 2 | ----- | 84 |
| | Intrasegment direct | 15 | 3 | | |
| | Intersegment direct | 15 | 5 | | |
| | Intrasegment indirect through memory | 18 + EA | 2-4 | | |
| | Intrasegment indirect through register | 11 | 2 | ----- | 80 |
| | Intersegment indirect | 24 + EA | 2-4 | | |
| JNE/ | Jump if not equal/ | 16/4 | 2 | | |
| JNZ | Jump if not zero | | | ----- | 80 |
| JNO | Jump if not overflow | 16/4 | 2 | | |
| JNP/ | Jump if not parity/ | 16/4 | 2 | | |
| JPO | Jump if parity odd | | | ----- | 80 |
| JNS | Jump if not sign | 16/4 | 2 | | |
| JO | Jump if overflow | 16/4 | 2 | | |
| JP/ | Jump if parity/ | 16/4 | 2 | | |
| JPE | Jump if parity even | | | ----- | 80 |
| JS | Jump if sign | 16/4 | 2 | | |
| LAHF | Load AH from flags | 4 | 1 | | 92 |
| LDS | Load pointer using DS | 16 + EA | 2-4 | | |
| LEA | Load effective address | 2 + EA | 2-4 | ----- | 60 |

TABLE A-1 (continued)

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|------------------------------|-------------------|-----------------|-------------------|-----------|
| | | | | O D I T S Z A P C | |
| LES | Load pointer using ES | 16 + EA | 2-4 | - - - - - | 60 |
| LOCK | Lock bus | 2 | 1 | - - - - - | 455 |
| LODS/ | Load string/ | | 1 | - - - - - | 208 |
| LODSB/ | Load byte string/ | | | - - - - - | |
| LODSW | Load word string | | | - - - - - | |
| | Not repeated | 12 | | | |
| | Repeated | 9 + 13/re | | | |
| LOOP | Loop | 17/5 | 2 | - - - - - | 88 |
| LOOPE/ | Loop if equal/ | 18/6 | 2 | - - - - - | 88 |
| LOOPZ | Loop if zero | | | - - - - - | |
| LOOPNE/ | Loop if not equal/ | 19/5 | 2 | - - - - - | 88 |
| LOOPNZ | Loop if not zero | | | - - - - - | |
| MOV | Move | | | - - - - - | 60 |
| | Accumulator to memory | 10 | 3 | | |
| | Memory to accumulator | 10 | 3 | | |
| | Register to register | 2 | 2 | | |
| | Memory to register | 8 + EA | 2-4 | | |
| | Register to memory | 9 + EA | 2-4 | | |
| | Immediate to register | 4 | 2-3 | | |
| | Immediate to memory | 10 + EA | 3-6 | | |
| | Register to SS, DS, or ES | 2 | 2 | | |
| | Memory to SS, DS, or ES | 8 + EA | 2-4 | | |
| | Segment register to register | 2 | 2 | | |
| | Segment register to memory | 9 + EA | 2-4 | | |
| MOVS/ | Move string/ | | 1 | - - - - - | 208 |
| MOVSB/ | Move byte string/ | | | - - - - - | |
| MOVSW | Move word string | | | - - - - - | |
| | Not repeated | 18 | | | |
| | Repeated | 9 + 17/rep | | | |
| MUL | Unsigned multiplication | | | x - - - u u u u x | 70 |
| | 8-bit register | 70-77 | 2 | | |
| | 16-bit register | 118-133 | 2 | | |
| | 8-bit memory | (76-83) + EA | 2-4 | | |
| | 16-bit memory | (124-139) + EA | 2-4 | | |
| NEG | Negate | | | x - - - x x x x x | 69 |
| | Register | 3 | 2 | | |
| | Memory | 16 + EA | 2-4 | | |
| NOP | No operation | 3 | 1 | - - - - - | 91 |
| NOT | Logical NOT | | | - - - - - | 93 |
| | Register | 3 | 2 | | |
| | Memory | 16 + EA | 2-4 | | |
| OR | Logical OR | | | 0 - - - x x u x 0 | 93 |
| | Register to register | 3 | 2 | | |
| | Memory to register | 9 + EA | 2-4 | | |
| | Register to memory | 16 + EA | 2-4 | | |

TABLE A-1 (continued)

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|--------------------------------------|--------------------|-----------------|---------------------|-----------|
| | | | | O D I T S Z A P C | |
| OUT | Immediate to accumulator | 4 | 2-3 | | |
| | Immediate to register | 4 | 3-4 | | |
| | Immediate to memory | 17 + EA | 3-6 | | |
| | Output to I/O port | | | - - - - - | 232 |
| POP | Fixed port | 10 | 2 | | |
| | Variable port | 8 | 1 | | |
| | Pop word off stack | | | - - - - - | 153 |
| POPF | Register | 8 | 1 | | |
| | Segment register SS, DS, or ES | 8 | 1 | | |
| | Memory | 17 + EA | 2-4 | | |
| | Pop flags off stack | 8 | 1 | r r r r r r r r r r | 153 |
| PUSH | Push word onto stack | | | - - - - - | 153 |
| | Register | 11 | 1 | | |
| | Segment register | 10 | 1 | | |
| PUSHF | Memory | 16 + EA | 2-4 | | |
| | Push flags onto stack | 10 | 1 | - - - - - | 153 |
| RCL | Rotate left through carry | | | x - - - - - x | 98 |
| | Register with single-shift | 2 | 2 | | |
| | Register with variable- shift | 8 + 4/bit | 2 | | |
| | Memory with single-shift | 15 + EA | 2-4 | | |
| | Memory with variable- shift | 20 + EA + 4/bit | 2-4 | | |
| | | | | x - - - - - x | 98 |
| RCR | Rotate right through carry | | | | |
| | Register with single-shift | 2 | 2 | | |
| | Register with variable- shift | 8 + 4/bit | 2 | | |
| | Memory with single-shift | 15 + EA | 2-4 | | |
| | Memory with variable- shift | 20 + EA + 4/bit | 2-4 | | |
| | | | | x - - - - - x | 98 |
| REP | Repeat string operation | 2 | 1 | - - - - - | 213 |
| REPE/ | Repeat operation while equal/ | 2 | 1 | - - - - - | 213 |
| REPZ | Repeat operation while zero | | | | |
| REPNE/ | Repeat operation while not equal/ | | | | |
| REPNZ | Repeat operation while not zero | 2 | 1 | - - - - - | 213 |
| RET | Return from procedure | | | | |
| | Intrasegment | 8 | 1 | | |
| | Intrasegment with con- stant | 12 | 3 | | |
| | Intersegment | 18 | 1 | | |
| | Intersegment with con- stant | 17 | 3 | | |
| ROL | Rotate left | | | x - - - - - x | 98 |
| | Register with single-shift | 2 | 2 | | |

TABLE A-1 (continued)

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|------------------------------|-----------------------------|-----------------|-------------------|-----------|
| | | | | O D I T S Z A P C | |
| ROR | Register with variable-shift | $8 + 4/\text{bit}$ | 2 | | |
| | Memory with single-shift | $15 + \text{EA}$ | 2-4 | | |
| | Memory with variable-shift | $20 + \text{EA}$ + 4/bit | 2-4 | | |
| | Rotate right | | | x - - - - - x | 98 |
| | Register with single-shift | 2 | 2 | | |
| | Register with variable-shift | $8 + 4/\text{bit}$ | 2 | | |
| | Memory with single-shift | $15 + \text{EA}$ | 2-4 | | |
| SAHF | Memory with variable-shift | $20 + \text{EA}$ + 4/bit | 2-4 | | |
| | Store AH into flags | 4 | 1 | - - - - r r r r r | 92 |
| | Shift arithmetic left/ | | | x - - - x x u x x | 98 |
| | Shift logical left | | | | |
| | Register with single-shift | 2 | 2 | | |
| | Register with variable-shift | $8 + 4/\text{bit}$ | 2 | | |
| | Memory with single-shift | $15 + \text{EA}$ | 2-4 | | |
| SAL/ | Memory with variable-shift | $20 + \text{EA}$ + 4/bit | 2-4 | | |
| | SAR | Shift arithmetic right | | x - - - x x u x x | 98 |
| | Register with single-shift | 2 | 2 | | |
| | Register with variable-shift | $8 + 4/\text{bit}$ | 2 | | |
| | Memory with single-shift | $15 + \text{EA}$ | 2-4 | | |
| | Memory with variable-shift | $20 + \text{EA}$ + 4/bit | 2-4 | | |
| | SBB | Subtract with borrow | | x - - - x x x x x | 65 |
| SCAS/ | Register from register | 3 | 2 | | |
| | Memory from register | $9 + \text{EA}$ | 2-4 | | |
| | Register from memory | $16 + \text{EA}$ | 2-4 | | |
| | Immediate from accumulator | 4 | 2-3 | | |
| | Immediate from register | 4 | 3-4 | | |
| | Immediate from memory | $17 + \text{EA}$ | 3-6 | | |
| | SCASB/ | Scan string/ | 1 | x - - - x x x x x | 208 |
| SCASW | Scan byte string/ | | | | |
| | Scan word string | | | | |
| | Not repeated | 15 | | | |
| | Repeated | $9 + 15/\text{rep}$ | | | |
| | SHR | Shift logical right | | x - - - x x u x x | 98 |
| | Register with single-shift | 2 | 2 | | |
| | Register with variable-shift | $8 + 4/\text{bit}$ | 2 | | |
| STC | Memory with single-shift | $15 + \text{EA}$ | 2-4 | | |
| | Memory with variable-shift | $20 + \text{EA}$ + 4/bit | 2-4 | | |
| | Set carry flag | 2 | 1 | - - - - - - - - 1 | 92 |
| | STD | Set direction flag | 2 | - 1 - - - - - - | 92 |

TABLE A-1 (continued)

| Mnemonic | Description | Clock cycles | Number of bytes | Flags | Page ref. |
|----------|----------------------------------|--------------|-----------------|---------------------|-----------|
| | | | | O D I T S Z A P C | |
| STI | Set interrupt flag | 2 | 1 | - - 1 - - - - | 92 |
| STOS/ | Store string/ | | 1 | - - - - - - - - | 208 |
| STOSB/ | Store byte string/ | | | | |
| STOSW | Store word string | | | | |
| | Not repeated | 11 | | | |
| | Repeated | 9 + 10/rep | | | |
| SUB | Subtraction | | | x - - - x x x x x x | 65 |
| | Register from register | 3 | 2 | | |
| | Memory from register | 9 + EA | 2-4 | | |
| | Register from memory | 16 + EA | 2-4 | | |
| | Immediate from accumulator | 4 | 2-3 | | |
| | Immediate from register | 4 | 3-4 | | |
| | Immediate from memory | 17 + EA | 3-6 | | |
| TEST | Test | | | 0 - - - x x u x 0 | 93 |
| | Register with register | 3 | 2 | | |
| | Memory with register | 9 + EA | 2-4 | | |
| | Immediate with accumulator | 4 | 2-3 | | |
| | Immediate with register | 5 | 3-4 | | |
| | Immediate with memory | 11 + EA | 3-6 | | |
| WAIT | Wait while TEST pin not asserted | 3 + 5n | 1 | - - - - - - - - | 457 |
| XCHG | Exchange | | | - - - - - - - - | 60 |
| | Register with accumulator | 3 | 1 | | |
| | Register with memory | 17 + EA | 2-4 | | |
| | Register with register | 4 | 2 | | |
| XLAT/ | Translate | 11 | 1 | - - - - - - - - | 221 |
| XLATB | | | | | |
| XOR | Logical exclusive OR | | | 0 - - - x x u x 0 | 93 |
| | Register with register | 3 | 2 | | |
| | Memory with register | 9 + EA | 2-4 | | |
| | Register with memory | 16 + EA | 2-4 | | |
| | Immediate with accumulator | 4 | 2-3 | | |
| | Immediate with register | 4 | 3-4 | | |
| | Immediate with memory | 17 + EA | 3-6 | | |

| EA | No. of Clock Cycles |
|---|---------------------|
| Direct | 6 |
| Register indirect | 5 |
| Register relative | 9 |
| Based indexed (BP)+(DI) or (BX)+(SI) (BP)+(SI) or (BX)+(DI) | 7 8 |
| Based indexed relative (BP)+(DI)+DISP or (BX)+(SI)+DISP or (BP)+(SI)+DISP or (BX)+(DI)+DISP | 11 12 |

Index

A

Absolute memory reference, 147

Address, 2, 14–16

base, 19

beginning segment, 29

decoder, 347–48

logical, 297

physical, 28, 297

return, 17

segment, 29

space, 14

virtual, 297

Addressing mode, 35 (*See also* 8086/8088, addressing modes)

Algorithms:

best fit, 293

first fit, 293

first in/first out (FIFO), 299

least frequently used (LFU), 299

least recently used (LRU), 299

Alias, 592–93

Alphanumeric code, 6, 11–14

ASCII, 11–12

EBCDIC, 11

American Standard Code for Information Interchange (ASCII)
(*See* Alphanumeric code, ASCII)

Analog-to-digital (A/D), 374–75, 383

Applications:

high-end, 1

low-end, 1

Architecture, 2–3 (*See also* Intel devices)

ASM-89, 514–15 (*See also* 8089 I/O processor)

ASM-86, 54, 56, 179, 482, 485

combine-types, 145–47

AT, 145, 147

COMMON, 145–46

MEMORY, 145, 147

PUBLIC, 145–46

STACK, 145, 147

constant suffixes (B,D,O,H), 56

control functions:

%EXIT, 180–81

%IF, 179–81

%LEN, 181–82

%REPEAT, 179–81

ASM-86 (cont.)
 %SET, 181
 %WHILE, 179–81
 controlled expansion, 179–82
 directives (*See* 8086/8088, directives)
 identifier (*See* Identifier)
 instructions (*See* 8086/8088, instructions)
 macro facilities, 174–76
 macros (*See* Macro)
 operand (*See* Operand)
 operators:
 arithmetic, 56
 attribute, 57–58
 BYTE, 107
 DWORD, 107
 FAR, 84, 160
 LENGTH, 118–19
 NEAR, 84, 160
 OFFSET, 119
 SEG, 119
 SHORT, 84
 SIZE, 119
 TYPE, 119
 WORD, 107
 duplication (DUP), 103–4
 logical, 56
 MASK, 111
 PTR, 58, 106–7
 WIDTH, 111
 prefixes (*See* 8086/8088, prefixes)
Assembler, 5, 120–28
Assembly process, 120–28
Attribute operators (*See* ASM-86, operators, attribute)

B

Backward reference, 125
Base, 6
Battery, 443–44
 ampere-hour rating, 444
 discharge current rating, 444

Baud, 355
 rate factor, 363
Bit, 6
 manipulation (*See also* 8086/8088, instructions, logical)
 clearing, 95
 complementing, 95
 setting, 94
 testing, 95
 parity, 352
 start, 352
 stop, 352
Bits per second (bps), 355
Block structure, high-level language, 528
Block transfers, 230–31, 251–58 (*See also* Direct memory access; Input/output)
Branch:
 conditional, 16–18
 unconditional, 16–18
Breakpoint, 173
Buffering, 234
 double, 234, 245
 multiple, 234
 triple, 234, 258–59
Bus, 3, 308
 access logic, 464
 address, 3
 arbiter, 342, 455–56, 471–72, 496 (*See also* Intel devices, 8289 bus arbiter)
 arbitration, 464
 contention, 451, 453
 control, 3
 control logic, 3, 308–9, 451–53, 458, 460, 463–64, 471
 cycle, 251, 310
 data, 3
 drivers, 347–48
 external, 308
 grant, 251, 319, 322, 326–27, 460–62, 464
 internal, 308
 master, 340–42

receivers, 347–48
request, 251, 319, 322, 326–27, 455,
 459–62, 464
slave, 340–42
standard, 310, 340
system (*See* System bus)
timing (*See* System bus, timing)
utilization, 470–71
Byte, 14

C

Central processing unit (CPU), 2–3
Channel program, 496
Checksum, 407
Clock, 2, 310
 cycle, 310
 generator, 317
Code length, 13
Collating sequence, 13
Comité Consultatif Internationale Téléphonique et Télégraphique (CCITT), 355
Comments, 101 (*See also* 8086/8088, instruction formats, assembler)
Communication protocol, 477
Communication standards, 354–61
 CCITT V.24, 356, 359
 EIA RS-232-C, 355–60
 20-mA loop, 361
Compiler, 5
Completion routine, 256
Conditional assembly, 179 (*See also* ASM-86, controlled expansion)
Constant, 56 (*See also* 8086/8088, instruction formats, assembler)
 expression, 104
 suffixes (*See* ASM-86, constant suffixes)
Constructs, elementary, 186
Contact bounce, 385
Control functions (*See* ASM-86, control functions)
Control register, 347

Coprocessor, 456–63, 580
Coupling:
 control, 142, 183
 data, 142, 183
Critical section of code, 283
Cross reference symbol table, 127–28
Current task, 551
Cycle-stealing, 251, 457, 459
Cyclic redundancy check (CRC), 407

D

Daisy chain, 247–48, 347–48, 464–65,
 468–70
Data buffer register, 347
Data set, 357
Debouncing, 385
Denormal, 481, 485, 491
Digit, 6
Digital-to-analog (D/A), 374–75
Directives, 53 (*See also* 8086/8088, directives)
Direct memory access (DMA), 231,
 251–58 (*See also* Block transfers; Input/output)
Diskette, 402
 controller, 402–13
 cylinder, 404
 format, 404–7
 hard sectored, 405
 IBM 3740, 406–7
 load time, 406
 mini-, 404
 position time, 406
 rotational time, 406
 sector, 404
 soft sectored, 404
 track, 404
Displacement, 28 (*See also* Offset)
Display design, 385–87
DMA controller, 231, 251–58, 316, 347,
 395–402, 450, 519, 524, 526 (*See also* Direct memory access)

Do bit, 255
 Done bit, 256
 Double-precision arithmetic, 66
 Dynamic address translation, 297

E

Echoing, 351
 Effective address (EA), 28 (*See also*
 Offset)
 8089 I/O processor (IOP), 453, 460–
 63, 475, 496–516, 520
 architecture, 499–402
 channel attention (CA), 499
 instructions, 510–15
 operation, 502–10
 8087 numeric data processor (NDP),
 453, 456, 460, 462–63, 477–96,
 520
 architecture, 482–85
 instructions, 485–93
 number formats, 479–82
 long integer, 479–80
 long real, 479–82
 packed BCD, 479–80, 482
 short integer, 479–80
 short real, 479–81
 temporary real, 479–82
 word integer, 479–80
 8086/8088 (*See also* ASM-86):
 addressing modes, 35–39
 branch-related, 38–39
 data-related, 35–37
 fields (MOD and R/M), 42–43
 architecture, 26–33
 directives, 100–118
 ASSUME, 114–16
 data definition (DB,DW,DD,
 DQ,DT), 101–7, 482 (*See also*
 Preassignment)
 END, 116–17
 ENDP, 159–60
 ENDS, 108, 113–14

EQU, 112–13
 EVEN, 117–18
 EXTRN, 148
 LABEL, 107
 operand (*See* Operand, directive)
 ORG, 118
 PROC, 158–60
 PUBLIC, 148
 RECORD, 111
 SEGMENT, 113–16
 storage allocation (*See* 8086/8088,
 directives, data definition)
 STRUC, 108
 indicator bits (W,D,S,V,Z), 41
 instruction formats, 39–47
 assembler, 55, 57
 machine, 39–47, 129–34
 instruction queue, 27, 33, 50
 instructions, 536–43
 arithmetic, 63–78
 binary, 65–73
 packed BCD, 73–75
 unpacked BCD, 75–78
 branch, 78–86
 conditional, 78–82
 intersegment, 84
 intrasegment, 84
 unconditional, 82–86
 data transfer, 59–63
 escape (ESC), 457
 flag manipulation, 92–93
 halt (HLT), 91–92
 interrupt, 171–73
 return, 172
 logical, 93–97
 loop, 86–90
 no operation (no op–NOP), 90–
 92
 procedure call/return, 157–60
 rotate, 97–100
 shift, 97–100
 stack, 153
 string manipulation, 93, 208–12
 WAIT, 457, 484–86

- XLAT, 220–21
internal operation, 33–35
maximum mode, 319–24, 349, 413–15, 451
minimum mode, 314–19, 349, 397–98, 413
pin assignments/definitions, 26, 311–12, 314, 320
prefixes:
 LOCK, 320, 322, 455–56
 override, 42, 116
 REP, 212–14
register addresses, 42
registers:
 data group (AX,BX,CX,DX), 27
 index group (SI,DI), 28
 instruction (IR) (*See* 8086/8088, instruction queue)
 pointer group (IP,SP,BP), 28, 153–55
 processor status word (PSW), 31–33, (*See also* Flags)
 segment group (CS,SS,DS,ES), 28, 57, 153–55
80186:
 architecture, 534
 control block, 533–35
 DMA, 535–37
 interrupt, 535, 540–42
 peripheral select, 535, 542–44
 timer, 535, 537–539
 difference from 8086, 524
instructions:
 BOUND, 526–28
 ENTER, 526, 528–31, 567
 IMUL, 525
 INS, 526–27
 LEAVE, 526, 529–31, 567
 OUTS, 526–27
 PUSH, 525
 POP, 525
 POPA, 525–26
 PUSHA, 525–26
 shift, 525
LOCK signal, 524
pin assignments/definitions, 544–47
prefixes, 524
 REP (*See also* 8086 prefixes), 526
relocation register, 533
80286, 300–305, 520, 551
 architecture, 553–54
bus cycle status, 597
cache, 556
descriptors, 557–58, 573–74
descriptor table:
 global (GDT), 557
 interrupt (IDT), 583
 local (LDT), 557
descriptor table register:
 global (GDTR), 559
 interrupt (IDTR), 583
 local (LDTR), 560
gate:
 call, 564–65
 interrupt, 586–87
 task, 576, 578
 trap, 586–87
instructions, 556
 ARPL, 570–71
 CALL FAR, 561–64, 582–83
 CALL Gatename, 565–68, 582–83
 CALL Taskname, 575–77, 582–83
 CLTS, 580, 583
 IRET, 579, 583, 588
 JMP FAR, 561–64, 582–83
 JMP Gatename, 565–66, 582–83
 JMP Taskname, 575–77, 582–83
 LAR and LSL, 570–72
 loading and storing system registers, 582, 593
 privileged, 580, 582, 593
 RET, 561–62, 568–69, 582
 trusted, 590
 VERR and VERW, 570, 572
memory reference, 557
pin assignment/definition, 595–96
privilege level, 552–53

80286 (*cont.*)
 current (CPL), 560, 563–65, 576, 578–79, 589–90
 descriptor (DPL), 557–58, 561, 563, 565, 574, 576, 578–79, 589
 effective (EPL), 563, 582–83
 input/output (IOPL), 590
 requested (RPL), 559–61, 563, 565, 576
 registers, 554–55
 reset state, 592
 segment register, 556
 selector, 556, 559
 stacks, 567
 task state segment (TSS), 572–73
 timing, bus cycle, 597–601
 word count, 565, 568
 Electrical Industry Association (EIA), 355
Error:
 division, 170–71
 fatal, 264
 framing, 353
 overflow, 173
 overrun, 349, 353
 parity, 349, 353
 underrun, 354
 Event counter, 378–79
 Exceptions, 583–86
 Execution time, 47
 Exit, 186
 Expression, 56 (*See also* 8086/8088, instruction formats, assembler)
 Extended Binary Coded Decimal Interchange Code (EBCDIC) (*See* Alphanumeric code, EBCDIC)
 External symbol table, 149

F

File management routine, 5
 First in/first out (FIFO) policy, 275
 Flags (*See also* 8086/8088, registers, PSW):

condition (CF,PF,AF,ZF,SF,OF), 31–32
 control (TF,IF,DF), 31–33, 169–71
 programming, 188
 Floppy disk, 402 (*See also* Diskette)
 Forward reference, 125
 Fragmentation, 294
 Frame, 167
 Full duplex, 350–51

H

Half duplex, 351
 Handshaking logic, 347–48
 Hardware, 2–4
 Hierarchical diagram, 142, 184
 Horner's rule, 8, 223

I

Identifier, 53, 56 (*See also* ASM-86; 8086/8088, instructions)
 constant name, 112
 expression name, 112
 external (global), 147–49
 field, 108
 label (*See Label*)
 local (internal), 147–49
 name (*See Name*)
 pathname, 110
 procedure name, 159–60
 segment name, 113–14
 variable, 101, 108
 Idle character, 354
 Index register (*See Register, index*)
 Indicator bits (*See* 8086/8088, indicator bits)
 Initialization command word (ICW), 330–33 (*See also* Intel devices, 8259A)
 Input/output (I/O):
 driver, 5, 235

- handler, 235
interrupt, 230–31, 240–51
memory-mapped (*See Memory-mapped I/O*)
port, 4, 229
programmed, 230–31, 236–40
read command (IORC), 322
subsystem, 2
write command (IOWC), 322
 advanced, 323
Instruction pointer (IP), (*See 8086/8088, registers, pointer group*)
Instruction queue (*See 8086/8088, instruction queue*)
Instruction register (IR), 16–17 (*See also 8086/8088, instruction queue*)
Instructions, 53 (*See also 8086/8088, instructions*)
Intel devices:
 8089 (*See 8089 I/O processor*)
 8087 (*See 8087 numeric data processor*)
 8086/8088 (*See 8086/8088*)
 80186 (*See 80186*)
 80130, 282, 520–23
 80386, 520
 80287, 602–4 (*See also 8087 numeric data processor*)
 80286 (*See 80286*)
 8288 bus controller, 320–23, 453, 456, 544
 8284A clock, 317, 319
 8289 bus arbiter, 466–70, 472–75, 544
 8286/8287 transceivers, 317–18, 544
 8282/8283 latch, 315–17, 544
 8255A programmable peripheral interface (PPI), 371–74, 384, 415–17, 419
 8254 programmable interval timer, 380–84
 8259A priority interrupt controller (PIC), 329–39, 458–59
 8251A programmable communication interface (PCI), 361–69
 8279 keyboard/display controller, 387–95
 8272 diskette controller, 408–13, 515
 8237 DMA controller, 396–402, 416, 418
 82288 bus controller, 595–98
 82284 clock, 595, 598
 82289 bus arbiter, 595–98
Interface, 3–4
Interpreter, 5
Interprocessor communication, 451
Interrupt, 169–73, 231, 240–50, 539–42, 583–91
 acknowledge, 319, 322, 326–27, 329–32, 455
 external, 169, 241, 584
 internal, 169
 maskable, 93, 172–73, 241
 nonmaskable, 172, 241
 pointer (vector), 170, 241, 583
 priority, 231, 247–49, 329
 request, 319, 326, 329–32, 455, 521
 routine, 169, 231, 235, 241, 585–89
 sequence, 169, 240–41, 586, 591
 software, 172–73, 583–84
 type, 170–71, 583–84
Intersegment branch, 83 (*See also 8086/8088, instructions, branch*)
Intrasegment branch, 82 (*See also 8086/8088, instructions, branch*)
iRMX 86, 278–82, 520, 525
ISIS-II operating system, 447

K

Keyboard design, 385

L

Label, 55 (*See also 8086/8088, Instruction formats, assembler*)

- Language:
 assembler, 5
 high-level, 5
 machine, 5
- Library, 5
- Light-emitting diode (LED), 385–87
- Linkage editor (*See* Linker)
- Linked list, 193–96, 275–77
- Linker, 6, 143–45
- Linking, 143–51 (*See also* Linker)
- Loader, 6, 143–45 (*See also* Linker)
- Load module, 143
- Location counter, 121
 symbol (\$), 126
- Locator (LOC-86), 143, 149
- Logical expressions, evaluation of, 96
- Looping, 17 (*See also* 8086/8088, instructions, loop)
- M**
- Macro, 143, 174–79 (*See also* ASM-86)
 call, 174–75
 control functions (*See* ASM-86, control functions)
 definition, 174
 dummy parameter, 174–76
 expansion, 174–76
 local labels, 176–77
 name, 174
 nesting, 177–79
 processor, 176, 179
 prototype code, 174
- Mark state, 352
- Mask, 94
- Masking operation, 94
- Mask storage, 2–3, 251, 395, 402, 423
- Maximum mode (*See* 8086/8088, maximum mode)
- Memory, 423
 access time, 427, 429
 address setup time, 431–32
 backup power, 442–44
 buffer, 234
- chip enable logic, 432–34
- core, 424
- dual-port, 475
- main, 423–27
- management, 291–300, 520, 530
- map, 143
- module, 424–26
- nonvolatile, 423
- protection, 299
- random access, 424
- read command (MRDC), 323
- read cycle time, 431
- read only (ROM), 424, 444–48
 electrically erasable (E²ROM), 445, 448
 erasable programmable (EPROM), 445, 447
 masked, 445
 programmable (PROM), 445
- read recovery time, 431
- read/write (RAM):
 dynamic, 424, 429, 434–42
 static, 424, 427–34, 438
- refresh, 435–37
- semiconductor, 424
- volatile, 423
- write command (MWTC), 323
 advanced, 323
- write cycle time, 431–32
- write pulse width, 431–32
- write recovery time, 431
- Memory-mapped I/O, 417–18, 475
- Microcomputer:
 development system (MDS), 447
 network, 477
- Microelectronic devices:
 LSI, 1
 MSI, 1
 SSI, 1
 VLSI, 1
- Microprocessor unit (MPU), 2 (*See also* Central processing unit)
- Minimum code (*See* 8086/8088, minimum mode)
- Mnemonic, 53, 55, 101

Modem, 357
 Modular programming, 141
 Module description, 184–85
MULTIBUS, 310, 339–42 (*See also Bus*)
 Multiple key closures, 385
 Multiple-precision arithmetic, 66
 Multiprocessing, 450
 configurations:
 closely coupled, 451, 460–63
 coprocessor, 451, 456–60
 loosely coupled, 451–53, 463–77
 system, 274
Multiprogramming, 272, 551
 states, 274–75
 blocked, 274–75
 ready, 275
 running, 274–75
Mutual:
 blocking, 284
 exclusion, 283

N

Name, 56 (*See also* 8086/8088, instruction formats)
 Normalized real number, 480
 Not-a-number (NAN), 481, 485
 Null character, 13
 Number conversion, 8
 Number formats, 6–11 (*See also* 8087
 numeric data processor, number formats)
 binary, 6–10
 binary coded decimal (BCD), 6, 11
 conversions, 223–26 (*See also* Number conversion)
 real (floating point), 6, 478–81
Number system:
 binary, 7
 decimal, 7
 hexadecimal, 7
 octal, 7
 1's complement, 10

sign-magnitude, 9
 10's complement, 11
 2's complement, 9–10

O

Object module, 120
Offset, 19, 28 (*See also* Displacement; Effective address)
Operand, 35, 55–57 (*See also* 8086/8088, instruction formats)
 directive, 101–3
 double-word, 66
 signed, 66
 unsigned, 66
Operating system, 4–5
 processor (OSP), 520
Operation code (op code), 35
Operation command word (OCW), 330–31, 333–37 (*See also* Intel devices, 8259A)
Override prefix (*See* 8086/8088, prefixes, override)

P

Packing data, 100
 Parallel communication, 369–71
 Parameter, 163
 Parameter table, 163–64
 Partition allocation scheme, 292
 relocation, 294
 Pascal, 528–31
 Permanent symbol table, 122
 Personality card, 447
 Pin assignments (*See* Intel devices)
 Polling, 231, 247
 round robin, 240
 Power failure, 342
 Preassignment, 102–3, 108–11
 Priority controller, 468–69
 Procedure, 155–69 (*See also* 8086/8088, instructions)

Procedure (*cont.*)
 common, 287–91
 communication, 162–65
 procedure call (*See* 8086/8088, instructions, procedure call/return)
 recursive, 165–69
 reentrant, 288
 Process, 272
 management, 274–82
 Processor status word (PSW), 16–17
 Program counter (PC), 16–17
 Programmable timer, 378–79
 Program module, 141
 PROM programmer, 447
 Pure code, 288

Q

Queue:
 ready, 275–78
 pointer:
 backward (bqp), 278
 forward (fqp), 277
 status, 320, 453–54, 458, 460

R

Real-time clock, 275 (*See also* Programmable timer)
 Records, 111
 Register (*See also* 8086/8088, registers):
 base, 19
 index, 19
 instruction, 16–17
 Relocatable memory reference, 147
 Relocation, 149
 Resident monitor, 5 (*See also* Operating system)
 Return address (*See* Address, return)
 Rotating (*See* 8086/8088, instructions, rotate)

S

Segmentation, 298
 Segment definition (*See* 8086/8088, directives, SEGMENT)
 Segment fault, 300
 Segment register override prefix (*See* 8086/8088, prefixes, override)
 Segments, overlapping, 31, 115–16
 Semaphore, 285, 454
 P operator, 285
 V operator, 285
 Sentinel, 194
 Serial communication, 349–61
 asynchronous, 351–54
 standard (*See* Communication standards)
 synchronous, 351, 354
 Serially reusable resource, 282, 287
 Shifting (*See* 8086/8088, instructions, shift)
 Sign extension, 41
 Single-precision arithmetic, 66
 16-bit interface design, 414–19
 Software, 2, 4–6
 system, 4
 user, 4
 Sort:
 bubble, 89
 insertion, 89, 138
 Source module, 120
 Space state, 352
 Stack, 18, 28, 145, 147, 151–55
 base of, 153
 last in/first out (LIFO), 152
 pointer (SP), 19, 28, 153–55
 popping the, 153
 pushing the, 153
 top of the (TOS), 153
 Status register, 347
 String constant, 56 (*See also* 8086/8088, instruction formats, assembler)
 Structures, 107–10
 array of, 110
 invocation of, 108–10

Subroutine, 17–19, 155 (*See also Procedure*)
Symbol (identifier) table, 121
Sync character, 354
System bus, 3, 308
 interface, 308
 CPU, 308
 I/O, 346–47
 memory, 424–25
states, 324
 idle, 324
 wait, 324
timing, 324–29
System call, 278, 520
System tables, 250–51
System throughput, 273

T

Table translation, 220–23
Task switching, 552, 572–75
Texas Instruments TIL 311 display, 387–89
Text editor, 5
Text processing, 207
Time-shared system, 274
Time-slice, 274

Top down design, 183
Transceivers, 348 (*See also Intel devices, 8286/8287 transceivers*)
Translator, 5

U

Uniprogramming, 272
Universal Asynchronous Receiver and Transmitter (UART), 353
Unpacking data, 138

V

Variable, 55, 101, 108 (*See also 8086/8088*), instruction formats, assembler)
Virtual space, 297

W

Word, 14
 memory, 423

You'll be interested in
these other

**Eastern
Economy
Editions**

**THE 8088 AND 8086 MICROPROCESSORS: Programming, Interfacing,
Software, Hardware, and Applications**

by Walter A. Triebel, *Intel Corporation*, and Avtar Singh, *San Jose State University*.

This is a thorough study of the 8088 and 8086 microprocessor, their microcomputer system architectures and the circuitry used in the design of the microcomputer of the original IBM PC. Using a systems approach, the text's first part describes how to write, execute, and debug assembly language programs. The second part explains the function and operation of the microprocessors' hardware interfaces: memory, input/output and interrupt.

636 pages

MICROPROCESSORS: Theory and Applications

by M. Rafiquzzaman, *Professor, California State Poly University, Pomona, California; Adjunct Professor, University of Southern California, Los Angeles, California; and President, Rafi Technologies Inc., San Dimas, California.*

This book is revised and updated version of the author's successful text on the subject. It focuses on fundamental concepts and provides indepth treatment of typical 16-bit microprocessors (Intel 8086 and Motorola 68000) and overviews of 32-bit microprocessors such as Intel 80386/80486 and Motorola 68020/68030/68040 as well as 8-bit microprocessors. The text covers, the full range of the latest in microprocessors and microcomputers.

480 pages

0000 to 8085: Introduction to Microprocessor for Engineers and Scientists, 2nd Ed.

by P.K. Ghosh, *Professor, Indian Institute of Technology Kanpur; and P.R. Sridhar.*

In the second edition of this popular book, the 8085 processor and its peripherals have been used to explain the basic concepts of microprocessor operation and system realisation. The text can be used by electrical engineering undergraduates in their first course on microprocessor, and by engineering students in other disciplines and students of science with some preparation in digital electronics.

328 pages

ISBN-81-203-0409-8