Fundamentals of Computer Organization and Design

Sivarama P. Dandamudi
School of Computer Science
Carleton University

September 22, 2002

TEXTS IN COMPUTER SCIENCE

Editors
David Gries
Fred B. Schneider

Springer
New York
Berlin
Heidelberg
Hong Kong
London
Milan
Paris
Tokyo

TEXTS IN COMPUTER SCIENCE

Alagar and Periyasamy, Specification of Software Systems

Apt and Olderog, Verification of Sequential and Concurrent Programs, Second Edition

Back and von Wright, Refinement Calculus

Beidler, Data Structures and Algorithms

Bergin, Data Structure Programming

Brooks, C Programming: The Essentials for Engineers and Scientists

Brooks, Problem Solving with Fortran 90

Dandamudi, Fundamentals of Computer Organization and Design

Dandamudi, Introduction to Assembly Language Programming

Fitting, First-Order Logic and Automated Theorem Proving, Second Edition

Grillmeyer, Exploring Computer Science with Scheme

Homer and Selman, Computability and Complexity Theory

Immerman, Descriptive Complexity

Jalote, An Integrated Approach to Software Engineering, Second Edition

Kizza, Ethical and Social Issues in the Information Age, Second Edition

Kozen, Automata and Computability

(continued after index)

Sivarama P. Dandamudi

FUNDAMENTALS OF COMPUTER ORGANIZATION AND DESIGN

With 361 Illustrations

Springer

Sivarama P. Dandamudi
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada
sivarama@scs.carleton.ca

Series Editors:
David Gries
Department of Computer Science
415 Boyd Graduate Studies
Research Center
The University of Georgia
Athens, GA 30602-7404, USA

Fred B. Schneider
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501, USA

Cover illustration: "Outpouring," by Pam Clocksin; used with permission.

Library of Congress Cataloging-in-Publication Data
Dandamudi, Sivarama P., 1955–
Fundamentals of computer organization and design/Sivarama P. Dandamudi.
p. cm.
Includes bibliographical references and index.
ISBN 0-387-95211-X (hc: alk. paper)
1. Computer engineering. I. Title.
TK7885 .D283 2002
621.39'1—dc21 2002024154

ISBN 0-387-95211-X *Printed on acid-free paper.*

© 2003 Springer-Verlag New York, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 10791386

Typesetting: Pages created by the author using LaTeX.

www.springer-ny.com

Springer-Verlag New York Berlin Heidelberg
A member of BertelsmannSpringer Science+Business Media GmbH

To
my parents, **Subba Rao** and **Prameela Rani**,
my wife, **Sobha**,
and
my daughter, **Veda**

Preface

Computer science and engineering curricula have been evolving at a faster pace to keep up with the developments in the area. This often dictates that traditional courses will have to be compressed to accommodate new courses. In particular, it is no longer possible in these curricula to include separate courses on digital logic, assembly language programming, and computer organization. Often, these three topics are combined into a single course. The current textbooks in the market cater to the old-style curricula in these disciplines, with separate books available on each of these subjects. Most computer organization books do not cover assembly language programming in sufficient detail. There is a definite need to support the courses that combine assembly language programming and computer organization. This is the main motivation for writing this book. It provides a comprehensive coverage of digital logic, assembly language programming, and computer organization.

Intended Use

This book is intended as an undergraduate textbook for computer organization courses offered by computer science and computer engineering/electrical engineering departments. Unlike other textbooks in this area, this book provides extensive coverage of assembly language programming and digital logic. Thus, the book serves the needs of compressed courses.

In addition, it can be used as a text in vocational training courses offered by community colleges. Because of the teach-by-example style used in the book, it is also suitable for self-study by computer professionals and engineers.

Prerequisites

The objective is to support a variety of courses on computer organization in computer science and engineering departments. To satisfy this objective, we assume very little background on the part of the student. The student is assumed to have had some programming experience in a structured, high-level language such as C or Java™. This is the background almost all students in computer science and computer engineering programs typically acquire in their first year of study. This prerequisite also implies that the student has been exposed to the basics of the software-development cycle.

Features

Here is a summary of the special features that set this book apart:

- Most computer organization books assume that the students have done a separate digital logic course before taking the computer organization course. As a result, digital logic is covered in an appendix to provide an overview. This book provides detailed coverage of digital logic, including sequential logic circuit design. Three complete chapters are devoted to digital logic topics, where students are exposed to the practical side with details on several example digital logic chips. There is also information on digital logic simulators. Students can conveniently use these simulators to test their designs.
- This book provides extensive coverage of assembly language programming, comprising assembly language of both CISC and RISC processors. We use the Pentium as the representative of the CISC category and devote more than five chapters to introducing the Pentium assembly language. The MIPS processor is used for RISC assembly language programming. In both cases, students actually write and test working assembly language programs. The book's homepage has instructions on downloading assemblers for both Pentium and MIPS processors.
- We introduce concepts first in simple terms to motivate the reader. Later, we relate these concepts to practical implementations. In the digital logic part, we use several chips to show the type of implementations done in practice. For the other topics, we consistently use three processors—the Pentium, PowerPC, and MIPS—to cover the CISC to RISC range. In addition, we provide details on the Itanium and SPARC processors.
- Most textbooks in the area treat I/O and interrupts as an appendage. As a result, this topic is discussed very briefly. Consequently, students do not get any practical experience on how interrupts work. In contrast, we use the Pentium to illustrate their operation. Several assembly language programs are used to explain the interrupt concepts. We also show how interrupt service routines can be written. For instance, one example in the chapter on interrupts replaces the system-supplied keyboard service routine by our own. By understanding the practical aspects of interrupt processing, students can write their own programs to experiment with interrupts.

- Our coverage of system buses is comprehensive and up-to-date. We divide our coverage into internal and external buses. Internal buses discussed include the ISA, PCI, PCI-X, AGP, and PCMCIA buses. Our external bus coverage includes the EIA-232, SCSI, USB, and IEEE 1394 (FireWire) serial buses.
- Extensive assembly programming examples are used to illustrate the points. A set of input and output routines is provided so that the reader can focus on developing assembly language programs rather than spending time in understanding how input and output can be done using the basic I/O functions provided by the operating system.
- We do not use fragments of assembly language code in examples. All examples are complete in the sense that they can be assembled and run to give a better feeling as to how these programs work.
- All examples used in the textbook and other proprietary I/O software are available from the book's homepage (`www.scs.carleton.ca/~sivarama/org_book`). In addition, this Web site also has instructions on downloading the Pentium and MIPS assemblers to give opportunities for students to perform hands-on assembly programming.
- Most chapters are written in such a way that each chapter can be covered in two or three 60-minute lectures by giving proper reading assignments. Typically, important concepts are emphasized in the lectures while leaving the other material in the book as a reading assignment. Our emphasis on extensive examples facilitates this pedagogical approach.
- Interchapter dependencies are kept to a minimum to offer maximum flexibility to instructors in organizing the material. Each chapter clearly indicates the objectives and provides an overview at the beginning and a summary and key terms at the end.

Instructional Support

The book's Web site has complete chapter-by-chapter slides for instructors. Instructors can use these slides directly in their classes or can modify them to suit their needs. Please contact the author if you want the PowerPoint source of the slides. Copies of these slides (four per page) are also available for distribution to students. In addition, instructors can obtain the solutions manual by contacting the publisher. For more up-to-date details, please see the book's Web page at `www.scs.carleton.ca/~sivarama/org_book`.

Overview and Organization

The book is divided into eight parts. In addition, Appendices provide useful reference material. Part I consists of a single chapter and gives an overview of basic computer organization and design.

Part II presents digital logic design in three chapters—Chapters 2, 3, and 4. Chapter 2 covers the digital logic basics. We introduce the basic concepts and building blocks that we use in the later chapters to build more complex digital circuits such as adders and arithmetic logic units (ALUs). This chapter also discusses the principles of digital logic design using Boolean algebra, Karnaugh maps, and Quine–McCluskey methods. The next chapter deals

with combinational circuits. We present the design of adders, comparators, and ALUs. We also show how programmable logic devices can be used to implement combinational logic circuits. Chapter 4 covers sequential logic circuits. We introduce the concept of time through clock signals. We discuss both latches and flip-flops, including master–slave JK flip-flops. These elements form the basis for designing memories in a later chapter. After presenting some example sequential circuits such as shift registers and counters, we discuss sequential circuit design in detail. These three chapters together cover the digital logic topic comprehensively. The amount of time spent on this part depends on the background of the students.

Part III deals with system interconnection structures. We divide the system buses into internal and external buses. Our classification is based on whether the bus interconnects components that are typically inside a system. Part III consists of Chapter 5 and covers internal system buses. We start this chapter with a discussion of system bus design issues. We discuss both synchronous and asynchronous buses. We also introduce block transfer bus cycles as well as wait states. Bus arbitration schemes are described next. We present five example buses including the ISA, PCI, PCI-X, AGP, and PCMCIA buses. The external buses are covered in Part VIII, which discusses the I/O issues.

Part IV consists of three chapters and discusses processor design issues. Chapter 6 presents the basics of processor organization and performance. We discuss instruction set architectures and instruction set design issues. This chapter also covers microprogrammed control. In addition, processor performance issues, including the SPEC benchmarks, are discussed. The next chapter gives details about the Pentium processor. The information presented in this chapter is useful when we discuss Pentium assembly language programming in Part V. Pipelining and vector processors are discussed in the last chapter of this part. We use the Cray X-MP system to look at the practical side of vector processors. After covering the material in Chapter 6, instructors can choose the material from Chapters 7 and 8 to suit their course requirements.

Part V covers Pentium assembly language programming in detail. There are five chapters in this part. Chapter 9 provides an overview of the Pentium assembly language. All necessary basic features are covered in this chapter. After reading this chapter, students can write simple Pentium assembly programs without needing the information presented in the later four chapters. Chapter 10 describes the Pentium addressing modes in detail. This chapter gives enough information for the student to understand why CISC processors provide complex addressing modes. The next chapter deals with procedures. Our intent is to expose the student to the underlying mechanics involved in procedure calls, parameter passing, and local variable storage. In addition, recursive procedures are used to explore the principles involved in handling recursion. In all these activities, the important role played by the stack is illustrated. Chapter 12 describes the Pentium instruction set. Our goal is not to present the complete Pentium instructions, but a representative sample. Chapter 13 deals with the high-level language interface, which allows mixed-mode programming in more than one language. We use C and assembly language to illustrate the principles involved in mixed-mode programming. Each chapter uses several examples to show how various Pentium instructions are used.

Part VI covers RISC processors in two chapters. The first chapter introduces the general RISC design principles. It also presents details about two RISC processors: the PowerPC and

Intel Itanium. Although both are considered RISC processors, they also have some CISC features. We discuss a pure RISC processor in the next chapter. The Itanium is Intel's 64-bit processor that not only incorporates RISC characteristics but also several advanced architectural features. These features include instruction-level parallelism, predication, and speculative loads. The second chapter in this part describes the MIPS R2000 processor. The MIPS simulator SPIM runs the programs written for the R2000 processor. We present MIPS assembly language programs that are complete and run on the SPIM. The programs we present here are the same programs we have written in the Pentium assembly language (in Part V). Thus, the reader has an opportunity to contrast the two assembly languages.

Part VII consists of Chapters 16 through 18 and covers memory design issues. Chapter 16 builds on the digital logic material presented in Part II. It describes how memory units can be constructed using the basic latches and flip-flops presented in Chapter 4. Memory mapping schemes, both full- and partial-mapping, are also discussed. In addition, we discuss how interleaved memories are designed. The next chapter covers cache memory principles and design issues. We use an extensive set of examples to illustrate the cache principles. Toward the end of the chapter, we look at example cache implementations in the Pentium, PowerPC, and MIPS processors. Chapter 18 discusses virtual memory systems. Note that our coverage of virtual memory is from the computer organization viewpoint. As a result, we do not cover those aspects that are of interest from the operating-system point of view. As with the cache memory, we look at the virtual memory implementations of the Pentium, PowerPC, and MIPS processors.

The last part covers the I/O issues. We cover the basic I/O interface issues in Chapter 19. We start with I/O address mapping and then discuss three techniques often used to interface with I/O devices: programmed I/O, interrupt-driven I/O, and DMA. We discuss interrupt-driven I/O in detail in the next chapter. In addition, this chapter also presents details about external buses. In particular, we cover the EIA-232, USB, and IEEE 1394 serial interfaces and the SCSI parallel interface. The last chapter covers Pentium interrupts in detail. We use programming examples to illustrate interrupt-driven access to I/O devices. We also present an example to show how user-defined interrupt service routines can be written.

The appendices provide a wealth of reference material needed by the student. Appendix A primarily discusses computer arithmetic. Character representation is discussed in Appendix B. Appendix C gives information on the use of I/O routines provided with this book and the Pentium assembler software. The debugging aspect of assembly language programming is discussed in Appendix D. Appendix E gives details on running the Pentium assembly programs on a Linux system using the NASM assembler. Appendix F gives details on digital logic simulators. Details on the MIPS simulator SPIM are in Appendix G. Appendix H describes the SPARC processor architecture. Finally, selected Pentium instructions are given in Appendix I.

Acknowledgments

Several people have contributed to the writing of this book. First and foremost, I would like to thank my wife, Sobha, and my daughter, Veda, for enduring my preoccupation with this project.

I thank Wayne Yuhasz, Executive Editor at Springer-Verlag, for his input and feedback in

developing this project. His guidance and continued support for the project are greatly appreciated. I also want to thank Wayne Wheeler, Assistant Editor, for keeping track of the progress. He has always been prompt in responding to my queries. Thanks are also due to the staff at Springer-Verlag New York, Inc., particularly Francine McNeill, for its efforts in producing this book. I would also like to thank Valerie Greco for doing an excellent job of copyediting the text.

My sincere appreciation goes to the School of Computer Science at Carleton University for allowing me to use part of my sabbatical leave to complete this book.

Feedback

Works of this nature are never error-free, despite the best efforts of the authors and others involved in the project. I welcome your comments, suggestions, and corrections by electronic mail.

Ottawa, Ontario, Canada
December 2001

Sivarama P. Dandamudi
sivarama@scs.carleton.ca
http://www.scs.carleton.ca/~sivarama

Contents

Chapter 1

Overview of Computer Organization

Objectives

- To provide a high-level overview of computer organization;
- To discuss how architects, implementers, programmers, and users view the computer system;
- To describe the three main components: processor, memory, and I/O;
- To give a brief historical perspective of computers.

We begin each chapter with an overview of what you can expect in the chapter. This is our first overview. The main purpose of this chapter is to provide an overview of the computer systems. We start off with a brief introduction to computer systems from the user's viewpoint.

Computer systems are complex. To manage this complexity, we use a series of abstractions. The kind of abstraction used depends on what you want to do with the system. We present the material in this book from three perspectives: from the computer architect's view, from the programmer's view, and from the implementer's view. We give details about these three views in Sections 1.2 through 1.4.

A computer system consists of three major components: a processor, a memory unit, and an input/output (I/O) subsystem. A system bus interconnects these three components. The next three sections discuss these three components in detail. Section 1.5 provides an overview of the processor component. The processors we cover in this book include the Pentium, MIPS, PowerPC, Itanium, and SPARC. Section 1.6 presents some basic concepts about the memory system. Later chapters describe in detail cache and virtual memories. Section 1.7 gives a brief overview of how input/output devices such as the keyboard are interfaced to the system. A more

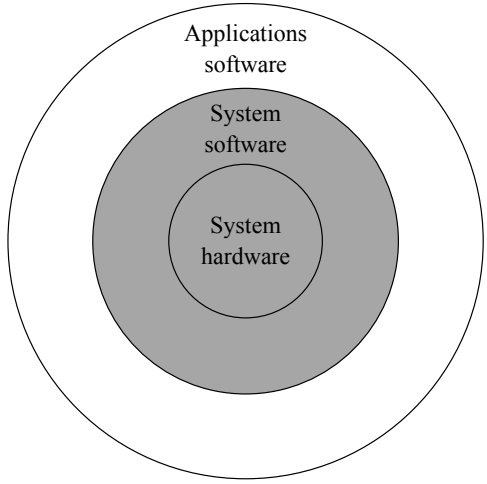


Figure 1.1 A user's view of a computer system.

detailed description on I/O interfacing can be found in the last two chapters. We conclude the chapter by providing a perspective on the history of computers.

1.1 Introduction

This book is about digital computer systems, which have been revolutionizing our society. Most of us use computers for a variety of tasks, from serious scientific computations to entertainment. You are reading this book because you are interested in learning more about these magnificent machines.

As with any complex project, several stages and players are involved in designing, implementing, and realizing a computer system. This book deals with inside details of a computer system, focusing on both hardware and software.

Computer hardware is the electronic circuitry that performs the actual work. Hardware includes things with which you are already familiar such as the processor, memory, keyboard, CD burner, and so on. Miniaturization of hardware is the most recent advance in the computer hardware area. This miniaturization gives us such compact things as PocketPCs and Flash memories.

Computer software can be divided into application software and system software. A user interacts with the system through an application program. For the user, the application is the computer! For example, if you are interested in browsing the Internet, you interact with the system through a Web browser such as the Netscape™ Communicator or Internet Explorer. For you, the system appears as though it is executing the application program (i.e., Web browser), as shown in Figure 1.1.

At the core is the basic hardware, over which a layer of system software hides the gory details about the hardware. Early ancestors of the Pentium and other processors were called *microprocessors* because they were less powerful than the processors used in the computers at that time.

The system software manages the hardware resources efficiently and also provides nice services to the application software layer. What is the system software? Operating systems such as Windows™, UNIX™, and Linux are the most familiar examples. System software also includes compilers, assemblers, and linkers that we discuss later in this book. You are probably more familiar with application software, which includes Web browsers, word processors, music players, and so on.

This book presents details on various aspects of computer system design and programming. We discuss organization and architecture of computer systems, how they are designed, and how they are programmed. In order to clarify the scope of this book, we need to explain these terms: computer architecture, computer organization, computer design, and computer programming.

Computer architecture refers to the aspects with which a programmer is concerned. The most obvious one is the design of an instruction set for the computer. For example, should the processor understand instructions to process multimedia data? The answer depends on the intended use of the system. Clearly, if the target applications involve multimedia, adding multimedia instructions will help improve the performance. Computer architecture, in a sense, describes the computer system at a logical level, from the programmer's viewpoint. It deals with the selection of the basic functional units such as the processor and memory, and how they should be interconnected into a computer system.

Computer organization is concerned with how the various hardware components operate and how they are interconnected to implement the architectural specifications. For example, if the architecture specifies a divide instruction, we will have a choice to implement this instruction either in hardware or in software. In a high-performance model, we may implement the division operation in hardware to provide improved performance at a higher price. In cheaper models, we may implement it in software. But cost need not be the only deciding criterion. For example, the Pentium processor implements the divide operation in hardware whereas the next generation Itanium processor implements division in software. If the next version of Itanium uses a hardware implementation of division, that does not change the architecture, only its organization.

Computer design is an activity that translates architectural specifications of a system into an implementation using a particular organization. As a result, computer design is sometimes referred to as *computer implementation.* A computer designer is concerned with the hardware design of the computer.

Computer programming involves expressing the problem at hand in a language that the computer can understand. As we show later, the native language that a computer can understand is called the machine language. But this is not a language with which we humans are comfortable. So we use a language that we can easily read and understand. These languages are called high-level languages, and include languages such as Java™ and C. We do not devote any space for these high-level languages as they are beyond the scope of this book. Instead, we discuss

in detail languages that are close to the architecture of a machine. This allows us to study the internal details of computer systems.

Computers are complex systems. How do we manage complexity of these systems? We can get clues from looking at how we manage complex systems in life. Think of how a large corporation is managed. We use a hierarchical structure to simplify the management: president at the top and employees at the bottom. Each level of management filters out unnecessary details on the lower levels and presents only an abstracted version to the higher-level management. This is what we refer to as abstraction. We study computer systems by using layers of abstraction.

Different people view computer systems differently depending on the type of their interaction. We use the concept of abstraction to look at only the details that are necessary from a particular viewpoint. For example, if you are a computer architect, you are interested in the internal details that do not interest a normal user of the system. One can look at computer systems from several different perspectives. We have already talked about the user's view. Our interest in this book is not at this level. Instead, we concentrate on the following views: (i) a programmer's view, (ii) an architect's view, and (iii) an implementer's view. The next three sections briefly discuss these perspectives.

1.1.1 Basic Terms and Notation

The alphabet of computers, more precisely digital computers, consists of 0 and 1. Each is called a *bit*, which stands for the binary digit. The term *byte* is used to represent a group of 8 bits. The term *word* is used to refer to a group of bytes that is processed simultaneously. The exact number of bytes that constitute a word depends on the system. For example, in the Pentium, a word refers to four bytes or 32 bits. On the other hand, eight bytes are grouped into a word in the Itanium processor. The reasons for this difference are explained later. We use the abbreviation "b" for bits, "B" for bytes, and "W" for words. Sometimes we also use *doubleword* and *quadword*. A doubleword has twice the number of bits as the word and the quadword has four times the number of bits in a word.

Bits in a word are usually ordered from right to left, as you would write digits in a decimal number. The rightmost bit is called the *least significant bit* (LSB), and the leftmost bit is called the *most significant bit* (MSB). However, some manufacturers use the opposite notation. For example, the PowerPC manuals use this notation. In this book, we consistently write bits of a word from right to left, with the LSB as the rightmost bit.

We use standard terms such as *kilo* (K), *mega* (M), *giga* (G), and so on to represent large integers. Unfortunately, we use two different versions of each, depending on the number system, decimal or binary. Table 1.1 summarizes the differences between the two systems. Typically, computer-related attributes use the binary version. For example, when we say 128 megabyte (MB) memory, we mean 128×2^{20} bytes. Usually, communication-related quantities and time units are expressed using the decimal system. For example, when we say that the data transfer rate is 100 megabits/second (Mb/s), we mean 100×10^6 Mb/s.

Throughout the text, we use various number systems: binary, octal, and hexadecimal. Now is a good time to refresh your memory by reviewing the material on number systems presented

Table 1.1 Terms to represent large integer values

Term	Decimal (base 10)	Binary (base 2)
K (kilo)	10^3	2^{10}
M (mega)	10^6	2^{20}
G (giga)	10^9	2^{30}
T (tera)	10^{12}	2^{40}
P (peta)	10^{15}	2^{50}

in Appendix A. If the number system used is not clear from the context, we use a trailing letter to specify the number system. We use "D" for decimal numbers, "B" for binary numbers, "Q" for octal numbers, and "H" for hexadecimal (or hex for short) numbers. For example, 10110101B is an 8-bit binary number whereas 10ABH is a hex number.

1.2 Programmer's View

A programmer's view of a computer system depends on the type and level of language she intends to use. From the programmer's viewpoint, there exists a hierarchy from low-level languages to high-level languages. As we move up in this hierarchy, the level of abstraction increases. At the lowest level, we have the *machine language* that is the native language of the machine. This is the language understood by the machine hardware. Since digital computers use 0 and 1 as their alphabet, machine language naturally uses 1s and 0s to encode the instructions. One level up, there is the assembly language as shown in Figure 1.2.

Assembly language does not use 1s and 0s; instead, it uses mnemonics to express the instructions. Assembly language is a close relative of the machine language. In the Pentium, there is a one-to-one correspondence between the instructions of the assembly language and its machine language. For example, to increment the variable `count`, we would write

```
inc   count
```

in Pentium assembly language. This assembly language instruction is translated into the machine language as

```
1111 1111 0000 0110 0000 1010 0000 0000B
```

which, as you can see, is very difficult to read. We improve the situation slightly by writing this instruction in hexadecimal notation as

```
FF060A00H
```

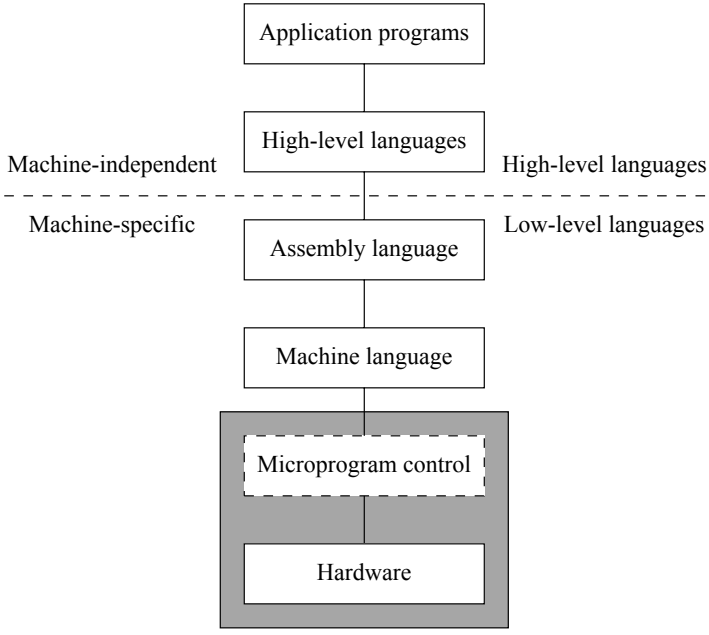


Figure 1.2 A programmer's view of a computer system.

Still, it is not a big help in understanding what this instruction does. Compared to the machine language, assembly language is far better in understanding programs. Since there is one-to-one correspondence between many assembly and machine language instructions, it is fairly straightforward to translate instructions from assembly language to machine language.

Assembler is the software that achieves this code translation. MASM (Microsoft Assembler), TASM (Borland Turbo Assembler), and NASM (Netwide Assembler) are some of the popular assemblers for the Pentium processors. As a result, only a masochist would consider programming in a machine language. However, life was not so easy for some of the early programmers. When microprocessors were first introduced, some programming was in fact done in machine language!

Although Pentium's assembly language is close to its machine language, other processors use the assembly language to implement a virtual instruction set that is more powerful and useful than the native machine language. In this case, an assembly language instruction could be translated into a sequence of machine language instructions. We show several examples of such assembly language instructions when we present details about the MIPS processor assembly language in Chapter 15.

Assembly language is one step above machine language; however, it is still considered a low-level language because each assembly language instruction performs a much lower-level task compared to an instruction in a high-level language. For example, the following C statement, which assigns the sum of four `count` variables to `result`

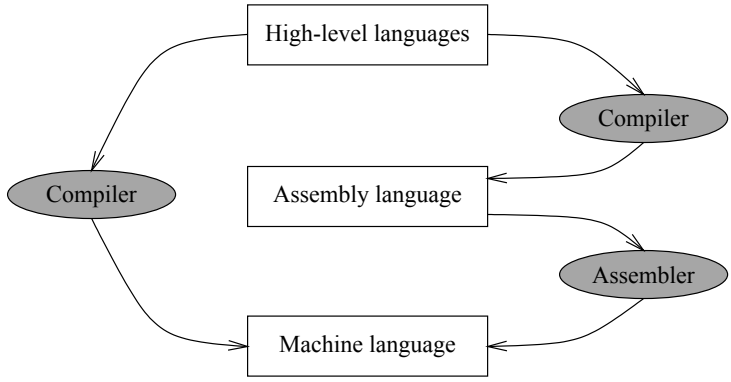


Figure 1.3 Translation of higher-level languages into machine language is done by assemblers and compilers. A compiler can translate a high-level language program directly into the machine language, or it can produce the equivalent assembly language.

```
result = count1 + count2 + count3 + count4;
```

is implemented in the Pentium assembly language as

```
mov    AX,count1
add    AX,count2
add    AX,count3
add    AX,count4
mov    result,AX
```

A *compiler* translates instructions from a high-level language to the machine language, either directly or via the assembly language (Figure 1.3).

Don't worry about the assembly language details here. The point to take away is that several assembly language instructions are required to implement a high-level language statement. As a result, assembly language code tends to be much larger than the equivalent high-level language code. Furthermore, assembly language instructions are native to a particular processor. For example, a program written in the Pentium assembly language cannot be executed on the PowerPC processor. Thus, assembly language programming is machine-specific, as shown in Figure 1.2. This machine dependence causes code portability problems.

The PC systems maintain backward compatibility in the sense that programs that executed on earlier Intel processors in the 1970s can still be run on current Pentium processors. This is possible because Intel processors maintain backward compatibility. However, Apple systems do not maintain such backward compatibility as the early Apple systems used Motorola processors, whereas the recent ones use PowerPC processors. Since these two processors have different instruction sets, programs that ran on one do not run on the other. Programming in assembly language also requires detailed knowledge about the system such as processor instruction set, memory organization, and so on.

One of the important abstractions that a programmer uses is the instruction set architecture (ISA). A machine language programmer views the machine at the level of abstraction provided by the ISA. The ISA defines the personality of a processor and indirectly influences the overall system design. The ISA specifies how a processor functions: what instructions it executes and what interpretation is given to these instructions. This, in a sense, defines a logical processor. If these specifications are precise, it gives freedom to various chip manufacturers to implement physical designs that look functionally the same at the ISA level. Thus, if we run the same program on these implementations, we get the same results. Different implementations, however, may differ in performance and price.

Implementations of the logical processor, shown shaded in Figure 1.2, can be done directly in the hardware or through another level of abstraction known as the *microprogram*. We use the dashed box to indicate that the microprogramming level is optional. We further discuss this topic in Section 1.5 and Chapter 6.

Two popular examples of ISA specifications are the SPARC and JVM. The rationale behind having a precise ISA-level specification for the SPARC is to let multiple vendors design chips that look the same at the ISA level. The JVM, on the other hand, takes a different approach. Its ISA-level specifications can be used to create a software layer so that the processor looks like a Java processor. Thus, in this case, we do not use a set of hardware chips to implement the specifications, but rather use a software layer to simulate the virtual processor. Note, however, that there is nothing stopping us from implementing these specifications in hardware (even though this is not usually the case). Thus, the underlying difference is whether the specifications are implemented in hardware or software.

Why create the ISA layer? The ISA-level abstraction provides details about the machine that are needed by the programmers to make machine language programs work on the machine. The idea is to have a common platform to execute programs. If a program is written in C, a compiler translates it into the equivalent machine language program that can run on the ISA-level logical processor. Similarly, if you write your program in FORTRAN, use a FORTRAN compiler to generate code that can execute on the ISA-level logical processor. For us, the abstraction at the ISA level is also important for one other reason. The ISA represents an interface between hardware and lowest-level software, that is, at the machine language level.

1.2.1 Advantages of High-Level Languages

High-level languages such as C and Java are preferred because they provide a convenient abstraction of the underlying system suitable for problem solving. The advantages of programming in a high-level language rather than in an assembly language include the following:

1. *Program development is faster in a high-level language.*
   Many high-level languages provide structures (sequential, selection, iterative) that facilitate program development. Programs written in a high-level language are relatively small and easier to code and debug.
2. *Programs written in a high-level language are easier to maintain.*
   Programming for a new application can take several weeks to several months, and the

lifecycle of such an application software can be several years. Therefore, it is critical that software development be done with a view toward software maintainability, which involves activities ranging from fixing bugs to generating the next version of the software. Programs written in a high-level language are easier to understand and, when good programming practices are followed, easier to maintain. Assembly language programs tend to be lengthy and take more time to code and debug. As a result, they are also difficult to maintain.

3. *Programs written in a high-level language are portable.*
   High-level language programs contain very few machine-specific details, and they can be used with little or no modification on different computer systems. In contrast, assembly language programs are written for a particular system and cannot be used on a different system.

1.2.2 Why Program in Assembly Language?

Despite these disadvantages, some programming is still done in assembly language. There are two main reasons for this: efficiency and accessibility to system hardware. *Efficiency* refers to how "good" a program is in achieving a given objective. Here we consider two objectives based on space (space-efficiency) and time (time-efficiency).

Space-efficiency refers to the memory requirements of a program (i.e., the size of the code). Program A is said to be more space-efficient if it takes less memory space than program B to perform the same task. Very often, programs written in an assembly language tend to generate more compact executable code than the corresponding high-level language version. You should not confuse the size of the source code with that of the executable code.

Time-efficiency refers to the time taken to execute a program. Clearly, a program that runs faster is said to be better from the time-efficiency point of view. Programs written in an assembly language tend to run faster than those written in a high-level language. However, sometimes a compiler-generated code executes faster than a handcrafted assembly language code!

As an aside, note that we can also define a third objective: how fast a program can be developed (i.e., the code written and debugged). This objective is related to *programmer productivity*, and assembly language loses the battle to high-level languages.

The superiority of assembly language in generating compact code is becoming increasingly less important for several reasons. First, the savings in space pertain only to the program code and not to its data space. Thus, depending on the application, the savings in space obtained by converting an application program from some high-level language to an assembly language may not be substantial. Second, the cost of memory (i.e., cost per bit) has been decreasing and memory capacity has been increasing. Thus, the size of a program is not a major hurdle anymore. Finally, compilers are becoming "smarter" in generating code that competes well with a handcrafted assembly code. However, there are systems such as mobile devices and embedded controllers in which space-efficiency is still important.

One of the main reasons for writing programs in assembly language is to generate code that is time-efficient. The superiority of assembly language programs in producing a code that runs

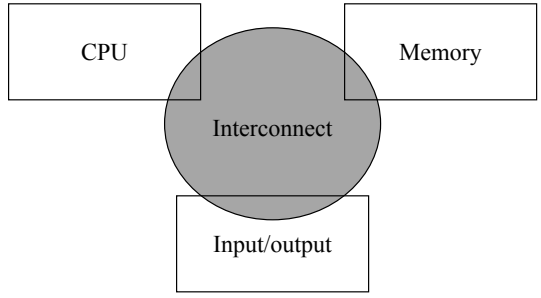


Figure 1.4 The three main components of a computer system are interconnected by a bus.

faster is a direct manifestation of *specificity*. That is, handcrafted assembly language programs tend to contain only the necessary code to perform the given task. Even here, a "smart" compiler can optimize the code that can compete well with its equivalent written in the assembly language.

Perhaps the main reason for still programming in an assembly language is to have direct control over the system hardware. High-level languages, on purpose, provide a restricted (abstract) view of the underlying hardware. Because of this, it is almost impossible to perform certain tasks that require access to the system hardware. For example, writing an interface program, called a *device driver*, to a new printer on the market almost certainly requires programming in an assembly language. Since assembly language does not impose any restrictions, you can have direct control over all of the system hardware. If you are developing system software (e.g., compiler, assembler, linker), you cannot avoid writing programs in assembly language.

In this book, we spend a lot time on the assembly language of Pentium and MIPS processors. Our reasons are different from what we just mentioned. We use assembly language as a tool to study the internal details of a computer.

1.3 Architect's View

A computer architect looks at the design aspect from a high level. She uses higher-level building blocks to optimize the overall system performance. A computer architect is much like an architect who designs buildings. For example, when designing a building, the building architect is not concerned with designing the elevator; as far as the architect is concerned, the elevator is a building block someone else designs. Similarly, a computer architect does not focus on low-level issues. To give you an example, let's look at a component called the arithmetic and logic unit (ALU) that is in all processors. This unit performs arithmetic operations such as addition and logical operations such as `and`. A computer architect, however, is not concerned with the internal details of the ALU.

From the architect's viewpoint, a computer system consists of three main components: a processor or central processing unit (CPU), a memory unit, and input/output (I/O) devices. An

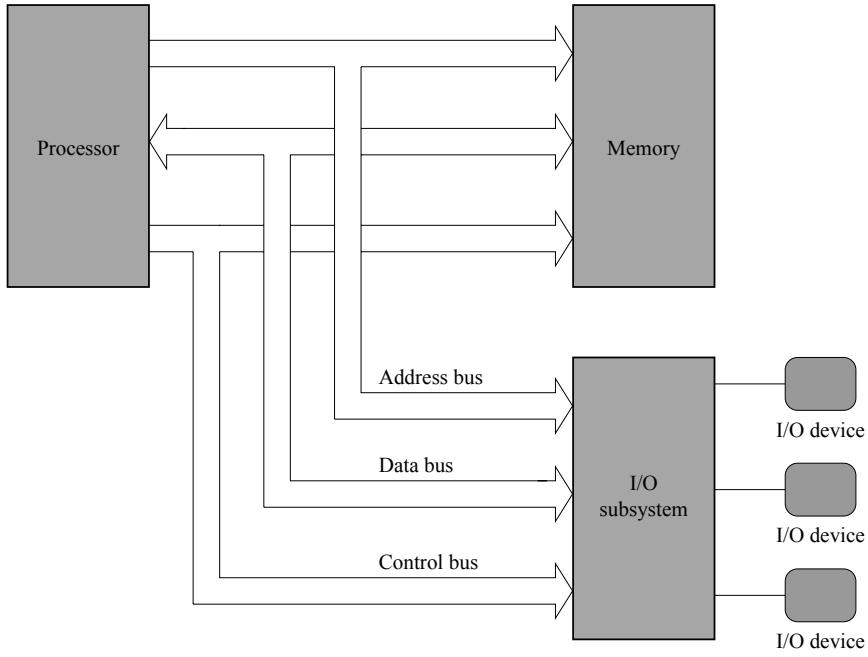


Figure 1.5 Simplified block diagram of a computer system.

interconnection network facilitates communication among these three components, as shown in Figure 1.4. An architect is concerned with the functional design of each of these components as well as integration of the whole system. Thus we can categorize architects into several classes, depending on their design goal. For example, a processor designer (or architect) is responsible for the processor component. She may deal with issues such as whether the design should follow the RISC philosophy or use the CISC design. We describe RISC and CISC designs in Section 1.5, and a later chapter gives more detailed information on them. On the other hand, a computer *system architect* designs the system using components such as the processor, memory unit, and I/O devices.

The interconnection network is called the *system bus*. The term "bus" is used to represent a group of electrical signals or the wires that carry these signals. As shown in Figure 1.5, the system bus consists of three major components: an address bus, a data bus, and a control bus.

The address bus width determines the amount of physical memory addressable by the processor. The data bus width indicates the size of the data transferred between the processor and memory or an I/O device. For example, the Pentium processor has 32 address lines and 64 data lines. Thus, the Pentium can address up to 2^{32}, or 4 GB of memory. Furthermore, each data transfer can move 64 bits of data. The Intel Itanium processor uses address and data buses that are twice the size of the Pentium buses (i.e., 64-bit address bus and 128-bit data bus). The

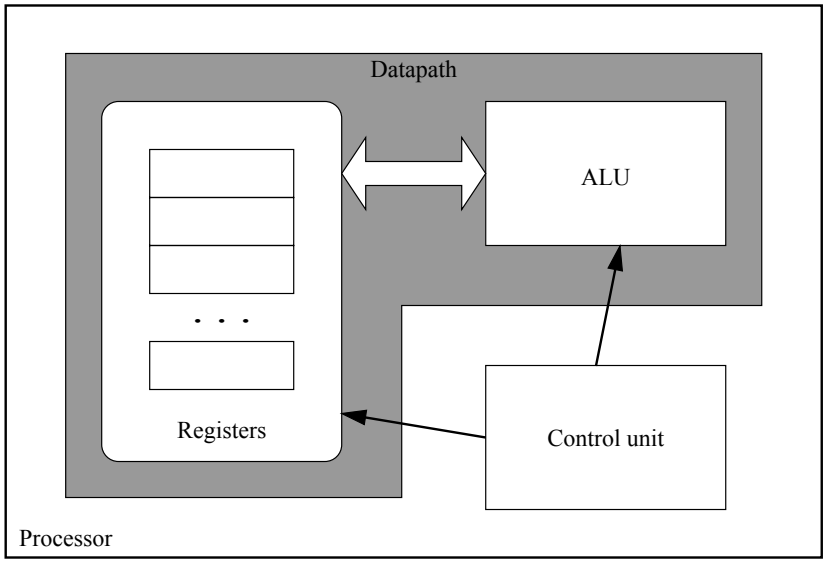


Figure 1.6 These three major components of a processor are interconnected by onchip buses. The datapath of a processor, shown shaded, consists of its register set and the arithmetic and logic unit.

Itanium, therefore, can address up to 2^{64} bytes of memory and each data transfer can move 128 bits.

The control bus consists of a set of control signals. Typical control signals include memory read, memory write, I/O read, I/O write, interrupt, interrupt acknowledge, bus request, and bus grant. These control signals indicate the type of action taking place on the system bus. For example, when the processor is writing data into the memory, the memory write signal is generated. Similarly, when the processor is reading from an I/O device, it generates the I/O read signal.

The system memory, also called the *main* or *primary memory*, is used to store both program instructions and data. Section 1.6 gives more details on the memory component.

As shown in Figure 1.5, the I/O subsystem interfaces the I/O devices to the system. I/O devices such as the keyboard, display screen, printer, and modem are used to provide user interfaces. I/O devices such as disks are used as secondary storage devices. We present details about the I/O subsystem in Chapters 19 and 20.

1.4 Implementer's View

Implementers are responsible for implementing the designs produced by computer architects. This group works at the digital logic level. At this level, logic gates and other hardware circuits are used to implement the various functional units.

From the implementer's viewpoint, the processor consists of the three components shown in Figure 1.6. The control unit fetches instructions from the main memory and decodes them to find the type of instruction. Thus, the control unit directly controls the operation of the processor. The datapath consists of a set of registers and one or more arithmetic and logic units (ALUs). Registers are used as a processor's scratchpad to store data and instructions temporarily. Because accessing data stored in the registers is faster than going to the memory, optimized code tends to put most-often accessed data in processor registers. Obviously, we would like to have as many registers as possible, the more the better. In general, all registers are of the same size. For example, registers in a 32-bit processor like the Pentium are all 32 bits wide. Similarly, 64-bit registers are used in 64-bit processors like the Itanium.

The number of processor registers varies widely. Some processors may have only about 10 registers, and others may have 100+ registers. For example, the Pentium has about 8 data registers and 8 other registers, whereas the Itanium has 128 registers just for integer data. There are an equal number of floating-point and application registers. We discuss the Pentium processor in Chapter 7 and the Itanium in Chapter 14.

Some of the registers contain special values. For example, all processors have a register called the *program counter* (PC). The PC register maintains a marker to the instruction that the processor is supposed to execute next. Some processors refer to the PC register as the instruction pointer (IP) register. There is also an *instruction register* (IR) that keeps the instruction currently being executed. Although some of these registers are not available, most processor registers can be used by the programmer.

The data from the register set are fed as input to the ALU through ALU input buses, as shown in Figure 1.7. Here, we have two buses (A and B) to carry the input operands required by the ALU. The ALU output is fed back to the register set using the C bus.

The memory interface consists of the four shaded registers. We have already mentioned the PC and IR registers. The memory address register (MAR) holds the address of the memory and the memory data register (MDR) holds the data.

The ALU hardware performs simple operations such as addition and logical `and` on the two input operands. The ALU control input determines the operation to be done on the input operands. The ALU output can be placed back in one of the registers or can be written into the main memory. If the result is to be written into the memory, the ALU output should be placed in MDR. This value in MDR is written at the memory address in MAR.

In RISC processors, the results are always written into a register. These types of processors (e.g., MIPS and Itanium) have special instructions to move data between registers and memory. CISC processors such as the Pentium do not enforce such a restriction. As we show in later chapters, CISC processors allow the output to go either to one of the registers or to a memory location.

Implementers are concerned with the design of these components. Figure 1.8 shows a sample implementation of a simple 1-bit ALU design using digital logic gates. This ALU can perform logical AND and OR operations on the two inputs A and B; it can also perform two arithmetic operations: addition ($A + B + C_{in}$) and subtraction ($A - B - C_{in}$). Clearly, all of this does not make sense to you right now. The idea in presenting this material is to convey the

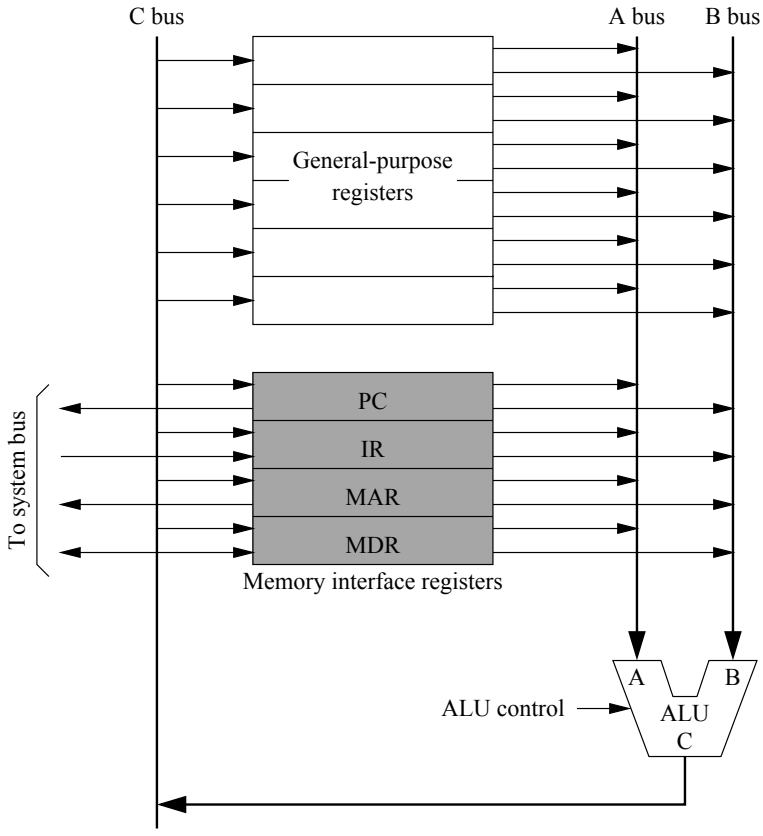


Figure 1.7 This datapath uses three internal buses to connect registers to the ALU.

high-level view, rather than the low-level details. We cover digital design details in Part II of this book.

Implementers can choose to implement the architecture in several different ways. The implementation, for example, can be done by using custom-designed chips, general-purpose programmable logic arrays (PLAs), or basic logic gates. An implementer optimizes implementation to achieve a specific objective such as minimization of cost or minimization of power consumption (e.g., for handheld devices).

1.5 The Processor

The processor acts as the controller of all actions or services provided by the system. Processor actions are synchronized to its clock input. A clock signal, which is a square wave, consists of clock cycles. The time to complete a clock cycle is called the clock period. Normally, we use the clock frequency, which is the inverse of the clock period, to specify the clock. The clock frequency is measured in Hertz, which represents one cycle/second. Hertz is abbreviated as Hz.

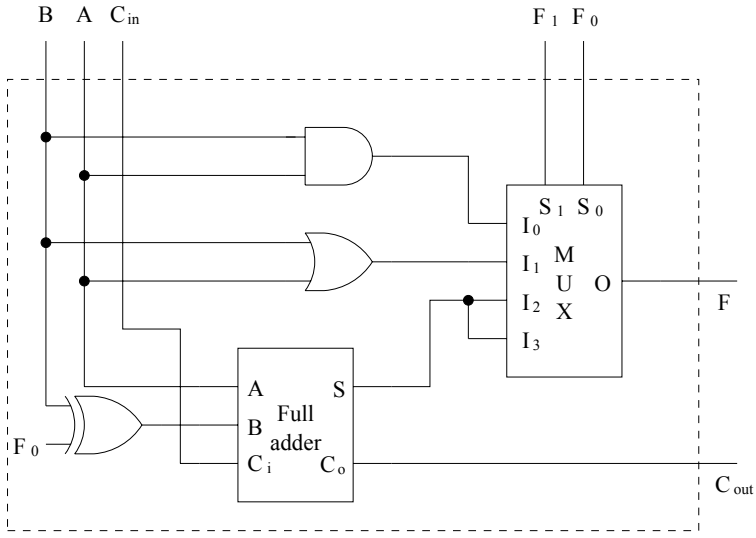


Figure 1.8 An example 1-bit ALU design. It can perform one of four functions, selected by F_1F_0 inputs.

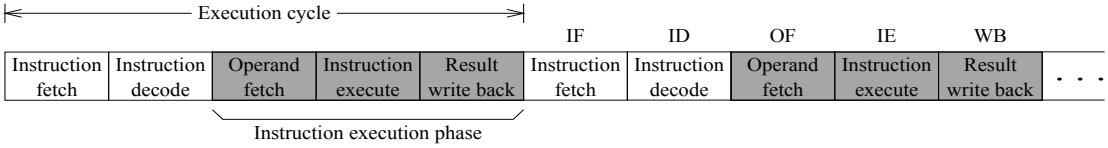


Figure 1.9 An execution cycle consists of fetch, decode, and execution phases. The execution phase consists of three steps.

Usually, we use mega Hertz (MHz) and giga Hertz (GHz) as in 1.8 GHz Pentium. We give more details about the clock signal in Section 4.2 on page 111.

The processor can be thought of as executing the following cycle forever (see Figure 1.9):

1. Fetch an instruction from the memory,
2. Decode the instruction (i.e., determine the instruction type),
3. Execute the instruction (i.e., perform the action specified by the instruction).

Execution of an instruction involves fetching any required operands, performing the specified operation, and writing the results back. This process is often referred to as the *fetch-execute* cycle, or simply the *execution* cycle.

This raises several questions. Who provides the instructions to the processor? Who places these instructions in the main memory? How does the processor know where in memory these instructions are located?

When we write programs—whether in a high-level language or in an assembly language—we provide a sequence of instructions to perform a particular task (i.e., solve a problem). A compiler or assembler will eventually translate these instructions to an equivalent sequence of machine language instructions that the processor understands.

The operating system, which provides instructions to the processor whenever a user program is not executing, loads the user program into the main memory. The operating system then indicates the location of the user program to the processor and instructs it to execute the program.

The features we have just described are collectively referred to as the *von Neumann architecture*, which uses what is known as the *stored program model*. The key features of this architecture are as follows:

- There is no distinction between instructions and data. This requirement has several main implications:
  1. Instructions are represented as numbers, just like the data themselves. This uniform treatment of instructions and data simplifies the design of memory and software.
  2. Instructions and data are not stored in separate memories; a single memory is used for both. Thus, a single path from the memory can carry both data and instructions.
  3. The memory is addressed by location, regardless of the type of data at that location.
- By default, instructions are executed in the sequential manner in which they are present in the stored program. This behavior can be changed, as you know, by explicitly executing instructions such as procedure calls.

In contrast to the single memory concept used in the von Neumann architecture, the *Harvard architecture* uses separate memories for instructions and data. The term now refers to machines that have a single main memory but use separate caches for instructions and data (see page 26).

1.5.1 Pipelining

What we have shown in Figure 1.9 is a simple execution cycle. In particular, notice that the control unit would have to wait until the instruction is fetched from memory. Furthermore, the ALU would have to wait until the required operands are fetched from memory. As we show later in this chapter, processor speeds are increasing at a much faster rate than the improvements in memory speeds. Thus, we would be wasting the control unit and ALU resources by keeping them idle while the system fetches instructions and data. How can we avoid this situation? Let's suppose that we can prefetch the instruction. That is, we read the instruction before the control unit needs it. These prefetched instructions are typically placed in a set of registers called the *prefetch buffers*. Then, the control unit doesn't have to wait.

How do we do this prefetch? Given that the program execution is sequential, we can prefetch the next instruction in sequence while the control unit is busy decoding the current instruction. Pipelining generalizes this concept of overlapped execution. Similarly, prefetching the required operands avoids the idle time experienced by the ALU.

	Time (cycles) ⟶									
Stage	1	2	3	4	5	6	7	8	9	10
S1: IF	I1	I2	I3	I4	I5	I6	·	· ·		
S2: ID		I1	I2	I3	I4	I5	I6	·	· ·	
S3: OF			I1	I2	I3	I4	I5	I6	·	· ·
S4: IE				I1	I2	I3	I4	I5	I6	· ·
S5: WB					I1	I2	I3	I4	I5	I6

Figure 1.10 A pipelined execution of the basic execution cycle shown in Figure 1.9.

Figure 1.10 shows how pipelining helps us improve the efficiency. As we have seen in Figure 1.9, the instruction execution can be divided into five parts. In pipelining terminology, each part is called a stage. For simplicity, let's assume that execution of each stage takes the same time (say, one cycle). As shown in Figure 1.10, each stage spends one cycle in executing its part of the execution cycle and passes the instruction on to the next stage. Let's trace the execution of this pipeline during the first few cycles. During the first cycle, the first stage S1 fetches the instruction. All other stages are idle. During Cycle 2, S1 passes the first instruction I1 to stage S2 for decoding and S1 initiates the next instruction fetch. Thus, during Cycle 2, two of the five stages are busy: S2 decodes I1 while S1 is busy with fetching I2. During Cycle 3, stage S2 passes instruction I1 to stage S3 to fetch any required operands. At the same time, S2 receives I2 from S1 for decoding while S1 fetches the third instruction. This process is repeated in each stage. As you can see, after four cycles, all five stages are busy. This state is called the pipeline full condition. From this point on, all five stages are busy.

Figure 1.11 shows an alternative way of looking at pipelined execution. This figure clearly shows that the execution of instruction I1 is completed in Cycle 5. However, after Cycle 5, notice that one instruction is completed in each cycle. Thus, executing six instructions takes only 10 cycles. Without pipelining, it would have taken 30 cycles.

Notice from this description that pipelining does not speed up execution of individual instructions; each instruction still takes five cycles to execute. However, pipelining increases the number of instructions executed per unit time; that is, instruction throughput increases.

1.5.2 RISC and CISC Designs

We have briefly mentioned the two basic types of processor design philosophies: reduced instruction set computers (RISC) and complex instruction set computers (CISC). First, let us talk about the trend line. The current trend in processor design is to use RISC philosophy. In the 1970s and early 1980s, processors predominantly followed the CISC designs. To understand this shift from CISC to RISC, we need to look at the motivation for going the CISC way initially. But first we have to explain what these two types of design philosophies are.

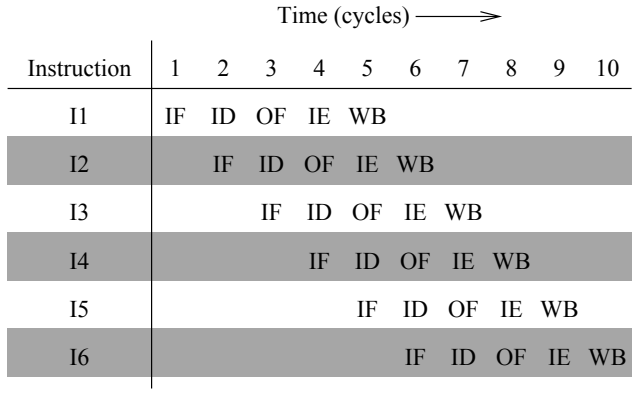
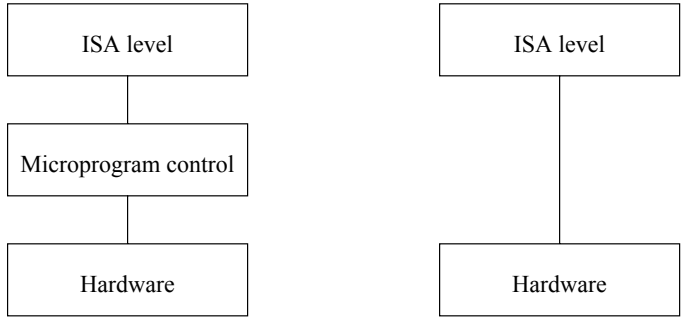


Figure 1.11 An alternative way of looking at the pipelined execution shown in Figure 1.10.

As the name suggests, CISC systems use complex instructions. What is a complex instruction? For example, adding two integers is considered a simple instruction. But, an instruction that copies an element from one array to another and automatically updates both array subscripts is considered a complex instruction. RISC systems use only simple instructions such as the addition. Furthermore, RISC systems assume that the required operands are in the processor's registers, not in main memory. As mentioned before, a CISC processor does not impose such restrictions. So what? It turns out that characteristics like simple instructions and restrictions like register-based operands not only simplify the processor design but also result in a processor that provides improved application performance. We give a detailed list of RISC design characteristics and its advantages in Chapter 14.

How come the early designers did not think about the RISC way of designing processors? Several factors contributed to the popularity of CISC in the 1970s. In those days, memory was very expensive and small in capacity. For example, even in the mid-1970s, the price of a small 16 KB memory was about $500. You can imagine the cost of memory in the 1950s and 1960s. So there was a need to minimize the amount of memory required to store a program. An implication of this requirement is that each processor instruction must do more, leading to complex instructions. This caused another problem. How can a processor be designed that can execute such complex instructions using the technology of the day? Complex instructions meant complex hardware, which was also expensive. This was a problem processor designers grappled with until Wilkes proposed microprogrammed control in 1951 [39].

A microprogram is a small run-time interpreter that takes the complex instruction and generates a sequence of simple instructions that can be executed by hardware. Thus the hardware need not be complex. Once it became possible to design such complex processors by using microprogrammed control, designers went crazy and tried to close the semantic gap between the instructions of the processor and high-level languages. This semantic gap refers to the fact that each instruction in a high-level language specifies a lot more work than an instruction in



(a) CISC implementation (b) RISC implementation

Figure 1.12 The ISA-level architecture can be implemented either directly in hardware or through a microprogrammed control.

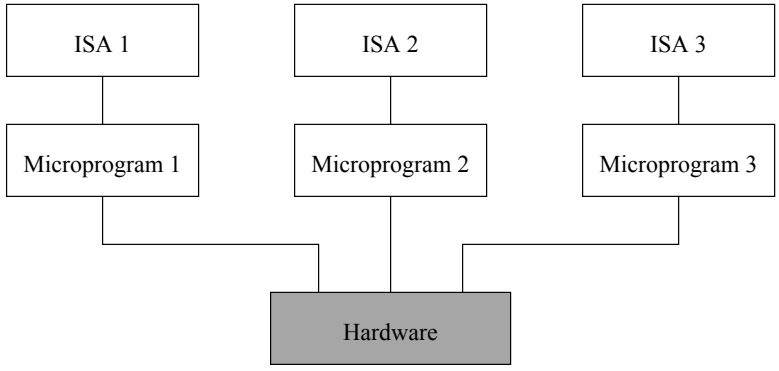


Figure 1.13 Variations on the ISA-level architecture can be implemented by changing the microprogram.

the machine language. Think of a `while` loop statement in a high-level language such as C, for example. If we have a processor instruction with the `while` loop semantics, we could just use one machine language instruction. Thus, most CISC designs use microprogrammed control, as shown in Figure 1.12.

RISC designs, on the other hand, eliminate the microprogram layer and use the hardware to directly execute instructions. Here is another reason why RISC processors can potentially give improved performance. One advantage of using microprogrammed control is that we can implement variations on the basic ISA architecture by simply modifying the microprogram; there is no need to change the underlying hardware, as shown in Figure 1.13. Thus, it is possible to come up with cheaper versions as well as high-performance processors for the same family.

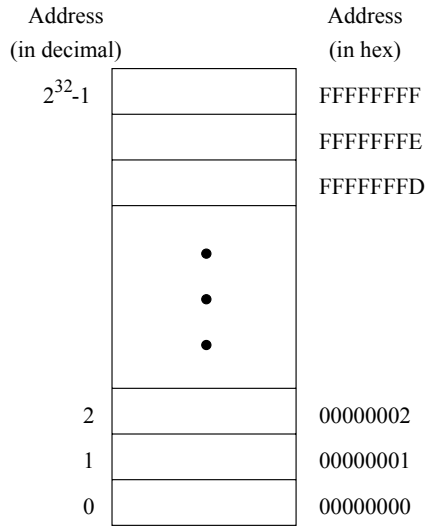


Figure 1.14 Logical view of the system memory.

1.6 Memory

The memory of a computer system consists of tiny electronic switches, with each switch in one of two states: *open* or *closed*. It is, however, more convenient to think of these states as 0 and 1, rather than open and closed. Thus, each switch can represent a bit. The memory unit consists of millions of such bits. In order to make memory more manageable, eight bits are grouped into a byte. Memory can then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory is identified by its sequence number starting with 0, as shown in Figure 1.14. This is referred to as the *memory address* of the byte. Such memory is called *byte addressable* memory because each byte has a unique address.

The Pentium can address up to 4 GB (2^{32} bytes) of main memory (see Figure 1.14). This magic number comes from the fact that the address bus of the Pentium has 32 address lines. This number is referred to as the *memory address space*. The memory address space of a system is determined by the address bus width of the processor used in the system. The actual memory in a system, however, is always less than or equal to the memory address space. The amount of memory in a system is determined by how much of this memory address space is *populated* with memory chips.

Although the 4-GB memory address space of the Pentium is large for desktop systems, it is not adequate for server systems. To support this market, 64-bit processors support even larger memory address space. Typically, these processors use 64-bit addresses. For example, the Intel 64-bit Itanium processor uses 64-bit addresses with an address space of 2^{64} bytes.

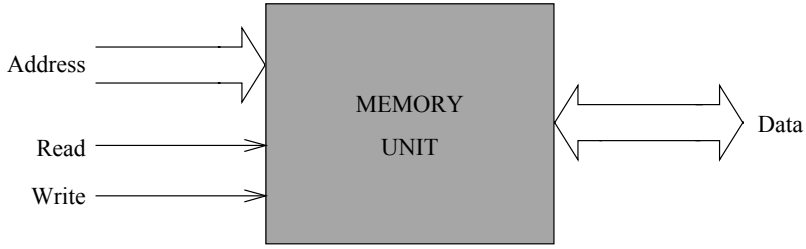


Figure 1.15 Block diagram of the system memory.

1.6.1 Basic Memory Operations

The memory unit supports two basic operations: *read* and *write*. The *read operation* reads previously stored data and the *write operation* stores a new value in memory. Both of these operations require a memory address. In addition, the write operation requires specification of the data to be written. The block diagram of the memory unit is shown in Figure 1.15. The address and data of the memory unit are connected to the address and data buses of the system bus, respectively. The read and write signals come from the control bus.

Two metrics are used to characterize memory. *Access time* refers to the amount of time required by the memory to retrieve the data at the addressed location. The other metric is the *memory cycle time*, which refers to the minimum time between successive memory operations.

The read operation is nondestructive in the sense that one can read a location of the memory as many times as one wishes without destroying the contents of that location. The write operation, however, is destructive, as writing a value into a location destroys the old contents of that memory location. It seems only natural to think that the read operation is nondestructive. You will be surprised to know that the DRAM you are familiar with has the destructive read property. Thus, in DRAMs, a read has to be followed by a write to restore the contents.

Steps in a Typical Read Cycle:

1. Place the address of the location to be read on the address bus,
2. Activate the memory read control signal on the control bus,
3. Wait for the memory to retrieve the data from the addressed memory location and place them on the data bus,
4. Read the data from the data bus,
5. Drop the memory read control signal to terminate the read cycle.

A simple Pentium read cycle takes three clock cycles. During the first clock cycle, Steps 1 and 2 are performed. The Pentium waits until the end of the second clock and reads the data and drops the read control signal. If the memory is slower (and therefore cannot supply data within the specified time), the memory unit indicates its inability to the processor and the processor waits longer for the memory to supply data by inserting *wait cycles*. Note that each wait cycle

introduces a waiting period equal to one system clock period and thus slows down the system operation.

Steps in a Typical Write Cycle:

1. Place the address of the location to be written on the address bus,
2. Place the data to be written on the data bus,
3. Activate the memory write control signal on the control bus,
4. Wait for the memory to store the data at the addressed location,
5. Drop the memory write signal to terminate the write cycle.

As with the read cycle, the Pentium requires three clock cycles to perform a simple write operation. During the first clock cycle, Steps 1 and 3 are done. The idea behind initiating Step 3 ahead of Step 2 is to give advance notice to the memory as to the type of operation. Step 2 is performed during the second clock cycle. The Pentium gives memory time until the end of the second clock and drops the memory write signal. If the memory cannot write data at the maximum processor rate, wait cycles can be introduced to extend the write cycle to give more time to the memory unit. We discuss hardware memory design issues in Chapter 16.

1.6.2 Byte Ordering

Storing data often requires more than a byte. For example, we need four bytes of memory to store an integer variable that can take a value between 0 and $2^{32} - 1$. Let us assume that the value to be stored is the one in Figure 1.16*a*.

Suppose that we want to store these 4-byte data in memory at locations 100 through 103. How do we store them? Figure 1.16 shows two possibilities: least significant byte (Figure 1.16*b*) or most significant byte (Figure 1.16*c*) is stored at location 100. These two byte ordering schemes are referred to as the *little endian* and *big endian*. In either case, we always refer to such multibyte data by specifying the lowest memory address (100 in this example).

Is one byte ordering scheme better than the other? Not really! It is largely a matter of choice for the designers. For example, Pentium processors use the little-endian byte ordering. However, most processors leave it up to the system designer to configure the processor. For example, the MIPS and PowerPC processors use the big-endian byte ordering by default, but these processors can be configured to use the little-endian scheme.

The particular byte ordering scheme used does not pose any problems as long as you are working with machines that use the same byte ordering scheme. However, difficulties arise when you want to transfer data between two machines that use different schemes. In this case, conversion from one scheme to the other is required. For example, the Pentium provides two instructions to facilitate such conversion: one to perform 16-bit data conversions and the other for 32-bit data.

1.6.3 Two Important Memory Design Issues

When designing system memory, some major issues need to be addressed:

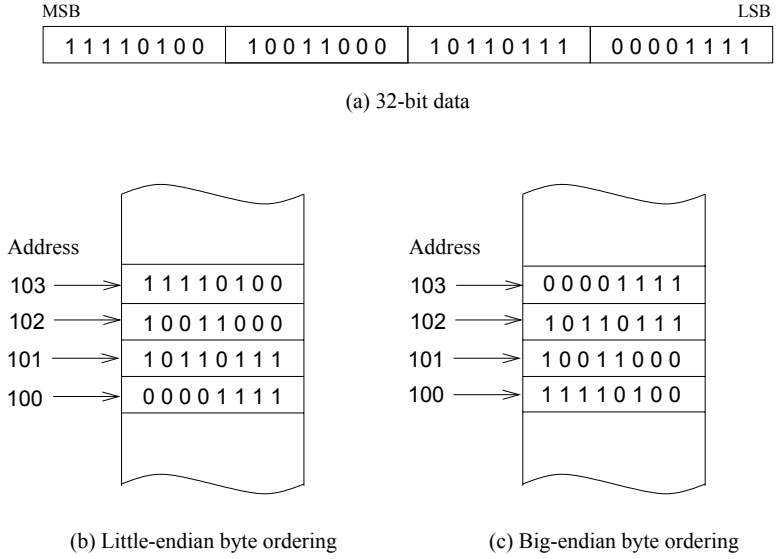


Figure 1.16 Two byte ordering schemes commonly used by computer systems.

1. *Slower Memories:* Advances in technology and processor architecture led to extremely fast processors. Technological advances pushed the basic clock rate into giga Hertz range. Simultaneously, architectural advances such as multiple pipelines and superscalar designs reduced the number of clock cycles required to execute an instruction. Thus, there is a lot of pressure on the memory unit to supply instructions and data at faster rates. If the memory can't supply the instructions and data at a rate required by the processor, what is the use of designing faster processors? To improve overall system performance, ideally, we would like to have lots of fast memory. Of course, we don't want to pay for it. Designers have proposed cache memories to satisfy these requirements.

2. *Physical Memory Size Limitation:* Even though processors have a large memory address space, only a fraction of this address space actually contains memory. For example, even though the Pentium has 4 GB of address space, most PCs now have between 128 MB and 256 MB of memory. Furthermore, this memory is shared between the system and application software. Thus, the amount of memory available to run a program is considerably smaller. In addition, if you run more programs simultaneously, each application gets an even smaller amount of memory. You might have experienced the result of this: terrible performance.

   Apart from the performance issue, this scenario also causes another more important problem: What if your application does not fit into its allotted memory space? How do you run such an application program? This is the motivation for proposing *virtual memory*, which we briefly describe later.

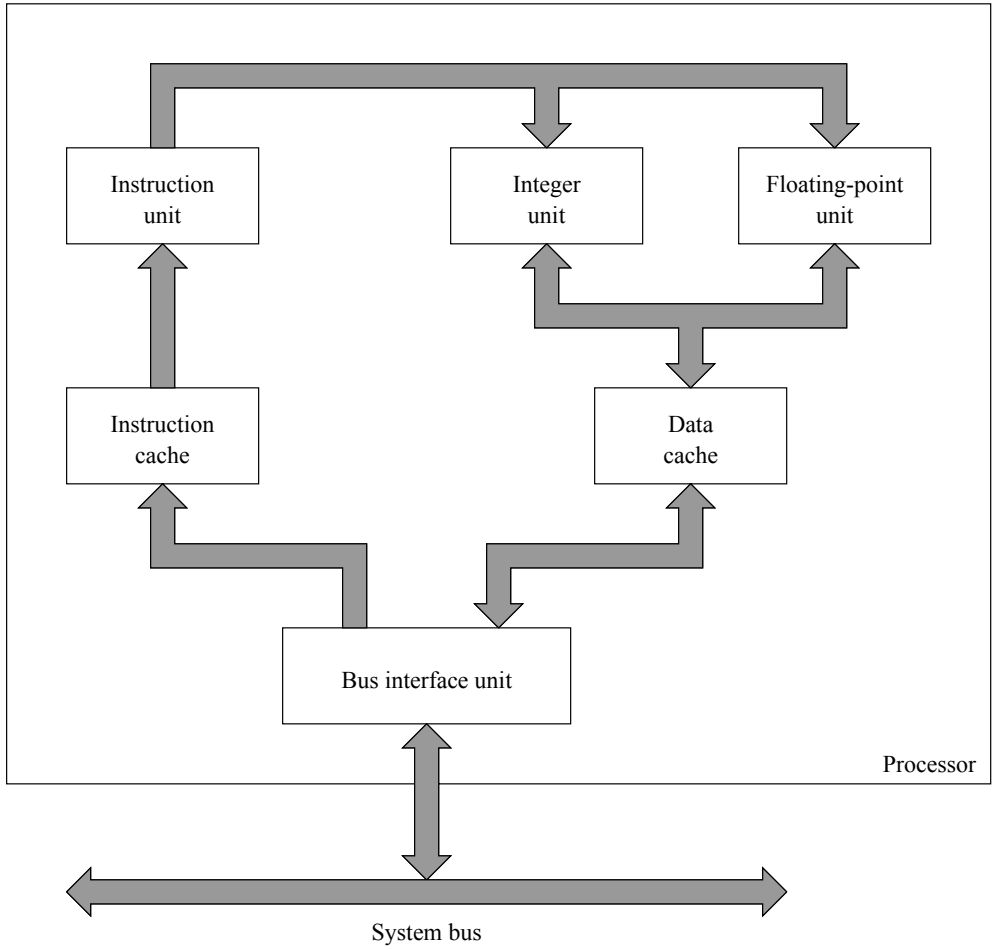


Figure 1.17 Most current processors use separate caches for instructions and data with separate instruction and data buses.

Cache memory: Cache memory successfully bridges the speed gap between the processor and memory. The cache is a small amount of fast memory that sits between the processor and the main memory. Cache memory is implemented by using faster memory technology compared to the technology used for the main memory. Abstractly, we can view the processor's register set as the fastest memory available to store data. The next best is the cache memory.

Cache memory is much smaller than the main memory. In PCs, for example, main memory is typically in the 128 to 256 MB range, whereas the cache is in the range of 16 to 512 KB.

The principle behind the cache memories is to prefetch the instructions and data from the main memory before the processor needs them. If we are successful in predicting what the

processor needs in the near future, we can preload the cache and supply the instructions and data from the faster cache. Early processors were designed with a common cache for both instructions and data. Most processors now use two separate caches: one for instructions and the other for data (Figure 1.17). This design uses separate buses for instructions and data. Such architectures are commonly referred to as the *Harvard architecture*.

It turns out that predicting processor future accesses is not such a difficult thing. To successfully predict the processor access needs, we need to understand the access referencing behavior of programs. Several researchers have studied program referencing behavior and shown that programs exhibit a phenomenon known as *locality* in their referencing behavior. This behavior can be exploited to successfully predict future accesses. In practice, we can predict with more than 90% accuracy! Cache memory is discussed in detail in Chapter 17.

Virtual memory: Virtual memory was developed to eliminate the physical memory size restriction mentioned before. There are some similarities between the cache memory and virtual memory. Just as with the cache memory, we would like to use the relatively small main memory and create the illusion (to the programmer) of a much larger memory, as shown in Figure 1.18. The programmer is concerned only with the virtual address space. Programs use virtual addresses and when these programs are run, their virtual addresses are mapped to physical addresses at run time.

The illusion of larger address space is realized by using much slower disk storage. Virtual memory can be implemented by devising an appropriate mapping function between the virtual and physical address spaces. As a result of this similarity between cache and virtual memories, both memory system designs are based on the same underlying principles. The success of the virtual memory in providing larger virtual address space also depends on the locality we mentioned before.

Before the virtual memory technique was proposed, a technique known as overlaying was used to run programs that were larger than the physical memory. In this technique, the programmer divides the program into several chunks, each of which could fit in the memory. These chunks are known as the overlays. The whole program (i.e., all overlays) resides on the disk. The programmer is responsible for explicitly managing the overlays. Typically, when an overlay in the memory is finished, it will bring in the next overlay that is required for program execution. Needless to say, this is not something a programmer relishes. Virtual memory takes this onerous task away from the programmer by automating the management of overlays without involving the programmer. Typically, virtual memory implementations provide much more functionality than the management of overlays. We discuss virtual memory in Chapter 18.

1.7 Input/Output

Input/output devices provide the means by which a computer system can interact with the outside world. Computer systems typically have several I/O devices, from slow devices such as the keyboard to high-speed disk drives and communication networks. Irrespective of the type of device, the underlying principles of interfacing an I/O device are the same. This interface typically consists of an I/O controller.

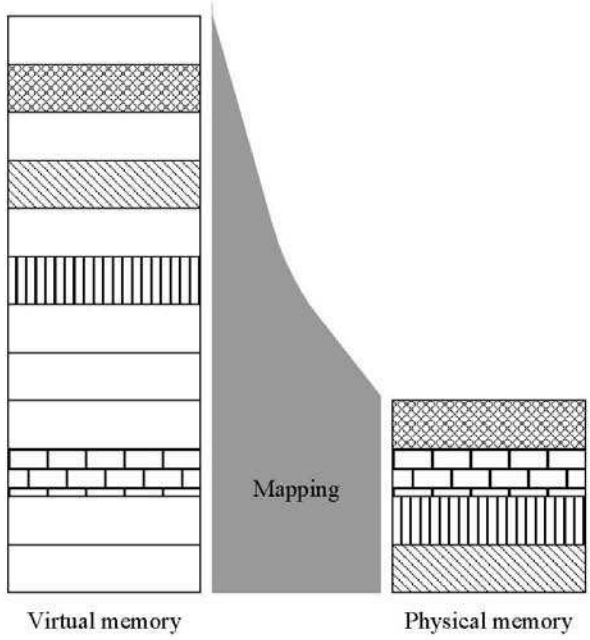


Figure 1.18 Virtual memory creates the illusion of a much larger memory to the programmer than what is physically available in the system.

Some I/O devices such as printers, keyboards, and modems are used to communicate with outside entities (could be a user or another computer). We also use I/O devices such as disk drives and CD writers to store data. Regardless of the intended purpose, the system communicates with the I/O devices through the system bus, as shown in Figure 1.5 on page 13. An *I/O controller* acts as an interface between the system bus and the I/O device, as shown in Figure 1.19.

I/O controllers have three types of internal registers—a data register, a command register, and a status register—as shown in Figure 1.19. When the processor wants to interact with an I/O device, it communicates only with the associated I/O controller. A processor can access the internal registers of an I/O controller through what are known as the *I/O ports*. An I/O port is a fancy name for the address of a register in an I/O controller. In that sense, the I/O port is like the memory address we use to specify a memory location. Since I/O ports appear as memory addresses, why not assign some part of the memory address space, where there is no physical memory, for I/O ports? We certainly can. If we map I/O ports like this, it is called memory-mapped I/O. Most processors, including the PowerPC and MIPS, support only memory-mapped I/O.

Sometimes we don't want to take part of memory address space for I/O ports, particularly if the memory address space we have is small for our applications. This was the case, for

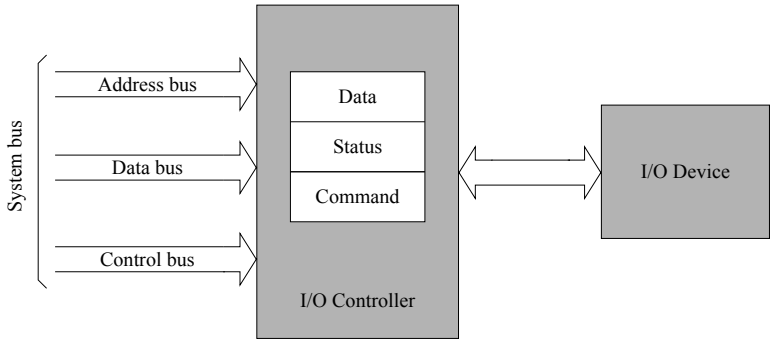


Figure 1.19 Block diagram of a generic I/O device interface.

example, when Intel introduced the 8086 processor. This processor had 20 address lines, which means it could address only one megabyte (1 MB) of memory. That is not a large address space considering that current PCs have 128 MB to 512 MB of physical memory.

In such cases, it is better to create a separate I/O address space. This mapping scheme is called isolated I/O. Because the Pentium is backward compatible to the earlier 8086 processor, the Pentium still supports isolated I/O. In memory-mapped I/O, writing to an I/O port is like writing to a memory location. Thus, memory-mapped I/O does not require any special design consideration. Thus, all processors, including the Pentium, inherently support memory-mapped I/O. In isolated I/O, special I/O instructions are needed to access the I/O address space. Details on these two mapping schemes and their impact on processor design are discussed in Chapter 19.

I/O ports provide the basic access to I/O devices via the associated I/O controller. We still will have to devise ways to transfer data between the system and I/O devices using the I/O ports. A simple way of transferring data is to ask the processor to do the transfer. In this scheme of things, the processor is responsible for transferring data word by word. Typically, it executes a loop until the data transfer is complete. This technique is called *programmed I/O*. One disadvantage of this scheme is that it wastes processor time. That is like asking a highly paid CEO of a company to take care of the company's reception area.

Is there another way of performing the I/O activity without wasting the processor's time? Carrying on with our analogy, we would naturally hire a receptionist and ask him to handle these low-level chores. Computer systems also employ a similar technique. It is called *direct memory access* (DMA). In DMA, the processor gives the command such as "transfer 10 KB to I/O port 125" and the DMA performs the transfer without bothering the processor. Once the operation is complete, the processor is notified. This notification is done by using an interrupt mechanism. We use DMA to transfer bulk data, not for single word transfers. A special DMA controller is used to direct the DMA transfer operations. We discuss these topics in detail in Chapters 19 and 20.

1.8 Interconnection: The Glue

You realize from our discussion so far that computer systems have several components interconnected by buses. We can talk about buses at various levels. The processor uses several internal buses to interconnect registers and the ALU. We also need interconnection to carry the control unit's signals. For example, in Figure 1.7, we used three buses (A, B, and C buses) to provide the interconnection between the register set and the ALU. Similarly, in Figure 1.17, data and instruction buses are used to connect various execution units to their caches. These are just two examples; a processor may have several such buses. These buses are called the onchip buses and are not accessible from outside the chip. We discuss the datapath in detail in Chapter 6.

The second type of buses is internal to a system. For example, the system bus shown in Figure 1.5 is typically inside the CPU box. Several bus standards have been proposed to facilitate interchangeability of system components. These include the ISA, PCI, AGP, and PCMCIA. A computer system typically has several of these buses (for a quick peek, look at Figure 5.14 on page 167). Chapter 5 describes various internal buses.

External buses, on the other hand, are used to interface the devices outside a typical computer system. Thus, by our classification, serial and parallel interfaces, universal serial bus (USB), and IEEE 1394 (also known as the FireWire) belong to the external category. These buses are typically used to connect I/O devices. External buses are discussed in Chapter 19.

Since the bus is a shared resource, we need to define how the devices connected to the bus will use it. For this purpose, we define *bus transaction* as a sequence of actions to complete a well-defined activity. Every bus transaction involves a master and a slave. Some examples of such activities are memory read, memory write, I/O read, and burst read. During a bus transaction, a *master* device will initiate the transaction and a *slave* device will respond to the master's request. In a memory read/write transaction, the processor is the master and the memory is the slave. Some units such as memory can only act as slaves. Other devices can act both as master and slave (but not at the same time). The DMA controller is an example. It acts as a slave when receiving a command from the processor. However, during the DMA transfer cycles, it acts as the master.

A bus transaction may perform one or more *bus operations*. For example, the Pentium burst read transfers four words. Thus this bus transaction consists of four memory read operations. Each operation may take several bus cycles. A bus cycle is the clock cycle of the bus clock.

Bus systems with more than one potential bus master need a bus arbitration mechanism to allocate the bus to a bus master. The processor is the bus master most of the time, but the DMA controller acts as the bus master during DMA transfers. In principle, bus arbitration can be done either statically or dynamically. In the static scheme, bus allocation among the potential masters is done in a predetermined way. For example, we might use a round-robin allocation that rotates the bus among the potential masters. The main advantage of a static mechanism is that it is easy to implement. However, since bus allocation follows a predetermined pattern rather than the actual need, a master may be given the bus even if it does not need it. This kind of allocation leads to inefficient use of the bus. Consequently, most bus arbitration implementations use a dynamic scheme, which uses a demand-driven allocation scheme. We present details on bus arbitration in Chapter 5.

1.9 Historical Perspective

This section traces the history of computers from their mechanical era. Our treatment is very brief. There are several sources that cover this material, including [8, 14, 9, 37, 28, 33].

1.9.1 The Early Generations

Before the vacuum tube generation, computing machines were either purely mechanical or electromechanical. Mechanical devices, called calculating machines, were built using gears and powered by a hand-operated crank. Perhaps the most well-known mechanical system, called the *difference engine*, was built by Charles Babbage (1792–1871). His *analytical engine*, a successor of the difference engine, had many of the components we have in our current computers. It had an ALU (it was called the *mill*), a memory (called the *store*), and input and output devices of the time.

The move away from the mechanical gears and cranks took place in the 1930s with the availability of electromagnetic relays. George Stibitz, a Bell Telephone Laboratories mathematician, developed the first demonstrable electromechanical machine. It was exhibited at a meeting of the American Mathematical Society at Dartmouth College in 1940. Independently, Konrad Zuse of Germany built several relay machines. But his work was kept secret due to Germany's involvement in World War II. His machines were later destroyed by the Allied bombing. Others involved in the development of relay generation machines include John Atanasoff of Iowa State College.

1.9.2 Vacuum Tube Generation: Around the 1940s and 1950s

Vacuum tubes brought computers from the mechanical to the electronic era. Clearly, delays were substantially reduced. Presper Eckert and John Mauchly of the University of Pennsylvania designed the ENIAC (electronic numerical integrator and computer) system, which became operational in World War II. It used about 18,000 vacuum tubes and could perform nearly 5000 additions per second. There was no concept of the program as we know it. Reprogramming the machine took most of a day rewiring! It was under these circumstances that John von Neumann, along with others, proposed the concept of the stored program that we use even today. The idea was to keep a program in the memory and read the instructions from it, rather than hardwiring the program. He also proposed an architecture that clearly identified the components we have presented in this chapter: ALU, control, input, output, and memory. This architecture is known as the *von Neumann architecture.*

Magnetic core memories were invented during this timeframe. Core memories were used until the 1970s! Even today, we use the term core to mean the main memory. You might have heard about "core dumps" to check the contents of main memory. There is also a current research area that works on out-of-core computations. As mentioned before, Maurice Wilkes proposed the microprogramming concept during this time.

1.9.3 Transistor Generation: Around the 1950s and 1960s

The invention of the transistor at Bell Labs in 1948 has led to the next generation of computer systems. Transistors have several significant improvements over the previous generation's basic building block, the vacuum tube. Compared to vacuum tubes, transistors are small in size, consume substantially less power, and have much lower failure rates.

Magnetic core memories were still widely used for main memory. High-level languages such as FORTRAN were developed to ease the programming of mathematical and scientific applications. IBM became a dominant player during this period.

1.9.4 IC Generation: Around the 1960s and 1970s

The next generation systems benefited from our ability to put several transistors on a single silicon chip. This has led to the development of integrated circuits (ICs), in which an entire circuit is fabricated on a single chip. Some of these ICs are still available on the market (see our discussion of digital logic chips in the next chapter). Texas Instruments and Fairchild Semiconductor made ICs for sale in 1958.

ICs quickly replaced the magnetic core memory. IBM still held its dominant position with the introduction of mainframe systems. There have been developments on the operating system front as well. Multiprogramming and time-sharing were proposed to improve response times and system efficiency. The arrival of the disk drive definitely helped in this endeavor. IBM introduced their System/360 model in the mid-1960s. Digital Equipment Corporation (DEC) (now part of Compaq) started selling minicomputers to universities.

1.9.5 VLSI Generations: Since the Mid-1970s

Ever since ICs were made possible, the density has been growing at a phenomenal rate. By the mid-1970s, more than 10,000 components could be fabricated on a single chip. This has led to the development of smaller processors on a chip. These processors were called *microprocessors,* to contrast them with the processors in mainframe systems from IBM and minicomputers from DEC.

Intel produced the first microprocessor 4004 in 1971. It required only 23,000 transistors. To gauge the progress made since then, compare this number with the number of transistors in the Pentium when it was introduced in 1993: 3 million. We now have the technology to put 100 million transistors on a chip.

With the introduction of personal computers (PCs), several new players came into existence. These are the names that need no introduction: Intel, Microsoft, Apple, and so on. As we have discussed in this chapter, technological advances coupled with architectural improvements continue to lead computer system design.

We are in the midst of an information revolution. If we can get biological computers to work, that would qualify as the next generation. Imagine that in 20 years, the Pentiums and PowerPCs will be looked upon as primitive processors!

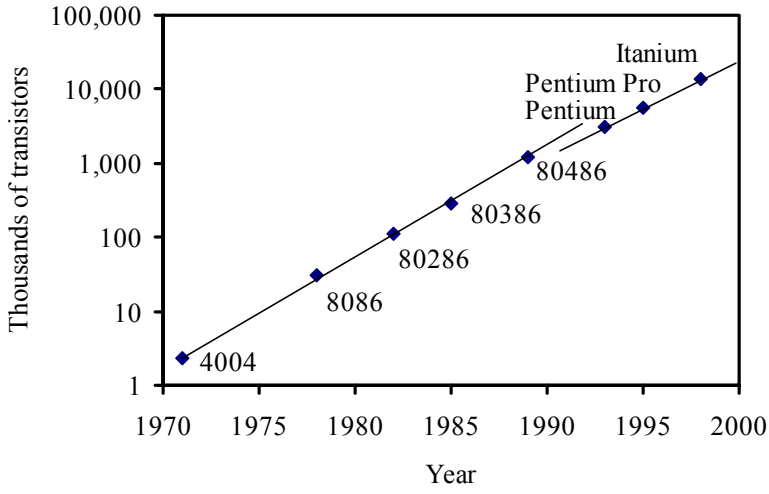


Figure 1.20 Transistor density in Intel processors.

1.10 Technological Advances

The current trend toward miniaturization has its roots in the IC concept. Every component of a computer system has experienced phenomenal improvement over time. It is not only the components that we discussed in this chapter—processor, memory, disk storage, and buses—but also communications networks are experiencing similar growth. This integration of computer and communication bodes well for the future generations of systems. This section briefly comments on the rate of growth for some of these components.

The primary driving force for the improvement in processors and memory is our ability to pack more and more transistors onto a single chip. Gordon Moore, cofounder of Intel, observed in 1965 that the transistor count per chip was doubling every year. This observation, known as *Moore's law*, continued to hold into the early 1970s. Then things slowed down a bit as shown in Figure 1.20. Until the 1990s, the transistor count doubled every 18 to 24 months. In the 1990s, it slowed down further, doubling about every 2.5 years. This tremendous rate of growth in density allows us to design more powerful processors and larger capacity memories. In addition, the higher density has the following implications:

- We get increased reliability due to fewer external connections,
- Our ability to reduce the size is leading to the current emphasis on device miniaturization,
- We get increased speed due to shorter path lengths.

Memory capacities are also improving at a similar pace. Until the 1990s, dynamic RAMs (DRAMs), which is what we use for our main memory, quadrupled every three years. This rate of growth in capacity gives us the same average rate (of doubling every 18 months) as the processors. The recent growth in density appears to have slowed down to quadrupling every

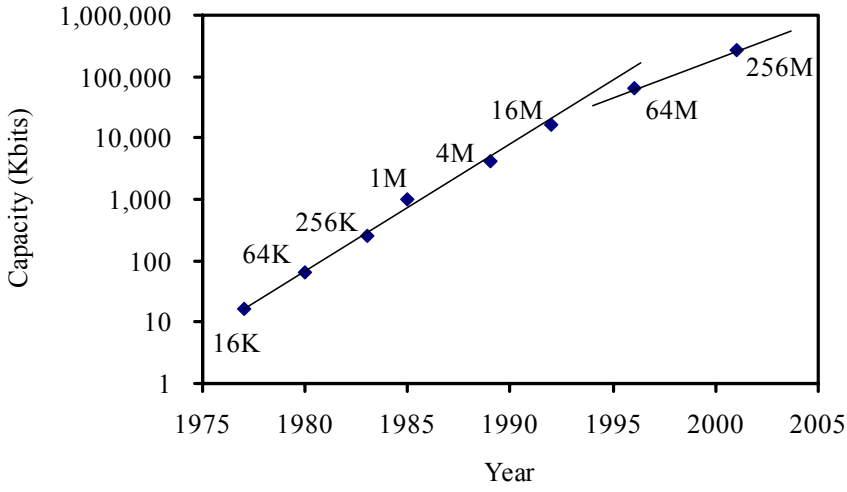


Figure 1.21 Memory bit capacity of DRAMs.

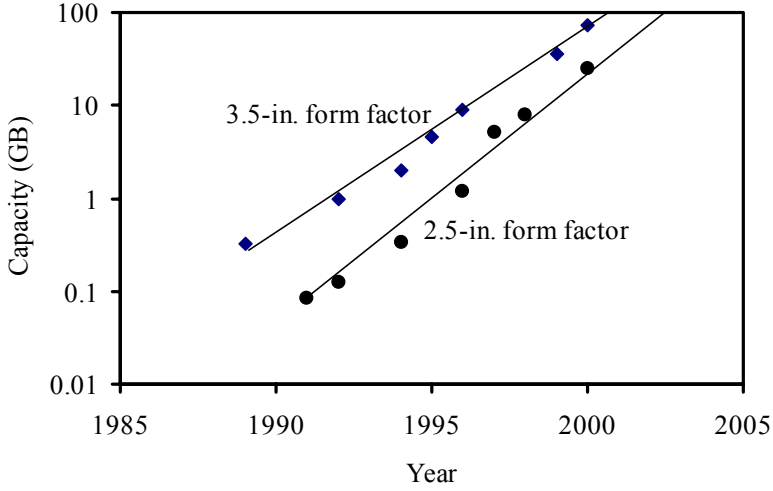


Figure 1.22 Capacities of 3.5-in. and 2.5-in. form factor disk drives from IBM.

four to five years as shown in Figure 1.21. Disk drive capacities are also increasing substantially as shown in Figure 1.22.

We talked a great deal about the capacities. From the capacity point of view, all three components seem to progress in unison. However, when you look at the operational speed, processors are way ahead of the memory access times for DRAMs. This speed differential is, in part, due

to the improvements in architecture. For example, with pipelining and superscalar designs, we can increase the rate of instruction execution. Currently, processors seem to improve at 25 to 40% per year, whereas memory access times are improving at about 10% per year. As we have seen in this chapter, we need to bridge this gap. For example, we can use caches to bridge the speed gap between processors and memory, and between memory and disk storage.

1.11 Summary and Outline

How one views a computer system depends on the type of use and level of interaction. A user interested in running a specific application program does not need to know a lot of internal system details. A rudimentary knowledge of how to turn the system on, how to execute a program (e.g., point-and-click), and a few details about the user-level interface provided by the system are sufficient. If you are a programmer, you need to know a lot more. Even within this group, the kind of language you use determines the level of detail you need to know. An assembly language programmer should know more details about the components such as the processor, memory, and I/O subsystem. A Java programmer, on the other hand, need not know all these details.

In this chapter, we have essentially presented an overview of computer system organization and architecture. Our goal in presenting this information is to give you a gentle, high-level introduction to the book's subject matter. In that sense, this chapter serves as an introduction to the entire book.

We have divided the rest of the book into seven parts. Part II covers digital logic design concepts. It consists of three chapters that give details on the nuts and bolts of computer systems. The first chapter of Part VII is also related to this part as it deals with the design at the digital logic level. These four chapters give you a good grounding on the basic hardware devices used to implement major functional units of a computer system.

System interconnects are covered mainly in Part III. This part consists of a single chapter, which covers internal buses including PCI and PCMCIA. There are two other chapters that deal with buses as well. Chapter 6 describes onchip buses required to implement the datapath of a processor. External buses, including USB and IEEE 1394, are described in Chapter 19.

Processor details are covered in several parts. Part IV presents the basic processor design issues, details about pipelining, and vector and Pentium processors. RISC processor details are covered in Part VI. This part discusses the PowerPC, Intel Itanium, and MIPS. A complete chapter is dedicated to MIPS assembly language programming. SPARC processor details are given in Appendix H. Pentium assembly language programming is in Part V, which consists of five chapters. In this part, we devote a complete chapter to describe the interaction between assembly language and high-level languages. We use C as the representative of a high-level language.

Memory design and related topics are presented in Part VII. This part consists of three chapters. The first chapter describes memory design at the digital logic level. The remaining two chapters give details on cache and virtual memories.

Part VIII presents details on the I/O subsystem in two chapters. The first chapter covers programmed I/O, DMA, and external buses. We use an example assembly language program to describe how programmed I/O works. In the next chapter, we redo the same example using interrupts to bring out the similarities and differences between programmed I/O and interrupt-driven I/O. This chapter deals with the interrupt mechanism, focusing on the Pentium interrupts. We use assembly language programs to explain some of the features of the interrupt mechanism.

The appendices provide a variety of reference information. Topics covered here include computer arithmetic as well as details on assembling, linking, and debugging assembly language programs. We also present details about digital circuit simulators and the MIPS simulator. For your easy reference, Pentium instructions are given in one of the appendices.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Address bus
- Assembler
- Assembly language
- Big endian
- Byte addressable memory
- Byte ordering
- CISC
- Control bus
- Data bus
- Datapath
- Execution cycle
- Harvard architecture
- I/O controller
- I/O ports
- Little endian
- Machine language
- Microprogram
- Memory access time
- Memory address space
- Memory cycle time
- Memory operations
- Memory read cycle
- Memory write cycle
- Pipelining
- Programmer productivity
- RISC
- Space-efficiency
- System bus
- Time-efficiency
- von Neumann architecture
- Wait cycles

1.12 Exercises

1–1 Describe the instruction execution cycle.

1–2 What are the main components of a system bus? Describe the functionality of each component.

1–3 What is the purpose of providing various registers in a processor?

1–4 What is the relationship between assembly language and machine language?

1–5 Why is assembly language called a low-level language?

1–6 What are the advantages of high-level languages compared to assembly language?

1–7 Why do we still program in assembly language?

1–8 What is the purpose of the datapath in a processor?

1–9 What is the role of the microprogram?

1–10 What benefits do we get by using pipelining in a processor?

1–11 Explain why CISC processors tend to use microprogramming but not the RISC processors.

1–12 Describe the little-endian and big-endian byte ordering schemes.

1–13 What is a byte addressable memory?

1–14 If a processor has 16 address lines, what is the physical memory address space of this processor? Give the address of the first and last addressable memory locations in hex.

1–15 What is the purpose of cache memory?

1–16 What is the purpose of virtual memory?

1–17 What is an I/O port?

1–18 What is bus arbitration?

Chapter 2

Digital Logic Basics

Objectives

- To introduce basic logic gates;
- To discuss properties of logical expressions;
- To show how logical expressions can be simplified and implemented;
- To illustrate the digital logic design process.

Viewing computer systems at the digital logic level exposes us to the nuts and bolts of the basic hardware. We cover the necessary digital logic background in three chapters. In this first chapter, we look at the basics of digital logic. We start off with a look at the basic gates such as AND, OR, and NOT gates. The completeness property and implementation of these gates using transistors are discussed next. We then describe how logical functions can be derived from the requirement specifications.

We introduce Boolean algebra to manipulate logical expressions. Once a logical expression is obtained, we have to optimize (simplify) this expression so that we can use minimum hardware to implement the logical function. There are several methods to simplify logical expressions. We present three methods: the algebraic, Karnaugh map, and Quine–McCluskey methods. The first one is based on Boolean algebra and is difficult to use as a general method. The Karnaugh map method is a graphical method suitable for simplifying logical expressions with a small number of variables. The last method is a tabular one and is suitable for simplifying logical expressions with a large number of variables. Furthermore, the Quine–McCluskey method is suitable for automating the simplification process. Toward the end of the chapter, we take a look at how we can implement logical functions using gates other than the three basic gates.

2.1 Introduction

The hardware that is responsible for executing machine language instructions can be built using a large number of a few basic building blocks. These building blocks are called *logic gates*. These logic gates implement the familiar logical operations such as AND, OR, NOT, and so on, in hardware. For example, as we show later, we can build hardware circuits using only AND and NOT gates or their equivalent. The purpose of this chapter is to provide the basics of the digital hardware.

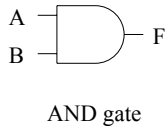
Logic gates are in turn built using transistors. One transistor is enough to implement a NOT gate. But we need three transistors to implement the AND gate. In that sense, transistors are the basic electronic components of digital hardware circuits. For example, the Pentium processor introduced in 1993 consists of about 3 million transistors. In 2000, it was possible to design chips that use 100 million transistors. How do the designers of these chips manage such complexity? Obviously, they need a higher-level abstraction to aid the design process. Of course, design automation is also a must. For example, logic gates such as AND and OR represent a higher-level abstraction than the basic transistor level. After going through this chapter, you will realize that even this level of abstraction is not good enough; there are still millions of gates to handle in designing a processor. In the next two chapters, we discuss even higher levels of abstractions.

Our discussion of digital logic design is divided into three chapters. This chapter deals with the basics of digital logic gates and their implementation. As we mentioned, we need to devise higher-level abstractions to reduce the complexity of digital circuit design. We look at two higher levels of abstractions—combinational and sequential circuits—in the next two chapters. In combinational circuits, the output of the circuit depends solely on the *current* inputs applied to the circuit. The adder is an example of a combinational circuit. The output of an adder depends only on the current inputs. On the other hand, the output of a sequential circuit depends not only on the current inputs but also on the past inputs. That is, output depends both on the current inputs as well as on how it got to the current state. For example, in a binary counter, the output depends on the current value. The next value is obtained by incrementing the current value (in a way, the current state represents a snapshot of the past inputs). That is, we cannot say what the output of a counter will be unless we know its current state. Thus, the counter is a sequential circuit. We discuss combinational circuits in Chapter 3 and sequential circuits in Chapter 4. The circuits we design in this chapter are also combinational circuits.

2.2 Basic Concepts and Building Blocks

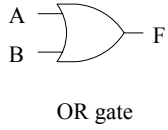
2.2.1 Simple Gates

You are familiar with the three basic logical operators: AND, OR, and NOT. Digital circuits to implement these and other logical functions are called gates. Figure 2.1 shows the symbol notation used to represent AND, OR, and NOT gates. We have also included the truth table for each gate. A *truth table* is a list of all possible input combinations and their corresponding output. For example, if you treat a logical zero as representing false and a logical 1 truth, you can see that the truth table for the AND gate represents the logical AND operation.



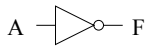
AND gate

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



OR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1



NOT gate

A	F
0	1
1	0

Logic symbol Truth table

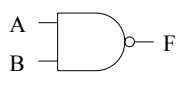
Figure 2.1 Basic logic gates: Logic symbols and truth tables.

In logical expressions, we use the dot, +, and overbar to represent the AND, OR, and NOT operations, respectively. For example, the output of the AND gate in Figure 2.1 is written as $F = A \cdot B$. Assuming that single letters are used for logical variables, we often omit the dot and write the previous AND function as F = A B. Similarly, the OR function is written as F = A + B. The output of the NOT gate is expressed as $F = \overline{A}$. Some authors use a prime to represent the NOT operation as in F = A′ mainly because of problems with typesetting the overbar.

The precedence of these three logical operators is as follows. The AND operator has a higher precedence than the OR operator, whereas the unary NOT operator has the highest precedence among the tree operators. Thus, when we write a logical expression such as $F = A\ \overline{B} + \overline{A}\ B$, it implies $F = (A\ (\overline{B})) + ((\overline{A})\ B)$. As in arithmetic expressions, we can use parentheses to override the default precedence.

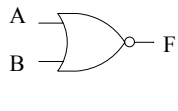
Even though the three gates shown in Figure 2.1 are sufficient to implement any logical function, it is convenient to implement certain other gates. Figure 2.2 shows three popularly used gates. The NAND gate is equivalent to an AND gate followed by a NOT gate. Similarly, the NOR gates are a combination of OR and NOT gates. It turns out that, contrary to our intuition, implementing the NAND and NOR gates requires only two transistors whereas the AND and OR gate implementations require three transistors.

The exclusive-OR (XOR) gate generates a 1 output whenever the two inputs differ. This property makes it useful in certain applications such as parity generation. Another interesting and useful gate that is not shown here is the exclusive-NOR gate. This gate is equivalent to an



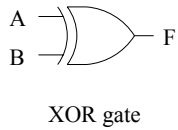
NAND gate

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



NOR gate

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



XOR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Logic symbol Truth table

Figure 2.2 Some additional useful gates.

XOR followed by a NOT gate. This gate output, which is a complement of the XOR gate, is 1 whenever the two inputs match. The exclusive-NOR gate is also called the *equivalence* or *coincidence* gate. All the gates we have discussed here are available commercially (see page 50 for some sample gates).

2.2.2 Completeness and Universality

Number of Functions

Let us assume that we are working with two logical variables. We know that we can define various functions on two variables. These include the AND, OR, NAND, NOR, and XOR functions discussed in the last section. The question that we want to answer is: How many different logical functions can we define on N logical variables? Once we know the answer to this question, we can use this information, for example, to make a statement about the universality of a gate. For example, the NAND gate is universal. This means that we can implement any logical function using only the NAND gates (we can use as many NAND gates as we want).

To get an intuition, let us focus on two variables. Since two variables can have four combinations of inputs (i.e., four rows in the truth table) and we can assign a 1 or 0 as output for each row, we can define 16 different functions as shown in Table 2.1.

Table 2.1 Number of functions that can be defined on two logical variables

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Looking at this table, we see that some functions are useless (e.g., F_0 and F_{15}) as they are independent of the input. There are some other functions that we can readily identify with the logic gates described in the last section (e.g., F_1 is the AND function, and F_6 is the XOR function).

We can generalize this to N logical variables by noting that there are 2^N rows in the truth table of an N-variable logical expression. Thus, the number of functions that can be defined on N variables is 2^{2^N}.

Complete Sets

We say that a set of gates is *complete* if we can implement any logical function using only the gates in this complete set. What this tells us is that, theoretically, we don't need gates outside this set to implement a logical function. Here are some complete sets:

{AND, OR, NOT}
{AND, NOT}
{OR, NOT}
{NAND}
{NOR}

A complete set is *minimal* if it does not contain redundant elements. That is, if we delete an element from the set, it should not remain complete. In the above complete sets, we see that the set AND, OR, NOT is not minimal as we can remove either AND or OR (but not both) to get another complete set.

How do we prove that a set is complete? Essentially, we have to show that, using only the gates in the set, we can construct AND, OR, and NOT gates. Figure 2.3 shows how we can construct these three gates by using only the NAND gates. A similar proof is given in Figure 2.4 for the NOR gates. NAND and NOR gates are called *universal gates* because we can implement any logical function using only the NAND or NOR gates.

We close this section with a final note on equivalence proofs. It is not strictly necessary to construct AND, OR, and NOT gates as we did in Figures 2.3 and 2.4. Assuming that we proved the completeness of {AND, NOT} and {OR, NOT}, it is sufficient to construct either AND and NOT or OR and NOT gates. We leave it as an exercise to show how OR gates can be

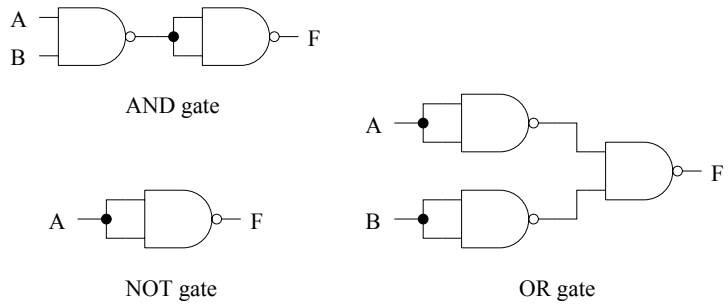


Figure 2.3 Implementation of AND, OR, and NOT gates by using only NAND gates.

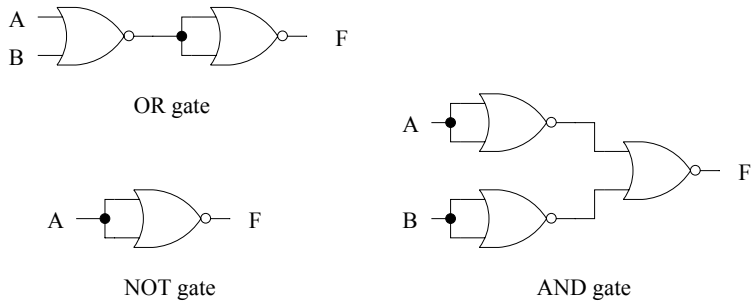


Figure 2.4 Implementation of AND, OR, and NOT gates by using only NOR gates.

constructed using only AND and NOT gates. Similarly, you can show that the AND gate can be constructed using only OR and NOT gates.

2.2.3 Implementation Details

Transistor Implementation

The concepts involved in implementing digital circuits can be described by looking at their transistor implementations. Figure 2.5 shows a transistor with three connection points: base, collector, and emitter. A transistor can be operated in either a linear or switching mode. In linear mode, a transistor amplifies the input signal applied to the base. This is the mode the transistor operates in your amplifier. In digital circuits, the transistor operates in the switching mode. In this mode, the transistor acts as a switch between the collector and emitter points. The voltage applied to the base input of the transistor determines whether the switch is open (open circuit between collector and emitter points) or closed (short circuit between collector and emitter). A high voltage (typically above 2 V) causes the transistor to act as a closed switch, and a low voltage (typically below 0.8 V) forces the transistor to act as an open switch.

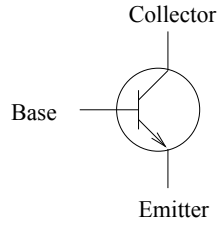


Figure 2.5 A transistor.

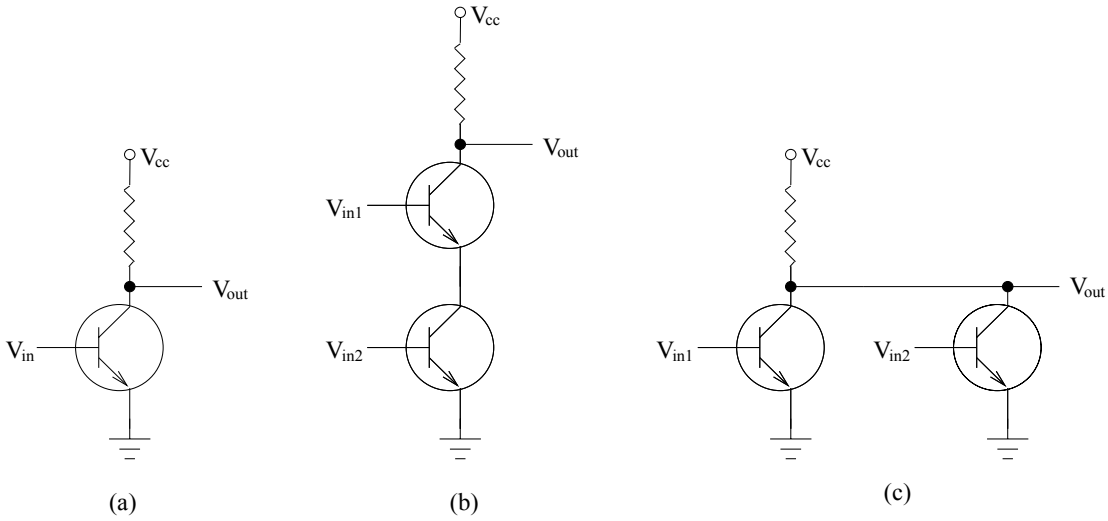


Figure 2.6 Implementation of simple gates: (a) NOT gate; (b) NAND gate; (c) NOR gate.

When the transistor behaves as described, it is fairly simple to build a NOT gate as shown in Figure 2.6*a*. The collector of the transistor is tied to V_{cc} through a resistor. V_{cc} is typically 5 V. Assuming that 0 V represents logical 0 and +5 V represents a logical 1, we can see that the single transistor implementation shown in Figure 2.6*a* corresponds to a NOT gate. When V_{in} is low, there is an open circuit between the collector and emitter. Thus, no current flows through the resistor. This causes the V_{out} to be +5 V. On the other hand, when a high voltage is applied to V_{in}, there is a short circuit between the collector and emitter points, which results in a low V_{out}.

It is left as an exercise to verify that the NAND gate is implemented by the circuit shown in Figure 2.6*b* and the NOR gate by Figure 2.6*c*. It is interesting to note that AND gate implementation actually requires three transistors as it is implemented as a NAND gate followed by a NOT gate.

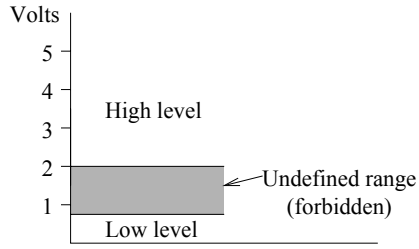


Figure 2.7 Low and high logic voltage levels for TTL logic circuits.

In closing this section, we briefly mention the key technologies used to manufacture digital circuits. There are two main semiconductor technologies: bipolar and MOS (metal oxide semiconductor). Bipolar implementations are, in general, faster than the MOS implementations. The two major bipolar types are the TTL (transistor–transistor logic) and ECL (emitter-coupled logic). Relatively speaking, TTL is slower than ECL circuits. If you open your PC and look at the motherboard, you will see quite a few of these TTL chips (described next).

MOS technology allows us to build high-density chips as it consumes less power and takes less space on the chip compared to their bipolar cousins. In MOS technology, transistors are implemented in a different way than the bipolar implementations we have discussed. However, logically, it still acts as a switch. Even though NMOS, PMOS, and HMOS types exist, CMOS (complementary MOS) is the dominant technology used to implement processors and memories. For example, the Pentium processor uses about 3 million transistors.

Gallium arsenide (GaAs) technology provides an alternative to the semiconductor technology. It has superior speed characteristics when compared to the bipolar technology. However, GaAs technology poses several difficulties in manufacturing (such as poor reliability) that limited its applicability to high-density gate implementations such as microprocessors.

Examples of Logic Chips

A small set of independent logic gates (such as AND, NOT, NAND, etc.) are packaged into an integrated circuit chip, or "chip" for short. The smallest of these ICs uses a 14-pin DIP (dual inline package). Some example chips are shown in Figure 2.8. There are two rows of pins (the reason why this package is called a dual inline package) numbered 1 through 14. Pin 7 is Ground and pin 14 is V_{cc}. A small notch or a dot is used for proper orientation of the chip (i.e., to identify pin 1).

The V_{in} input should be less than 0.8 V to be treated as a low-level voltage and greater than 2 V for high level as shown in Figure 2.7. The voltage range in between these two levels is forbidden. The output voltage levels produced are less than 0.4 V (for low level) and 2.4 V (for high level). For *positive logic*, the low-voltage level is interpreted as 0 and the high level as 1. For *negative logic*, the low-voltage level is treated as representing 1 and the high level as 0. By default, we use the positive logic in our discussion.

There is a *propagation delay* associated with each gate. This delay represents the time required for the output to react to an input. The propagation delay depends on the complexity of the circuit and the technology used. Typical values for the TTL gates are in the range of a few nanoseconds (about 5 to 10 ns). A nanosecond (ns) is 10^{-9} second.

In addition to propagation delay, other parameters should be taken into consideration in designing and building logic circuits. Two such parameters are fanin and fanout. *Fanin* specifies the maximum number of inputs a logic gate can have. *Fanout* refers to the driving capacity of an output. Fanout specifies the maximum number of gates that the output of a gate can drive.

These ICs are called small-scale integrated (SSI) circuits and typically consist of about 1 to 10 gates. Medium-scale integrated (MSI) circuits represent the next level of integration (typically between 10 and 100 gates). Both SSI and MSI were introduced in the late 1960s. LSI (large-scale integration), introduced in early 1970s, can integrate between 100 and 10,000 gates on a single chip. The final degree of integration, VLSI (very large scale integration), was introduced in the late 1970s and is used for complex chips such as microprocessors that require more than 10,000 gates.

2.3 Logic Functions

2.3.1 Expressing Logic Functions

Logic functions can be specified in a variety of ways. In a sense their expression is similar to problem specification in software development. A logical function can be specified verbally. For example, a majority function can be specified as: Output should be 1 whenever the majority of the inputs is 1. Similarly, an even-parity function can be specified as: Output (parity bit) is 1 whenever there is an odd number of 1s in the input. The major problem with verbal specification is the imprecision and the scope of ambiguity.

We can make this specification precise by using a truth table. In the truth table method, for each possible combination of the input, we specify the output value. The truth table method makes sense for logical functions as the alphabet consists of only 0 and 1. The truth tables for the 3-input majority and even-parity functions are shown in Table 2.2.

The advantage of the truth table method is that it is precise. This is important if you are interfacing with a client who does not understand other more concise forms of logic function expression. The main problem with the truth table method is that it is cumbersome as the number of rows grows exponentially with the number of logical variables. Imagine writing a truth table for a 10-variable function!

We can also use logical expressions to specify a logical function. Logical expressions use logical operators as discussed in Section 2.2. The logical expressions for our 3-input majority and even-parity functions are shown below:

- 3-input majority function = $A\ B + B\ C + A\ C$,
- 3-input even-parity function = $\overline{A}\ \overline{B}\ C + \overline{A}\ B\ \overline{C} + A\ \overline{B}\ \overline{C} + A\ B\ C$.

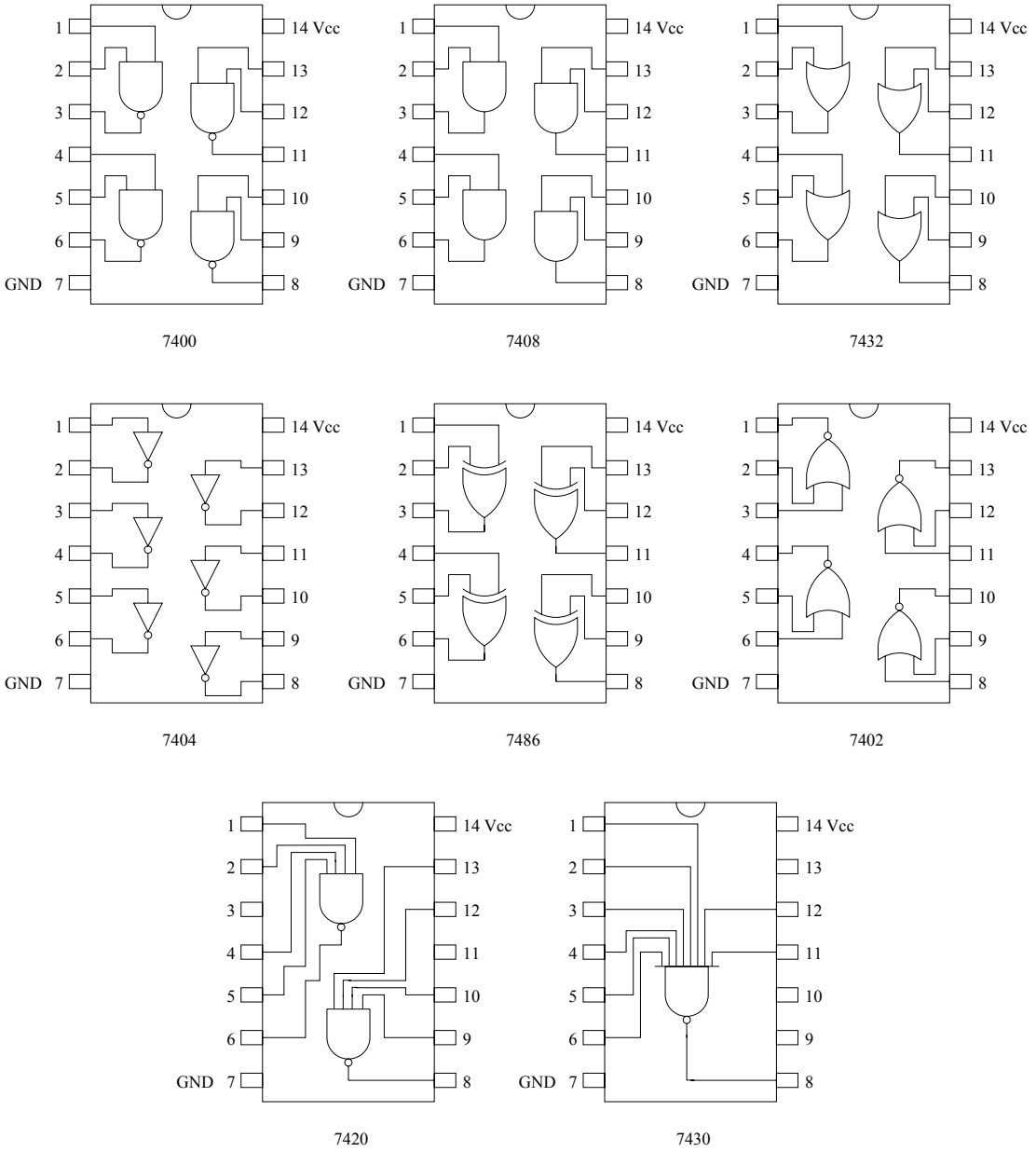


Figure 2.8 Some examples of commercial TTL logic circuits.

Table 2.2 Truth tables for the majority and even-parity functions

Majority function

A	B	C	F_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Even-parity function

A	B	C	F_2
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

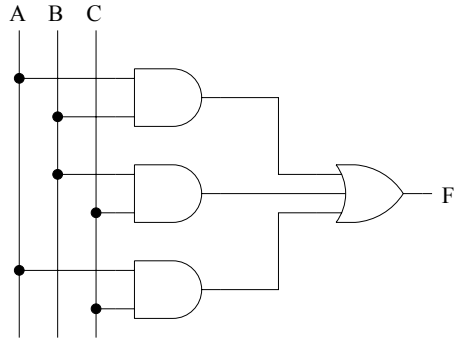


Figure 2.9 Three-input majority function.

An advantage of this form of specification is that it is compact while it retains the precision of the truth table method. Another major advantage is that logical expressions can be manipulated to come up with an efficient design. We say more on this topic in Section 2.7.1.

The final form of specification uses a graphical notation. Figure 2.9 shows the logical circuit to implement the 3-input majority function. As with the last two methods, it is also precise but is more useful for hardware engineers to implement the logical function.

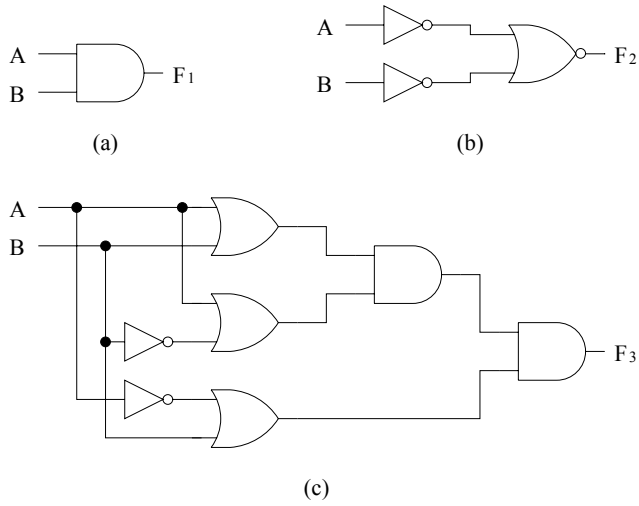


Figure 2.10 Three circuit designs to implement F = A B logical function.

2.3.2 Logical Circuit Equivalence

Logical functions can be implemented in a variety of ways. If two logical circuits are performing the same logical function F, we say that these two circuits are equivalent. Establishing logical circuit equivalence is important because it allows us to pick an efficient design for implementation. By "efficient" we mean a circuit that uses a minimum number of gates. Later we show that we can also talk about minimizing the number of chips rather than the gate count.

To illustrate the point, look at the three circuits shown in Figure 2.10. The legend of the figure claims that all three are performing a simple AND operation. We discuss later how we can verify this claim. If we take the claim to be true, these three circuits are equivalent. Here, we obviously pick the first circuit that uses a single 2-input AND gate.

Now, how do we prove that these three logic circuits are equivalent? This is a two-step process. First, we have to derive the logical expression for each circuit. Then, we have to show that the three logical expressions are equivalent.

Deriving Logical Expressions

Deriving a logical expression from a given logical circuit involves tracing the path from input to output and writing intermediate logical expressions along the path. The process is illustrated in Figure 2.11. The output of the top OR gate is (A + B). Noting the inputs of the middle OR gate are A and $\overline{B}$, we write the logical expression for the output of this gate as (A + $\overline{B}$). Continuing the process finally leads us to the following expression for the logical function F_3,

$$F_3 = (A + B)\ (A + \overline{B})\ (\overline{A} + B).$$

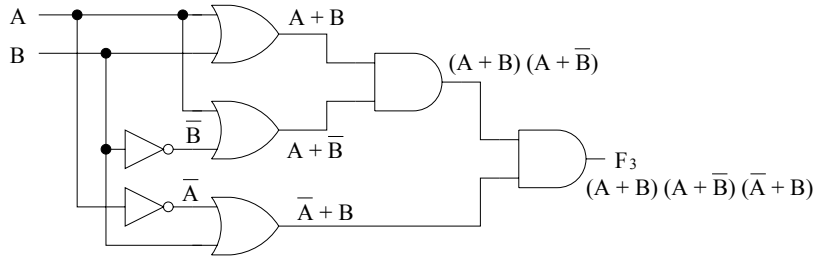


Figure 2.11 Deriving the logical expression from the circuit diagram.

Table 2.3 Truth table to prove that F_1 and F_3 functions are equivalent

A	B	$F_1 = AB$	$F_3 = (A + B)(\overline{A} + B)(A + \overline{B})$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

To show that this logical circuit is equivalent to a 2-input AND gate, we have to show that the logical expression for F_3 reduces to A B. We focus on this aspect next.

Establishing Logical Equivalence

There are two ways of establishing logical equivalence of two functions. The first is the truth table method. The other method involves manipulating logical expressions by applying Boolean algebra rules. We discuss the truth table method now. The Boolean algebra method is described in Section 2.4.2.

The truth table method is conceptually simple but tedious for logical expressions that involve more than a few logical variables. The principle is to look at each possible combination of the input and test if the two functions give the same output. If so, the two functions are equivalent. This process is shown in Table 2.3. Notice the use of two output columns, one for each function. Since the outputs of these two functions are identical, we conclude that functions F_1 and F_3 are equivalent.

Since F_1 and F_3 are derived from the circuits in Figures 2.10*a* and 2.10*c*, we conclude that these two circuits are equivalent. We leave it as an exercise for the reader to show that Figures 2.10*a* and 2.10*b* are equivalent.

2.4 Boolean Algebra

This section discusses how we can use the Boolean algebra to manipulate logical expressions. We need Boolean identities to facilitate this manipulation. These are discussed next. Following this discussion, we show how the identities developed can be used to establish logical equivalence. In Section 2.7.1, we show how these identities can also be used to simplify logical expressions.

2.4.1 Boolean Identities

Table 2.4 presents some basic Boolean laws. For most laws, there are two versions: an **and** version and an **or** version. If there is only one version, we list it under the **and** version. We can transform a law from the **and** version to the **or** version by replacing each 1 with a 0, 0 with a 1, + with a ·, and · with a +. This relationship is called *duality*.

The last law is particularly interesting as it is useful in moving negation in and out of logical expressions. For example, de Morgan's law is useful in coming up with a NAND or NOR gate based design (see Section 2.10.1).

The complement law suggests that if x and y are complements of each other, the following must be true: $x \cdot y = 0$ and $x + y = 1$. This observation is useful in proving de Morgan's law (see Exercise 2–12).

We can use the truth table method (as in Table 2.3) to show that these laws hold. We can also prove some of these laws. To illustrate the process, we prove the absorption law.

$$
\begin{aligned}
x &= x \cdot (x + y) && \\
&= (x \cdot x) + (x \cdot y) && \text{(Distribution law)} \\
&= x + (x \cdot y) && \text{(Idempotence law)} \\
&= (x \cdot 1) + (x \cdot y) && \text{(Identity law)} \\
&= x \cdot (1 + y) && \text{(Distribution law)} \\
&= x \cdot 1 && \text{(Null law)} \\
&= x && \text{(Identity law).}
\end{aligned}
$$

Notice that in our attempt to prove the **and** version of the absorption law, we have also proved the **or** version.

2.4.2 Using Boolean Algebra for Logical Equivalence

We can use a similar procedure to establish logical equivalence of two logical functions. Typically, we start with one function and derive the other function to show the logical equivalence. As an example, we show that functions F_1 and F_3 in Table 2.3 are equivalent.

Table 2.4 Boolean laws

Name	**and** version	**or** version
Identity	$x \cdot 1 = x$	$x + 0 = x$
Complement	$x \cdot \overline{x} = 0$	$x + \overline{x} = 1$
Commutative	$x \cdot y = y \cdot x$	$x + y = y + x$
Distribution	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	$x + (y \cdot z) = (x + y) \cdot (x + z)$
Idempotent	$x \cdot x = x$	$x + x = x$
Null	$x \cdot 0 = 0$	$x + 1 = 1$
Involution	$\overline{\overline{x}} = x$	—
Absorption	$x \cdot (x + y) = x$	$x + (x \cdot y) = x$
Associative	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x + (y + z) = (x + y) + z$
de Morgan	$\overline{x \cdot y} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x} \cdot \overline{y}$

$$
\begin{aligned}
\mathrm{A\,B} &= \underbrace{(\mathrm{A} + \mathrm{B})\,(\mathrm{A} + \overline{\mathrm{B}})}_{(\mathrm{A\,A}+\mathrm{A}\,\overline{\mathrm{B}}+\mathrm{B\,A}+\mathrm{B}\,\overline{\mathrm{B}})}\,(\overline{\mathrm{A}} + \mathrm{B}) \\
&= (\underbrace{\mathrm{A\,A}}_{\mathrm{A}} + \underbrace{\mathrm{A}\,\overline{\mathrm{B}} + \mathrm{B\,A}}_{\mathrm{A}(\overline{\mathrm{B}}+\mathrm{B})} + \underbrace{\mathrm{B}\,\overline{\mathrm{B}}}_{0})\,(\overline{\mathrm{A}} + \mathrm{B}) \\
&= (\mathrm{A} + \underbrace{\mathrm{A}\,(\overline{\mathrm{B}} + \mathrm{B})}_{\mathrm{A}} + 0)\,(\overline{\mathrm{A}} + \mathrm{B}) \\
&= \underbrace{(\mathrm{A} + \mathrm{A} + 0)}_{\mathrm{A}}\,(\overline{\mathrm{A}} + \mathrm{B}) \\
&= \mathrm{A}\,(\overline{\mathrm{A}} + \mathrm{B}) = \mathrm{A}\,\overline{\mathrm{A}} + \mathrm{A\,B} = 0 + \mathrm{A\,B} = \mathrm{A\,B}\,.
\end{aligned}
$$

Sometimes it may be convenient to reduce both sides to the same expression in order to establish equivalence.

2.5 Logic Circuit Design Process

To provide proper perspective to our discussion of the remaining topics in this chapter, we briefly review a simplified digital circuit design process shown in Figure 2.12. As in the programming activity, the input specification may be given in plain English. For example, this description can be something like, “Design a circuit to implement the majority function on

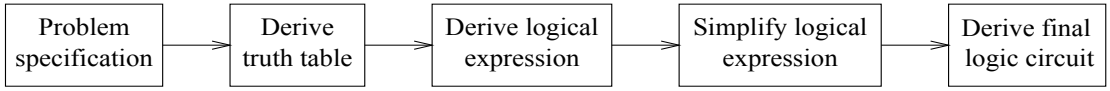


Figure 2.12 A simple logic circuit design process.

three inputs." This kind of description makes a lot of assumptions such as the definition of the majority function. Even a simple function such as the majority function can be defined in several ways. We have been using a simple majority function in our discussion with each input having the same weight. However, we can define other majority functions. For example, the weight of inputs may not be the same, or somebody may have veto power on the final outcome as in the UN Security Council (see Exercises 2–9 and 2–10). Thus, our next job is to derive a precise description of the problem from this imprecise (possibly incomplete) description of the problem. If we are going to design a combinational logic circuit, for example, we can use a truth table to precisely define the problem specification.

How do we get the final logic circuit from this truth table? We use two steps to get the final circuit design as shown in Figure 2.12. We derive a logical expression from the truth table. The logical expression may be in sum-of-products or product-of-sums form, as we show in the next section. We, however, do not implement this logical expression directly as it may not be in a minimal form to get an efficient design. We need to simplify this logical expression to minimize implementation cost using one of the methods we discuss in Section 2.7. We derive the final logic circuit design from this simplified logical expression.

Note that minimizing implementation is often interpreted as minimizing the number of gates. To a degree of approximation, this is true. We follow this objective in our simplification methodologies. Observe, however, that when implementing a digital circuit, we are actually concerned with the number of chips required to implement the circuit, not the number of gates.

We do not need these two steps if we intend to implement the logical circuit using building blocks like multiplexers and PLAs. In that case, our implementation follows directly from the truth table. Multiplexers and PLAs are discussed in Chapter 3.

2.6 Deriving Logical Expressions from Truth Tables

We can write a logical expression from a truth table in one of two forms: *sum-of-products* (SOP) and *product-of-sums* (POS) forms. In sum-of-products form, we specify the combination of inputs for which the output should be 1. In product-of-sums form, we specify the combinations of inputs for which the output should be 0. As in Section 2.4.1, you see the duality of these two forms.

2.6.1 Sum-of-Products Form

In this form, each input combination for which the output is 1 is expressed as an **and** term. This is the *product term* as we use $\cdot$ to represent the AND operation. These product terms are ORed together. That is why it is called sum-of-products as we use + for the OR operation to get the

final logical expression. In deriving the product terms, we write the variable if its value is 1 or its complement if 0. We now consider two examples to illustrate this process.

Example 2.1: Let us first look at the 3-input majority function. The truth table is given in Table 2.2 on page 51. There are four 1 outputs in this function. So, our logical expression will have four product terms. The first product term we write is for row 4 with a 1 output. Since A has a value of 0, we use its complement in the product term while using B and C as they have 1 as their value in this row. Thus, the product term for this row is $\overline{A}\,B\,C$. The product term for row 6 is $A\,\overline{B}\,C$. Product terms for rows 7 and 8 are $A\,B\,\overline{C}$ and $A\,B\,C$, respectively. ORing these four product terms gives the logical expression as

$$\text{3-input majority function} = \overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C\,.$$

Example 2.2: From the truth table for the even-parity function given in Table 2.2 on page 51, we can derive the following sum-of-products expression:

$$\text{3-input even-parity function} = \overline{A}\;\overline{B}\;C + \overline{A}\;B\;\overline{C} + A\;\overline{B}\;\overline{C} + A\;B\;C\,.$$

Notation: A notation that provides compact representation of logical expressions uses the decimal values of the input combinations for which the output is 1. For example, the first term in the majority function is written as 3 (for the combination 011). To indicate that it is a sum-of-products expression, we use Σ as shown in the following expression:

$$\text{3-input majority function} = \Sigma\,(3, 5, 6, 7)\,.$$

Similarly, we can write the even-parity function using the Sigma notation as

$$\text{3-input even-parity function} = \Sigma\,(1, 2, 4, 7)\,.$$

2.6.2 Product-of-Sums Form

This is the dual form of the sum-of-products form. We essentially complement what we have done to obtain the sum-of-products expression. Here we look for rows that have a 0 output. Each such row input variable combination is expressed as an OR term. In this OR term, we use the variable if its value in the row being considered is 0 or its complement if 1. We AND these sum terms to get the final product-of-sums logical expression. The product-of-sums expression for the two truth tables is given below:

$$\text{Majority function} = (A + B + C)\,(A + B + \overline{C})\,(A + \overline{B} + C)\,(\overline{A} + B + C)\,,$$

$$\text{Even-parity function} = (A + B + C)\,(A + \overline{B} + \overline{C})\,(\overline{A} + B + \overline{C})\,(\overline{A} + \overline{B} + C)\,.$$

Notation: We can use a compact notation as we did with the sum-of-products expressions by listing only those sum terms for which the output is zero. We use Π to indicate that this is a product-of-sums expression. The majority function expression can be written as

$$\text{3-input majority function} = \Pi\,(0, 1, 2, 4)\,.$$

The even-parity function can be written using the Pi notation as

$$\text{3-input even-parity function} = \Pi\,(0, 3, 5, 6)\,.$$

2.6.3 Brute Force Method of Implementation

The sum-of-products and product-of-sums logical expressions can be used to come up with a crude implementation that uses only the AND, OR, and NOT gates. The implementation process is straightforward. We first illustrate the process for sum-of-products expressions. For each input, derive its complement using an inverter. Implement each product term by using a single n-input AND gate, where n is the number of Boolean variables. Then, connect the outputs of these AND gates to a single OR gate. The number of inputs to the OR gate is equal to the number of product terms in the logical expression. The output of the OR gate represents the logical function. Figure 2.13 shows the brute force implementation of the sum-of-products expression for the even-parity function.

In a similar fashion, we can also implement product-of-sums expressions. In this implementation, we use an OR gate to implement each sum term and a single AND gate to get the final output. Figure 2.14 shows an implementation of the product-of-sums expression for the even-parity function. Since these two forms of logical expressions are representing the same truth table, they are equivalent. As the two circuits given in Figures 2.13 and 2.14 implement these two logical expressions, we know that these two circuits are equivalent as well.

2.7 Simplifying Logical Expressions

Let us now focus on how we can simplify the logical expressions obtained from the truth table. Our focus is on sum-of-products expressions. There are three techniques: the algebraic manipulation, Karnaugh map, and Quine–McCluskey methods. Algebraic manipulation uses the Boolean laws given on page 55 to derive a simplified logical expression. The Karnaugh map method uses a graphical form and is suitable for simplifying logical expressions with a small number of variables. The last method is a tabular method and is particularly suitable for simplifying logical expressions with a large number of variables. In addition, the Quine–McCluskey method can be used to automate the simplification process.

2.7.1 Algebraic Manipulation

In this method, we use the Boolean laws (see page 55) discussed in Section 2.4.1. The process is very similar to that used to show logical equivalence of two functions. There is one big problem

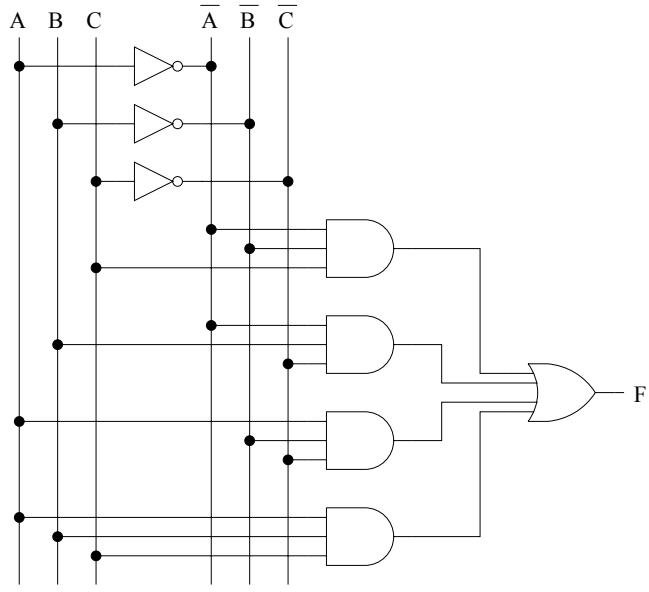


Figure 2.13 Brute force method of implementing the logical sum-of-products expression for the 3-input even-parity function.

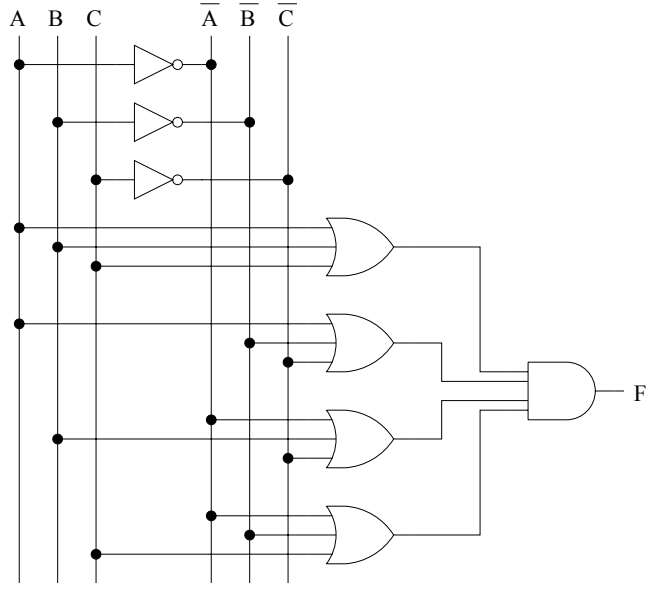


Figure 2.14 Brute force method of implementing the logical product-of-sums expression for the 3-input even-parity function.

though. Here we do not know what the target expression is. To illustrate this point, let us look at the sum-of-products expression for the majority function. A straightforward simplification leads us to the following expression:

$$\begin{aligned}\text{Majority function} &= \overline{A}BC + A\overline{B}C + \underbrace{AB\overline{C} + ABC}_{AB} \\ &= \overline{A}BC + A\overline{B}C + AB.\end{aligned}$$

Do you know if this is the final simplified form? This is the hard part in applying algebraic manipulation (in addition to the inherent problem of which rule should be applied). This method definitely requires good intuition, which often implies that one needs experience to know if the final form has been derived. In our example, the expression can be further simplified. We start by rewriting the original logical expression by repeating the term A B C twice and then simplifying the expression as shown below.

$$\begin{aligned}\text{Majority function} &= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC + \underbrace{ABC + ABC}_{\text{Added extra}} \\ &= \underbrace{\overline{A}BC + ABC}_{BC} + \underbrace{A\overline{B}C + ABC}_{AC} + \underbrace{AB\overline{C} + ABC}_{AB} \\ &= BC + AC + AB.\end{aligned}$$

This is the final simplified expression. In the next section, we show a simpler method to derive this expression. Figure 2.9 on page 51 shows an implementation of this logical expression.

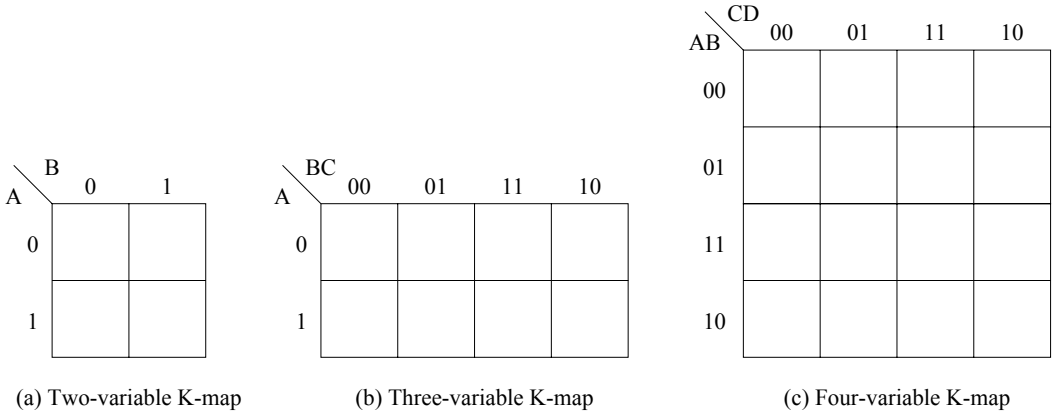
2.7.2 Karnaugh Map Method

This is a graphical method and is suitable for simplifying logical expressions with a small number of Boolean variables (typically six or less). It provides a straightforward method to derive minimal sum-of-products expressions. This method is preferred to the algebraic method as it takes the guesswork out of the simplification process. For example, in the previous majority function example, it was not straightforward to guess that we have to duplicate the term A B C twice in order to get the final logical expression.

The Karnaugh map method uses maps to represent the logical function output. Figure 2.15 shows the maps used for 2-, 3-, and 4-variable logical expressions. Each cell[1] in these maps represents a particular input combination. Each cell is filled with the output value of the function corresponding to the input combination represented by the cell. For example, the bottom left-hand cell represents the input combination A = 1 and B = 0 for the two-variable map (Figure 2.15*a*), A = 1, B = 0, and C = 0 for the three-variable map (Figure 2.15*b*), and A = 1, B = 0, C = 0, and D = 0 for the four-variable map (Figure 2.15*c*).

The basic idea behind this method is to label cells such that the neighboring cells differ in only one input bit position. This is the reason why the cells are labeled 00, 01, 11, 10 (notice

[1] The pigeonholes are usually referred to as squares. We prefer cells as we later talk about square areas.



(a) Two-variable K-map (b) Three-variable K-map (c) Four-variable K-map

Figure 2.15 Maps used for simplifying 2-, 3-, and 4-variable logical expressions using the Karnaugh map method.

the change in the order of the last two labels from the normal binary number order). What we are doing is labeling with a Hamming distance of 1. Hamming distance is the number of bit positions in which two binary numbers differ. This labeling is also called *gray code*. Why are we so interested in this gray code labeling? Simply because we can then eliminate a variable as the following holds:

$$A\,B\,\overline{C}\,D + A\,B\,C\,D = A\,B\,D.$$

Figure 2.16 shows how the maps are used to obtain minimal sum-of-products expressions for three-variable logical expressions. Notice that each cell is filled with the output value of the function corresponding to the input combination for that cell. After the map of a logical function is obtained, we can obtain a simplified logical expression by grouping neighboring cells with 1 into areas. Let us first concentrate on the majority function map shown in Figure 2.16a. The two cells in the third column are combined into one area. These two cells represent inputs $\overline{\text{A}}\,\text{B}\,\text{C}$ (top cell) and A B C (bottom cell). We can, therefore, combine these two cells to yield a product term B C. Similarly, we can combine the three 1s in the bottom row into two areas of two cells each. The corresponding product terms for these two areas are A C and A B as shown in Figure 2.16a. Now we can write the minimal expression as B C + A C + A B, which is what we got in the last section using the algebraic simplification process. Notice that the cell for A B C (third cell in the bottom row) participates in all three areas. This is fine. What this means is that we need to duplicate this term two times to simplify the expression. This is exactly what we did in our algebraic simplification procedure.

We now have the necessary intuition to develop the required rules for simplification. These simple rules govern the simplification process:

1. Form regular areas that contain 2^i cells, where $i \geq 0$. What we mean by a regular area is that they can be either rectangles or squares. For example, we cannot use an "L" shaped area.

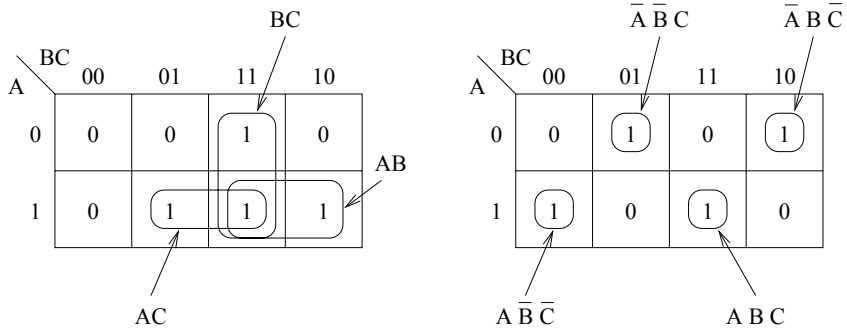


Figure 2.16 Three-variable logical expression simplification using Karnaugh maps: (a) majority function; (b) even-parity function.

2. Use a minimum number of areas to cover all cells with 1. This implies that we should form as large an area as possible and redundant areas should be eliminated. The importance of eliminating redundancy is illustrated later using an example (see Figure 2.19).

Once minimal areas have been formed, we write a logical expression for each area. These represent terms in the sum-of-products expressions. Write the final expression by connecting the terms with OR.

In Figure 2.16*a*, we cannot form a regular area with four cells. Next we have to see if we can form areas of two cells. The answer is yes. Let us assume that we first formed a vertical area (labeled B C). That leaves two 1s uncovered by an area. So, we form two more areas to cover these two 1s. We also make sure that we indeed need these three areas to cover all 1s. Our next step is to write the logical expression for these areas.

When writing an expression for an area, look at the values of a variable that is 0 as well as 1. For example, for the area identified by B C, the variable A has 0 and 1. That is, the two cells we are combining represent $\overline{A}\,B\,C$ and $A\,B\,C$. Thus, we can eliminate variable A. The variables B and C have the same value for the whole area. Since they both have the value 1, we write $B\,C$ as the expression for this area. It is straightforward to see that the other two areas are represented by $A\,C$ and $A\,B$.

If we look at the Karnaugh map for the even-parity function (Figure 2.16*b*), we find that we cannot form areas bigger than one cell. This tells us that no further simplification is possible for this function.

Notice that, in the three-variable maps, the first and last columns are adjacent. We did not need this fact in our previous two examples. You can visualize the Karnaugh map as a tube, cut open to draw in two dimensions. This fact is important because we can combine these two columns into a square area as shown in Figure 2.17. This square area is represented by $\overline{C}$.

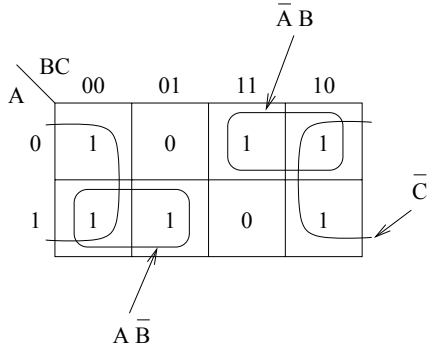


Figure 2.17 An example Karnaugh map that uses the fact that the first and last columns are adjacent.

You might have noticed that we can eliminate $\log_2 C$ variables from the product term, where C is the number of cells in the area. For example, the four-cell square in Figure 2.17 eliminates two variables from the product term that represents this area.

Figure 2.18 shows an example of a four-variable logical expression simplification using the Karnaugh map method. *It is important to remember the fact that first and last columns as well as first and last rows are adjacent.* Then it is not difficult to see why the four corner cells form a regular area and are represented by the expression $\overline{B}\,\overline{D}$. In writing an expression for an area, look at the input variables and ignore those that assume both 0 and 1. For example, for this weird square area, looking at the first and last rows, we notice that variable A has 0 for the first row and 1 for the last row. Thus, we eliminate A. Since B has a value of 0, we use $\overline{B}$. Similarly, by looking at the first and last columns, we eliminate C. We use $\overline{D}$ as D has a value of 0. Thus, the expression for this area is $\overline{B}\,\overline{D}$. Following our simplification procedure to cover all cells with 1, we get the following minimal expression for Figure 2.18*a*:

$$\overline{B}\,\overline{D} + A\,\overline{C}\,D + A\,B\,D.$$

We also note from Figure 2.18 that a different grouping leads to different minimal expression. The logical expression for Figure 2.18*b* is

$$\overline{B}\,\overline{D} + A\,\overline{B}\,\overline{C} + A\,B\,D.$$

Even though this expression is slightly different from the logical expression obtained from Figure 2.18*a*, both expressions are minimal and logically equivalent.

In general, we start making up areas from the largest possible to the smallest. This strategy sometimes leads to redundancy as illustrated in Figure 2.19*a*. In this map, we first formed the square area consisting of the middle four cells. Then we have added four rectangles, each with two cells. Although these five areas cover all 1 cells, we notice that, after forming the four rectangles, the square area is really redundant as shown in Figure 2.19*b*.

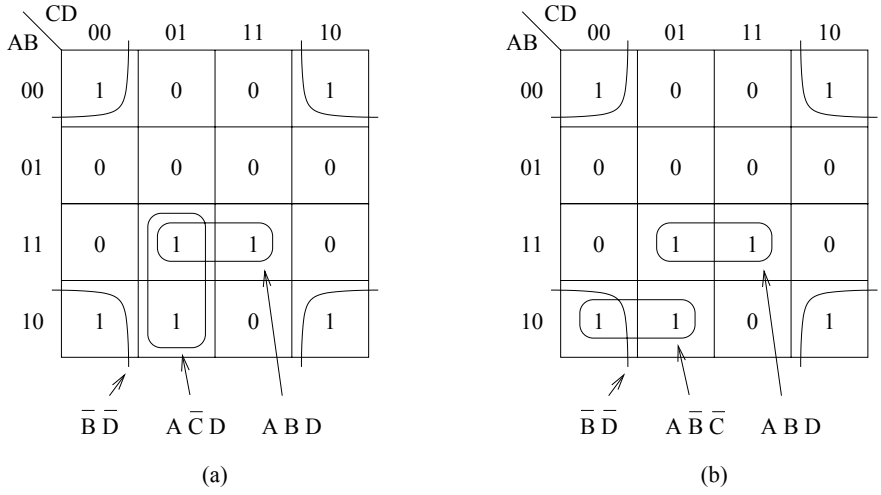


Figure 2.18 Different minimal expressions will result depending on the groupings.

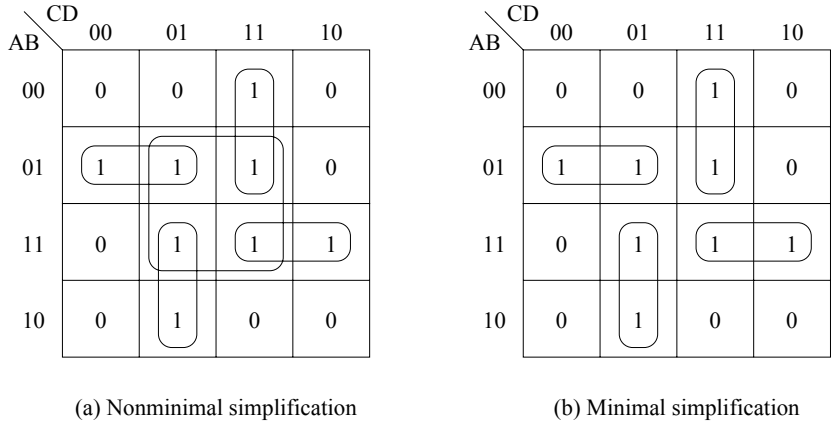


Figure 2.19 Example illustrating the need for redundancy check.

The best way to understand the Karnaugh map method is to practice until you develop your intuition. After that, it is unlikely you will ever forget how this method works even if you have not used it in years.

Seven-Segment Display Example

To show the utility of the Karnaugh map method, consider designing a logic circuit to drive a seven-segment display. This display unit that we are all familiar with (look at your VCR,

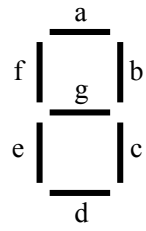


Figure 2.20 Seven-segment LED display.

calculator—they are everywhere) consists of seven segments of LEDs (light emitting diodes) as shown in Figure 2.20. Each diode emits light if current is passed through it. Depending on the digit we want to display, we selectively light only those segments that form the digit. For example, to display 7, we light segments `a`, `b`, and `c`.

Typically, a seven-segment decoder receives a BCD number and generates outputs for all seven segments. In this example, let us design a logic circuit that drives the LED `d`. The input to this circuit is a 4-bit BCD number. The truth table for this LED is shown in Table 2.5. In this truth table, a 1 for the segment indicates it is on; a 0 means it is off. We assume that the input is restricted to digits 0 through 9. Since the input values 10 through 15 are not given, the output for these six input combinations can be either a 0 or a 1. For obvious reasons, these outputs are called "don't cares." Such don't care outputs simplify the logic significantly as we show in a moment.

Figure 2.21a shows the Karnaugh map for this example. In this map, we are assuming that the output should be 0 for the last six inputs, that is, 10 through 15 (see the shaded area in Figure 2.21a). The simplified expression for this map is

$$\mathrm{A}\,\overline{\mathrm{B}}\,\overline{\mathrm{C}} + \overline{\mathrm{A}}\,\mathrm{C}\,\overline{\mathrm{D}} + \overline{\mathrm{A}}\,\overline{\mathrm{B}}\,\mathrm{C} + \overline{\mathrm{A}}\,\overline{\mathrm{B}}\,\overline{\mathrm{D}} + \overline{\mathrm{A}}\,\mathrm{B}\,\overline{\mathrm{C}}\,\mathrm{D}.$$

We could have elected to cover the top left cell with an area that includes this cell and the bottom left cell. In this case, we get

$$\mathrm{A}\,\overline{\mathrm{B}}\,\overline{\mathrm{C}} + \overline{\mathrm{A}}\,\mathrm{C}\,\overline{\mathrm{D}} + \overline{\mathrm{A}}\,\overline{\mathrm{B}}\,\mathrm{C} + \overline{\mathrm{B}}\,\overline{\mathrm{C}}\,\overline{\mathrm{D}} + \overline{\mathrm{A}}\,\mathrm{B}\,\overline{\mathrm{C}}\,\mathrm{D}.$$

This is slightly different from the other logical expression but is equivalent to the other one.

Don't Care Conditions

Since we don't care about the output for the shaded cells in Figure 2.21a, we can further simplify the last logical expression. We use "d" to represent the don't care output of a cell. The simplified expression for this map is

$$\mathrm{A} + \overline{\mathrm{B}}\,\overline{\mathrm{D}} + \mathrm{C}\,\overline{\mathrm{D}} + \overline{\mathrm{B}}\,\mathrm{C} + \mathrm{B}\,\overline{\mathrm{C}}\,\mathrm{D}.$$

The nice thing about the d cells is that they can be used to form an area without covering all such cells (as we would a 1 cell). That means, those d cells that are part of an area output a value

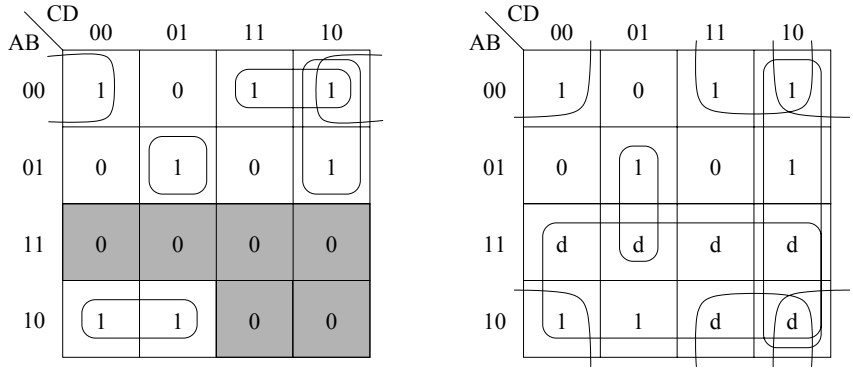
Table 2.5 Truth table for segment d

Number	A	B	C	D	Segment d
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0/1
11	1	0	1	1	0/1
12	1	1	0	0	0/1
13	1	1	0	1	0/1
14	1	1	1	0	0/1
15	1	1	1	1	0/1

of 1 and those that are not part of any area output 0. In our example, all d cells participate in at least one area. Thus, in this design, segment d is turned on for inputs 10 through 15, whereas it is turned off if we implement the logical expression obtained from Figure 2.21*a*.

A Seven-Segment Decoder/Driver Chip

In the last example, we have demonstrated how one can design a logic circuit to drive segment d. We could design six other driver circuits for the remaining segments to complete the driver circuit for a seven-segment display device. Because these display devices are ubiquitous, there are chips available that take a BCD number as input and generate output to drive all seven segments. One such chip is the 7449 chip (see Figure 2.22). This chip generates active-high segment driver outputs. It has four input bits for the BCD number. The only additional input



(a) Simplification with no don't cares

(b) Simplification with don't cares

Figure 2.21 Karnaugh maps for segment d of the seven-segment display.

is the $\overline{\text{BI}}$ signal. When $\overline{\text{BI}}$ is 1, the seven-segment outputs (a to f) are activated to drive the segments. The display assignments are shown in Figure 2.22a. When $\overline{\text{BI}}$ is 0, all seven segments are turned off (i.e., all seven outputs a to f are 0) irrespective of the BCD input. This input is useful in suppressing leading zeros (i.e., 00075 is displayed as 75 by blanking out the three leading displays).

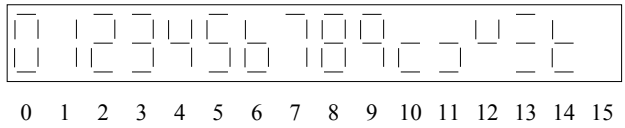
There is one difference between our logic for segment d and the output generated by the 7449 chip. We display 9 with the bottom horizontal LED on, whereas 7449 turns this LED off. Similarly, digit 6 can be displayed in two different forms. Look at your calculator and see the format it follows for digits 6 and 9.

2.7.3 Quine–McCluskey Method

The Karnaugh map method is not suitable for simplifying logical expressions with more than four variables. To simplify logical expressions with a higher number of variables, we have to use three-dimensional maps. We can push the Karnaugh map method to six variables but that's about it. The Quine–McCluskey method is a tabular method and is suitable for automating the simplification process. The Quine–McCluskey simplification process involves two steps:

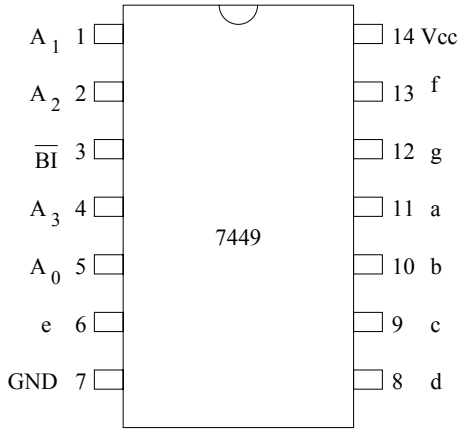
1. Obtain a simplified expression that is equivalent to the original expression. This expression need not be a minimal expression. This is done iteratively by looking at a pair of terms that contain a variable and its complement. This is equivalent to forming areas of size 2 in the Karnaugh map method. By iteratively applying this step, we form areas of larger size.

2. The second step essentially eliminates redundant terms from the simplified expression obtained in the last step. We needed this step even in the Karnaugh map method (e.g., see Figure 2.19).

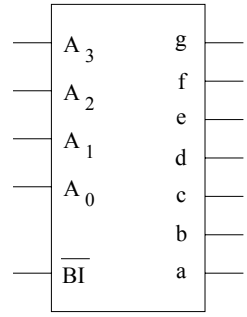


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

(a) Display designations



(b) Connection diagram



(c) Logic symbol

Figure 2.22 The 7449 seven-segment display driver chip.

We now illustrate the process by means of an example. Let us consider the segment d logical expression from the previous example. The logical expression can be written from the truth table in Table 2.5 as

$$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}C\,\overline{D} + \overline{A}\,\overline{B}CD + \overline{A}B\,\overline{C}D + \overline{A}BC\,\overline{D} + A\,\overline{B}\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{C}D.$$

We can express this logical expression more succinctly by using the notation described on page 57 as

$$\Sigma(0,\ 2,\ 3,\ 5,\ 6,\ 8,\ 9).$$

We start the Step 1 process by grouping the terms into the number of true conditions (i.e., number of 1s in the term) and sorting the groups as shown in column 1 of Table 2.6. We use a horizontal line to separate the groups. The first group at the top of the table is labeled group 0 as it has no 1 bits in the terms. We start simplifying the expression by finding all pairs of terms that differ in just one variable (i.e., form areas of size 2 cells). In effect, we are applying the rule $XY + X\overline{Y} = X$. Since the groups are sorted by the number of 1s in the terms, it is sufficient to compare the terms in two adjacent groups. That is, start with group 0 and compare each term in group 0 with all terms in group 1. If a pair is found, checkmark both terms and write the new

Table 2.6 Step 1: Finding prime implicants

	Column 1		Column 2
Group 0	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$	$\surd$	$\overline{A}\,\overline{B}\,\overline{D}$
Group 1	$\overline{A}\,\overline{B}\,C\,\overline{D}$	$\surd$	$\overline{B}\,\overline{C}\,\overline{D}$
	$A\,\overline{B}\,\overline{C}\,\overline{D}$	$\surd$	$\overline{A}\,\overline{B}\,C$
Group 2	$\overline{A}\,\overline{B}\,C\,D$	$\surd$	$\overline{A}\,C\,\overline{D}$
	$\overline{A}\,B\,\overline{C}\,D$		$A\,\overline{B}\,\overline{C}$
	$\overline{A}\,B\,C\,\overline{D}$	$\surd$	
	$A\,\overline{B}\,\overline{C}\,D$	$\surd$	

term in a new column (to be used in the next iteration). The new term is obtained by eliminating the variable that differs in the pair. Repeat the process by comparing each term in group 1 to all terms in group 2, and so on. Note that you will write a term into the next column only if it is not already present (i.e., no duplicate terms are allowed). This iteration produces the entries shown in column 2 of Table 2.6. This column represents the product terms for the areas with two cells. There is one term that is not checkmarked in column 1 of Table 2.6. This corresponds to the lone term we got from the Karnaugh map in Figure 2.21*a*.

We repeat the procedure on the column 2 entries. That is, we try to form areas of four cells. However, for this example, we do not generate any new terms. This means that no areas of size greater than two cells can be generated for this example. You can see that this condition is true from the Karnaugh map in Figure 2.21*a*.

To complete Step 1, we collect all the terms that are not checkmarked from the table. These terms are *prime implicants*. In our example, we have six prime implicants: one from column 1 with four variables and five from column 2, each with three variables.

Next we apply Step 2. This step eliminates any redundant terms from the set of prime implicants. To facilitate this objective, we create another table (called the prime implicant chart) with a row for each prime implicant and a column for each term in the original logical expression (see Table 2.7). Put a $\times$ mark in the table if the prime implicant for the row is in the column term. For example, the first column of the third row has an $\times$ mark as the row prime implicant $\overline{B}\ \overline{C}\ \overline{D}$ is in $\overline{A}\ \overline{B}\ \overline{C}\ \overline{D}$. What this step does is to mark those input terms that are represented by the prime implicant.

Next circle each $\times$ that is alone in a column. These prime implicants are called *essential prime implicants* and should appear in any final minimal form expression. Then place a square

Table 2.7 Step 2: Prime implicant chart for redundancy elimination

Prime implicants	Input product terms						
	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$	$\overline{A}\,\overline{B}\,C\,\overline{D}$	$\overline{A}\,\overline{B}\,C\,D$	$\overline{A}\,B\,\overline{C}\,D$	$\overline{A}\,B\,C\,\overline{D}$	$A\,\overline{B}\,\overline{C}\,\overline{D}$	$A\,\overline{B}\,\overline{C}\,D$
$\overline{A}\,B\,\overline{C}\,D$				$\otimes$			
$\overline{A}\,\overline{B}\,\overline{D}$	$\times$	$\times$					
$\overline{B}\,\overline{C}\,\overline{D}$	$\times$					$\times$	
$\overline{A}\,\overline{B}\,C$		$\boxed{\times}$	$\otimes$				
$\overline{A}\,C\,\overline{D}$		$\boxed{\times}$			$\otimes$		
$A\,\overline{B}\,\overline{C}$						$\boxed{\times}$	$\otimes$

around all the $\times$s in a row that has a $\otimes$. Thus, by using these prime implicants with a $\otimes$, we cover the input terms with $\boxed{\times}$ and $\otimes$. Thus, if we end up with a table in which each column has at least a $\otimes$ or a $\boxed{\times}$, we are done. This means that we get a single minimal form (i.e., there are no alternative minimal forms). We write the final minimal expression by using these prime implicants as sum-of-products terms.

If there are columns without a $\otimes$ or a $\boxed{\times}$, we select a minimum number of prime implicants to cover these columns. In our example, the first column is without a $\otimes$ or a $\boxed{\times}$. This means the term $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ is not covered by the essential prime implicants. We need either the second or the third prime implicant (both have a $\times$ in the $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ column). Thus, we get two final minimal expressions depending on whether the second or third prime implicant is selected. This should make sense to you from the Karnaugh map procedure. Thus, if we selected $\overline{A}\,\overline{B}\,\overline{D}$, our simplified expression for this example is

$$\overline{A}\,B\,\overline{C}\,D + \overline{A}\,\overline{B}\,\overline{D} + \overline{A}\,\overline{B}\,C + \overline{A}\,C\,\overline{D} + A\,\overline{B}\,\overline{C}.$$

We get the following expression if we selected $\overline{B}\,\overline{C}\,\overline{D}$:

$$\overline{A}\,B\,\overline{C}\,D + \overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,C + \overline{A}\,C\,\overline{D} + A\,\overline{B}\,\overline{C}.$$

These two expressions match the logical expressions we got with the Karnaugh map method.

Don't Care Conditions

How do we incorporate don't care conditions into the Quine–McCluskey method? Let us see how we handled the don't cares in the Karnaugh map method. We treated the don't cares as 1 when we needed to form areas, and yet we did not obligate ourselves to cover all the don't care cells. This precisely is what we will do in the Quine–McCluskey method as well. Since Step 1 is used to form areas iteratively, we include the don't care terms in this step as though they were regular terms for which the function outputs 1. We don't worry about the fact that such inclusion of don't care terms would generate some redundant terms (e.g., consisting of only don't care terms). We will depend on the next step to eliminate any such redundancies. In Step 2, since we are not obligated to cover the don't care terms, we do not list them. In other words, there won't be any columns created in the prime implicant chart for the don't care terms. In summary, we include the don't care terms in the first step and apply the Step 1 procedure and ignore them in Step 2 and apply the Step 2 procedure discussed before.

We illustrate the process by considering the seven-segment example used in the Karnaugh map method. Using the Sigma notation we described on page 57, the logical expression for the seven-segment example can be represented as

$$\Sigma(0,\ 2,\ 3,\ 5,\ 6,\ 8,\ 9)\ +\ \Sigma\,\mathrm{d}(10,\ 11,\ 12,\ 13,\ 14,\ 15)\,,$$

where we use $\Sigma\,\mathrm{d}$ to represent the don't care inputs.

By including all the don't care terms, we get the entries in Column 1 of Table 2.8. By following the procedure described before, we get the following terms that are not checked off: one term from column 2, three terms from column 3, and a single-variable term from the last column. Notice that this example generates several duplicates, all of which are eliminated.

Next construct the prime implicants chart, shown in Table 2.9. Here we do not include the don't care terms. From this table, we can see that all five terms are essential prime implicants. Thus, we end up with just one final minimal expression

$$\mathrm{A}\ +\ \overline{\mathrm{B}}\,\overline{\mathrm{D}}\ +\ \mathrm{C}\,\overline{\mathrm{D}}\ +\ \overline{\mathrm{B}}\,\mathrm{C}\ +\ \mathrm{B}\,\overline{\mathrm{C}}\,\mathrm{D}\,.$$

This matches the logical expression obtained with the Karnaugh map method (see page 65).

2.8 Generalized Gates

Even though we use multiple input gates as needed by our design, such gates may not be commercially available to build digital circuits. Even when available, there may be reasons not to use them. For example, we may have two of the four 2-input AND gates free in a 7408 chip. In that case, if we need a 3-input AND gate, we would like to use these free gates rather than adding a new chip. It is fairly easy to build higher-input gates using lower-input gates of the same kind for AND and OR gates. An example of building a 3-input AND gate using two 2-input AND gates is shown in Figure 2.23*a*. This process can be generalized to build arbitrarily large input gates of this kind. You are asked in Exercise 2–23 to show that the same construction procedure can be used even for the XOR gate.

Table 2.8 Step 1: Finding prime implicants

Column 1	Column 2	Column 3	Column 4
$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$ √	$\overline{A}\,\overline{B}\,\overline{D}$ √	$\overline{B}\,\overline{D}$	A
$\overline{A}\,\overline{B}\,C\,\overline{D}$ √	$\overline{B}\,\overline{C}\,\overline{D}$ √	$\overline{B}\,C$	
$A\,\overline{B}\,\overline{C}\,\overline{D}$ √	$\overline{A}\,\overline{B}\,C$ √	$C\,\overline{D}$	
$\overline{A}\,\overline{B}\,C\,D$ √	$\overline{A}\,C\,\overline{D}$ √	$A\,\overline{B}$ √	
$\overline{A}\,B\,\overline{C}\,D$ √	$\overline{B}\,C\,\overline{D}$ √	$A\,\overline{C}$ √	
$\overline{A}\,B\,C\,\overline{D}$ √	$A\,\overline{B}\,\overline{C}$ √	$A\,\overline{D}$ √	
$A\,\overline{B}\,\overline{C}\,D$ √	$A\,\overline{B}\,\overline{D}$ √	$A\,D$ √	
$A\,\overline{B}\,C\,\overline{D}$ √	$A\,\overline{C}\,\overline{D}$ √	$A\,C$ √	
$A\,B\,\overline{C}\,\overline{D}$ √	$\overline{B}\,C\,D$ √	$A\,B$ √	
$A\,\overline{B}\,C\,D$ √	$B\,\overline{C}\,D$		
$A\,B\,\overline{C}\,D$ √	$B\,C\,\overline{D}$ √		
$A\,B\,C\,\overline{D}$ √	$A\,\overline{B}\,D$ √		
$A\,B\,C\,D$ √	$A\,\overline{C}\,D$ √		
	$A\,\overline{B}\,C$ √		
	$A\,C\,\overline{D}$ √		
	$A\,B\,\overline{C}$ √		
	$A\,B\,\overline{D}$ √		
	$A\,C\,D$ √		
	$A\,B\,D$ √		
	$A\,B\,C$ √		

It is not as straightforward to build higher-input NAND or NOR gates using lower-input gates of the same kind. As an example, we show in Figure 2.23*b* how a 3-input NAND gate can be built using 2-input NAND gates. Note that it requires an additional inverter. A similar procedure can be used for the NOR gate. The key point is that we have to invert the output of a NAND gate before feeding it as input to the next NAND gate.

Since it is straightforward to build higher-input gates, we use them liberally in our logic circuits knowing that such circuits can be easily implemented in practice. However, we should be careful in designing such circuits as the propagation delay (discussed on page 49) associated

Table 2.9 Step 2: Prime implicant chart for redundancy elimination

Prime implicants	Input product terms: No don't care terms						
	$\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$	$\overline{A}\,\overline{B}\,C\,\overline{D}$	$\overline{A}\,\overline{B}\,C\,D$	$\overline{A}\,B\,\overline{C}\,D$	$\overline{A}\,B\,C\,\overline{D}$	$A\,\overline{B}\,\overline{C}\,\overline{D}$	$A\,\overline{B}\,\overline{C}\,D$
$B\,\overline{C}\,D$				⊗			
$\overline{B}\,\overline{D}$	⊗	×				×	
$C\,\overline{D}$		×			⊗		
$\overline{B}\,C$		×	⊗				
A						×	⊗

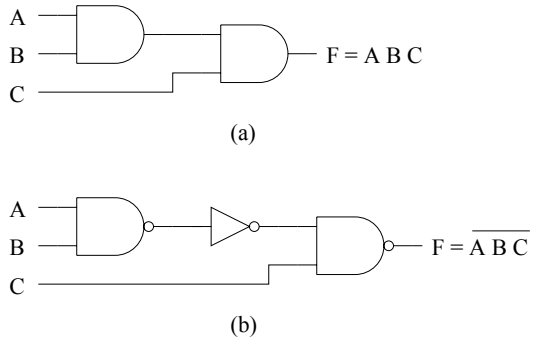


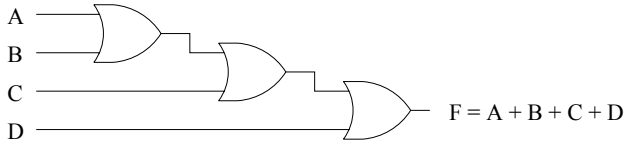
Figure 2.23 Constructing 3-input gates using 2-input gates.

with all equivalent circuits may not be the same. As an example, consider building a 4-input OR gate using 2-input OR gates. We can build a 4-input OR gate by cascading a series of three 2-input OR gates as shown in Figure 2.24a. The propagation delay of this circuit is three gate delays. On the other hand, the series-parallel approach used to derive the circuit shown in Figure 2.24b incurs only a two-gate propagation delay.

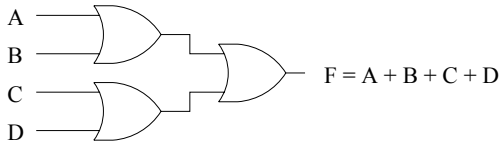
2.9 Multiple Outputs

So far we have considered logical functions with only a single output function. What if we have to design a circuit that has to generate more than one output? For example, how do we implement the truth table shown in Table 2.10? We can use the previous procedure by treating the truth table in Table 2.10 as two truth tables.

We can write simplified logical expressions for these two functions as



(a) Series implementation



(b) Series-parallel implemetation

Figure 2.24 Two logically equivalent 4-input OR gates built with three 2-input OR gates: (a) series implementation involves three gate delays; (b) series-parallel implementation involves only two gate delays.

Table 2.10 Truth table with two output functions

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$F_1 = \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C\,,$$

$$F_2 = A\,B + B\,C + A\,C\,.$$

Even though we have not stated in words what these functions are supposed to be doing, from our discussion so far we can readily identify that F_1 is the even-parity function and F_2

is the majority function. Interestingly, we can also assign another interpretation for these two functions. This is also the truth table for the full adder with F_1 representing the sum output and F_2 representing the $carry$ output C_{out}. The three inputs represent two single-bit inputs and a carry in C_{in}. We discuss adders in Chapter 3.

2.10 Implementation Using Other Gates

The synthesis process we have discussed so far uses the three basic gates—AND, OR, and NOT—for implementation. In this section, we show how implementations using other gates (such as NAND and NOR) can be obtained.

2.10.1 Implementation Using NAND and NOR Gates

It is sometimes useful to design logic circuits using only NAND gates. For example, implementing $(A + \overline{B})$ requires one OR gate (one 7432 chip) and one inverter (one 7406 chip). Noting that this expression is equivalent to $(\overline{\overline{A}\,B})$, we can implement the logical expression using two NAND gates; thus, only one 7400 chip is needed. As we have noted, NAND gates are universal as any logic function can be implemented using only NAND gates. Similarly, we can also use only NOR gates.

Let us see how we can derive a design that uses only NAND gates. As an example, consider the expression $(A\,B + C\,D)$. Implementing this expression requires two 2-input AND gates and a 2-input OR gate. Since $\overline{\overline{X}} = X$, we can double negate the expression.

$$A\,B + C\,D = \overline{\overline{A\,B + C\,D}}.$$

Now apply de Morgan's law to move the inner negation operation to yield

$$A\,B + C\,D = \overline{\overline{A\,B} \cdot \overline{C\,D}}.$$

Notice that the right-hand expression can be implemented using only NAND gates. Such an implementation requires three 2-input NAND gates.

How do we apply this technique to a logical function that has more than two product terms? Let us consider the simplified logical expression for the majority function. This function can be written as

$$\begin{aligned} A\,B + B\,C + A\,C &= \overline{\overline{A\,B + B\,C + A\,C}} \\ &= \overline{\overline{A\,B + B\,C} \cdot \overline{A\,C}} \\ &= \overline{\overline{A\,B} \cdot \overline{B\,C} \cdot \overline{A\,C}}. \end{aligned}$$

We need three 2-input NAND gates and a 3-input NAND gate to implement this function (see Figure 2.25).

We derive the following for the 3-input even-parity function:

$$\overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C = \overline{\overline{\overline{A}\,\overline{B}\,C} \cdot \overline{\overline{A}\,B\,\overline{C}} \cdot \overline{A\,\overline{B}\,\overline{C}} \cdot \overline{A\,B\,C}}.$$

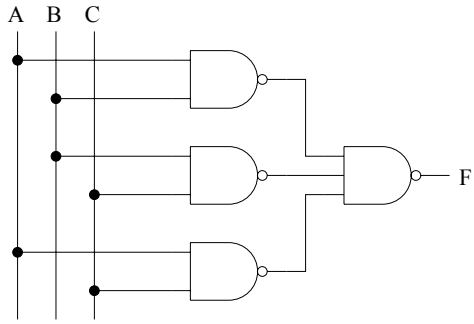


Figure 2.25 A majority function implementation using only NAND gates.

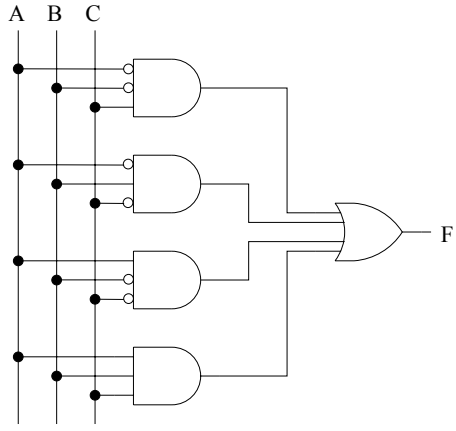


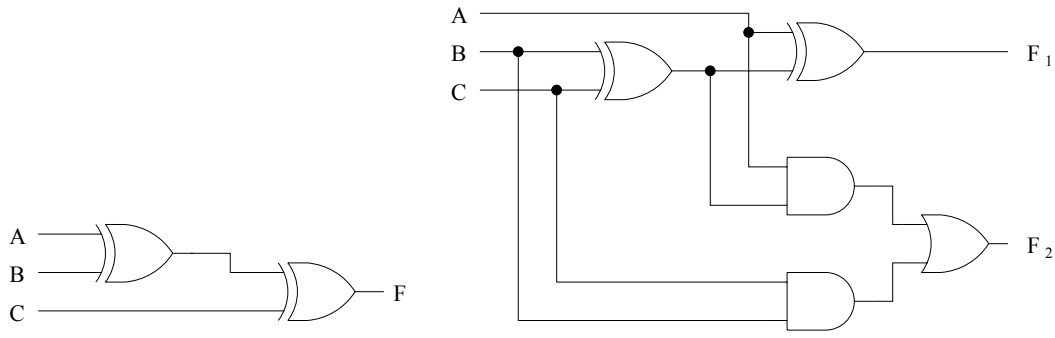
Figure 2.26 Logic circuit for the 3-input even-parity function using the bubble notation.

This requires three 2-input NAND gates for implementing the inverters (to get $\overline{A}$, $\overline{B}$, and $\overline{C}$), four 3-input NAND gates for the inner terms, and a 4-input NAND for the outer negation.

We can apply a similar technique for product-of-sums expressions to come up with NOR-only circuit designs.

Bubble Notation

In large circuits, drawing inverters can be avoided by following what is known as the "bubble" notation. Remember that we have been using the bubble to represent negation. Using the bubble notation simplifies the circuit diagrams. To appreciate the reduced complexity, compare the bubble notation circuit for the 3-input even-parity function in Figure 2.26 with that in Figure 2.13.



(a) Even-parity function (b) Two-output function

Figure 2.27 Logic circuit implementations using the XOR gate.

2.10.2 Implementation Using XOR Gates

Exclusive-OR gates are very useful in implementing certain types of functions. Notice that the XOR gate implements the logical expression of the form $\overline{A}\,B + A\,\overline{B}$. You can do pattern recognition of sorts to search for this type of expression and implement it using the XOR gate.

Let us look at a couple of examples. As a first example, consider the 3-input even-parity function. To use XOR gates, we have to transform the logical expression as follows:

$$\begin{aligned}
\overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C &= C\,(\overline{A}\,\overline{B} + A\,B) + \overline{C}\,(\overline{A}\,B + A\,\overline{B}) \\
&= C\,(\overline{\overline{\overline{A}\,\overline{B} + A\,B}}) + \overline{C}\,(\overline{A}\,B + A\,\overline{B}) \\
&= C\,(\overline{\overline{A}\,B + A\,\overline{B}}) + \overline{C}\,(\overline{A}\,B + A\,\overline{B}).
\end{aligned}$$

There is a big jump from the second expression to the final one. You can verify that

$$\overline{\overline{\overline{A}\,\overline{B} + A\,B}} = \overline{\overline{A}\,B + A\,\overline{B}}.$$

We can see from this expression that we need just two 2-input XOR gates to implement the even-parity function as shown in Figure 2.27*a*. We can implement this logic function by using only half of the 7486 chip. Compare this circuit with the one in Figure 2.26 or in Figure 2.13.

You will often find the trick we have used here—that is, double negating and removing the inner negation by applying de Morgan's law—very useful in simplifying or manipulating logical expressions into the desired form.

As another example, consider the two output functions in Table 2.10 on page 74. We can transform the logical expression for F_1 so that we can implement it using two 2-input XOR gates (Figure 2.27*a*). The second function

$$F_2 = B\,C + A\,B + A\,C$$

can be implemented using two 2-input OR gates and two 2-input AND gates by writing it as

$$F_2 = B\,C + A\,(B + C)\,.$$

We can, however, reduce the gate count by noting that XOR of A and B is available from the implementation of F_1. The required transformation to use this term is done as follows:

$$\begin{aligned} F_2 &= B\,C + A\,B + A\,C \\ &= B\,C + A\,B\,(C + \overline{C}) + A\,C\,(B + \overline{B}) \\ &= B\,C + A\,B\,C + A\,B\,\overline{C} + A\,B\,C + A\,\overline{B}\,C \\ &= B\,C + A\,(B\,\overline{C} + \overline{B}\,C)\,. \end{aligned}$$

Implementation of F_1 and F_2 are shown in Figure 2.27*b*. As we show in Chapter 3, this is the full adder circuit.

2.11 Summary

We have introduced several simple logic gates such as AND, OR, NOT gates as well as NAND, NOR, and XOR gates. Although the first three gates are considered the basic gates, we often find that the other three gates are useful in practice.

We have described three ways of representing the logical functions: truth table, logical expression, and graphical form. The truth table method is cumbersome for logical expressions with more than a few variables. The number of rows in the truth table is equal to 2^N, where N is the number of logical variables. Logical expression representation is useful to derive simplified expressions by applying Boolean identities. The graphical form is useful to implement logical circuits.

Logical expressions can be written in one of two basic forms: sum-of-products or product-of-sums. From either of these expressions, it is straightforward to obtain logic circuit implementations. However, such circuits are not the best designs as simplifying logical expressions can minimize the component count.

Several methods are available to simplify logical expressions. We have discussed three of them: the algebraic, Karnaugh map, and Quine–McCluskey methods.

Our focus has been on devising methodologies for implementing logical circuits using the basic AND, OR, and NOT gates. However, in the last couple of sections, we have shown how logic designs can be obtained so that other gates such as NAND and XOR can be used in the implementation.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- AND gate
- Boolean algebra
- Bubble notation
- Complete set
- de Morgan's law
- Don't cares
- Even parity function
- Fanin, Fanout
- Generalized gates
- Integrated circuits
- Karnaugh maps
- Logic circuit design process
- Logic circuit equivalence
- Logical expression derivation
- Logical expression equivalence
- Logical expression simplification
- Logical expressions
- Majority function
- Multiple outputs
- NAND gate
- NMOS, PMOS, HMOS, CMOS, GaAs
- NOR gate
- NOT gate
- OR gate
- Product-of-sums
- Propagation delay
- Quine–McCluskey method
- Seven-segment display
- SSI, MSI, LSI, VLSI
- Sum-of-products
- Transistor implementation of gates
- Truth table
- TTL, ECL
- Universal gates
- XOR gate

2.12 Web Resources

You can use one of the following Web sites for information on IC chips. In particular, you get all the data sheets for the TTL family of chips from these two sites:

Motorola URL: `http://www.mot.com`
Texas Instruments URL: `http://www.ti.com`.

2.13 Exercises

2–1 Implement the 2-input XOR gate using (a) only 2-input NAND gates and (b) only 2-input NOR gates.

2–2 Implement the 2-input exclusive-NOR gate using (a) only 2-input NAND gates and (b) only 2-input NOR gates.

2–3 Show how a NOT gate can be implemented using a 2-input XOR gate.

2–4 In the last exercise, you have shown how an XOR gate can act as an inverter. In this exercise, show that a 2-input XOR gate can act as a buffer that simply passes input to the output. Now explain why the XOR gate is called a *programmable inverter*.

2–5 Show how an AND gate can be implemented using OR and NOT gates.

2–6 Show how an OR gate can be implemented using AND and NOT gates.

2–7 Describe how the circuit in Figure 2.6*b* is implementing a NAND gate.

2–8 Describe how the circuit in Figure 2.6*c* is implementing a NOR gate.

2–9 In our discussion of the 3-input majority function, we have assigned equal weight (i.e., 1/3) to the three inputs. Suppose that one input has a weight of 1/2 and the other two inputs have 1/4 each. Show the truth table for the weighted 3-input majority function. Derive a simplified logical expression and show the corresponding implementation.

2–10 Another variation on the majority function assigns veto power to certain members. Independent of how the others have voted, a member with veto power can defeat the motion. Show the truth table for this 3-input majority function with only one member having veto power. Derive a simplified logical expression and show the corresponding implementation.

2–11 Prove the following using only the first four laws in Table 2.4:

(a) $x \cdot x = x$.

(b) $x + x = x$.

(c) $x \cdot 0 = 0$.

(d) $x + 1 = 1$.

2–12 Prove the **and** version of de Morgan's law given in Table 2.4. *Hint:* It is useful to consider the observation made about the complement law on page 54. Thus, to prove

$$\overline{x \cdot y} = \overline{x} + \overline{y}$$

it is sufficient to show that

$$(x \cdot y) \cdot (\overline{x} + \overline{y}) = 0$$

and

$$(x \cdot y) + (\overline{x} + \overline{y}) = 1$$

are true.

2–13 Prove the **or** version of de Morgan's law given in Table 2.4.

2–14 Write the **and** and **or** versions of de Morgan's law for three variables. Verify your answer using the truth table method.

2–15 Find how many 7400 chips are required to implement the 8-input NAND gate provided by 7430. See Figure 2.8 on page 50.

2–16 Prove the following using the Boolean algebra method:

(a) $(\overline{x + y}) \cdot (\overline{\bar{x} + \bar{y}}) = 0$.

(b) $x + y\bar{x} = x + y$.

(c) $\overline{\overline{\mathrm{A}}\,\overline{\mathrm{B}} + \mathrm{A}\,\mathrm{B}} = \overline{\overline{\mathrm{A}}\,\mathrm{B} + \mathrm{A}\,\overline{\mathrm{B}}}$.

2–17 Give the truth table for the 3-input equivalence gate. Derive logical expressions in sum-of-products and product-of-sum forms.

2–18 Using Boolean algebra show that the two logical expressions derived in the last exercise are equivalent.

2–19 Show that the two logic circuits in Figures 2.10*a* and 2.10*b* are equivalent.

2–20 Using Boolean algebra show that the following two expressions are equivalent:

$$\overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C\ ,$$
$$(A+B+C)\,(A+B+\overline{C})\,(A+\overline{B}+C)\,(\overline{A}+B+C)\,.$$

These two expressions represent the majority function in sum-of-products and product-of-sums form.

2–21 Using Boolean algebra show that the following two expressions are equivalent:

$$\overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,B\,C\ ,$$
$$(A+B+C)\,(A+\overline{B}+\overline{C})\,(\overline{A}+B+\overline{C})\,(\overline{A}+\overline{B}+C)\,.$$

These two expressions represent the even-parity function in sum-of-products and product-of-sums form.

2–22 Using Boolean algebra show that the following two expressions are equivalent:

$$A\,\overline{B}\,\overline{C} + \overline{A}\,C\,\overline{D} + \overline{A}\,\overline{B}\,C + \overline{A}\,\overline{B}\,\overline{D} + \overline{A}\,B\,\overline{C}\,D\,,$$
$$A + \overline{B}\,\overline{D} + C\,\overline{D} + \overline{B}\,C + B\,\overline{C}\,D\,.$$

2–23 Show how a 5-input XOR gate can be constructed using only 2-input XOR gates.

2–24 We want to build a logic circuit to generate the even-parity bit for 7-bit ASCII characters. In transmitting an ASCII character, we transmit 7 ASCII bits and a parity bit to facilitate rudimentary error detection. Design such a circuit using 2-input XOR gates. What modification would we make to this circuit if we wanted to generate odd parity?

2–25 Using Boolean algebra show that the two logical expressions obtained from Figures 2.18*a* and 2.18*b* are equivalent. That is, show that the following two logical expressions are equivalent:

$$\overline{B}\,\overline{D} + A\,\overline{C}\,D + A\,B\,D\,,$$
$$\overline{B}\,\overline{D} + A\,\overline{B}\,\overline{C} + A\,B\,D\,.$$

2–26 Show the truth table of a function that outputs a 1 whenever the 3-bit input, treated as representing a 3-bit unsigned binary number, is even. Derive a logical expression and simplify it using Boolean algebra to show that a single inverter can implement this function.

2–27 Show the truth table of a function that outputs a 1 whenever the 4-bit input, treated as representing a 4-bit unsigned binary number, is divisible by 4. Derive a simplified logical expression using the Karnaugh map method. Show an implementation of this function.

2–28 Redo the last exercise using the Quine–McCluskey method.

2–29 Show the truth table of a function that outputs a 1 whenever the 4-bit input, treated as representing a 4-bit unsigned binary number, is between 5 and 10 (both inclusive). Derive a simplified logical expression using the Karnaugh map method. Show an implementation of this function.

2–30 Redo the last exercise using the Quine–McCluskey method.

2–31 Show the truth table of a function that outputs a 1 whenever the 4-bit input, treated as representing a 4-bit signed binary number, is equal to ± 2, ± 4, or ± 5. Derive a simplified logical expression using the Karnaugh map method. Show an implementation of this function.

2–32 Redo the last exercise using the Quine–McCluskey method.

Chapter 3

Combinational Circuits

Objectives

- To describe higher-level building blocks that are useful in designing digital logic circuits;
- To introduce programmable logic devices to implement logic functions;
- To discuss principles involved in the design of arithmetic and logic units;
- To provide a sample of commercial combinational digital circuit ICs.

In the last chapter, we discussed the fundamentals of digital circuit design. Our design process focused on the basic gates. This chapter focuses on combinational circuits, which provide a higher level of abstraction that is useful in designing digital circuits and systems.

We describe several examples of combinational circuits that are commonly required in the design of digital circuits. The combinational circuits we present in this chapter include multiplexers and demultiplexers, decoders and encoders, comparators, and adders. We show how multiplexers can be used as universal building blocks to implement logical functions. Similarly, decoders along with OR gates can also be used to implement any logical function.

In addition, we also discuss programmable logic devices that are useful for implementing logic functions in a straightforward way. We present details on two programmable logic devices: the programmable logic array and programmable array logic. These programmable logic devices are useful for implementing logical functions with a minimum number of chips. Arithmetic and logic units (ALUs) are also discussed to illustrate how design of complex digital circuits can be simplified by using combinational circuits discussed in this chapter.

3.1 Introduction

We have so far focused on implementations using only the basic gates. One key characteristic of the circuits that we have designed in the last chapter is that the output of the circuit is a function

of the inputs. Such devices are called *combinational circuits* as the output can be expressed as a combination of the inputs. We continue our discussion of combinational circuits in this chapter.

Although gate-level abstraction is better than working at the transistor level, a higher level of abstraction is needed in designing and building complex digital systems. We now discuss some combinational circuits that provide this higher level of abstraction.

Higher-level abstraction helps the digital circuit design and implementation process in several ways. The most important ones are the following:

1. Higher-level abstraction helps us in the logical design process as we can use functional building blocks that typically require several gates to implement. This, therefore, reduces the complexity.
2. The other equally important point is that the use of these higher-level functional devices reduces the chip count to implement a complex logical function.

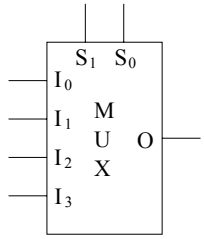
The second point is important from the practical viewpoint. If you look at a typical motherboard, these low-level gates take a lot of area on the printed circuit board (PCB). Even though the low-level gate chips such as the ones shown in Figure 2.8 on page 50 were introduced in the 1970s, you still find them sprinkled on your PCB along with your Pentium processor. In fact, they seem to take more space. Thus, reducing the chip count is important to make your circuit compact. The combinational circuits provide one mechanism to incorporate a higher level of integration. To drive the point home, assume that you want an 8-input NAND gate. We could use a single 14-pin DIP chip 7430 to do the job (see Figure 2.8 on page 50). How many 14-pin chips do we need to build the same using the 2-input NAND gate chip 7400?

The reduced chip count also helps in reducing the production cost (fewer ICs to insert and solder) and improving the reliability. Several combinational circuits are available for implementation. Here we look at a sampler of these circuits.

3.2 Multiplexers and Demultiplexers

A multiplexer (MUX) is characterized by 2^n data inputs, n selection inputs, and a single output. The block diagram representation of a 4-input multiplexer (4-to-1 multiplexer) is shown in Figure 3.1. The multiplexer connects one of 2^n inputs, selected by the selection inputs, to the output. Treating the selection input as a binary number, data input I_i is connected to the output when the selection input is i as shown in Figure 3.1.

Figure 3.2 shows an implementation of a 4-to-1 multiplexer. If you look closely, it somewhat resembles our logic circuit used by the brute force method for implementing sum-of-products expressions (compare this figure with Figure 2.13 on page 59). This visual observation is useful in developing our intuition about one important property of the multiplexers: they are universal in the sense that we can implement any logical function using only multiplexers. So, we can add one more entity to the complete set list on page 45. The best thing about using multiplexers in implementing a logical function is that you don't have to simplify the logical expression. We can proceed directly from the truth table to implementation, using the multiplexer as the building block.



S_1	S_0	O
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Figure 3.1 A 4-data input multiplexer block diagram and truth table.

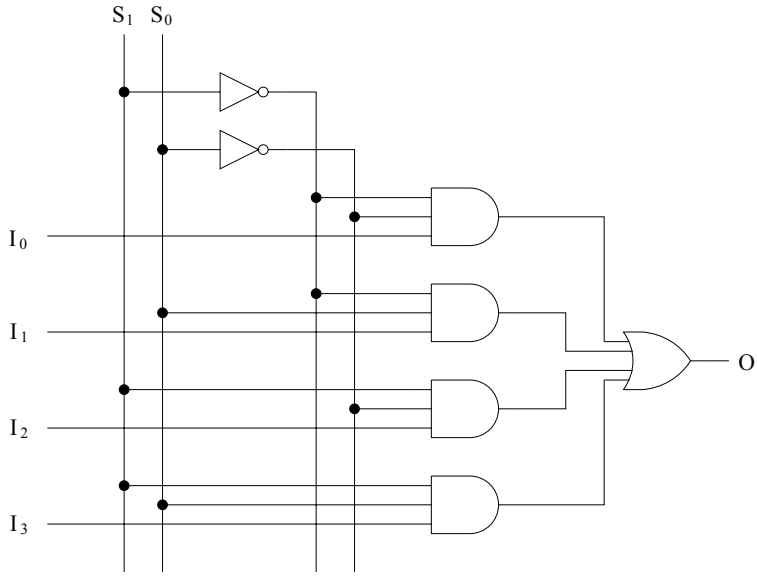


Figure 3.2 A 4-to-1 multiplexer implementation using the basic gates.

How do we implement a truth table using the multiplexer? Simple. Connect the logical variables in the logical expression as selection inputs and the function outputs as constants to the data inputs. To follow this straightforward implementation, we need a 2^b data input multiplexer with b selection inputs to implement a b variable logical expression. The process is best illustrated by means of an example.

Figure 3.3 shows how an 8-to-1 multiplexer can be used to implement our two running examples: the 3-input majority and 3-input even-parity functions. From these examples, you can see that the data input is simply a copy of the output column in the corresponding truth table. You just need to take care how you connect the logical variables: connect the most significant

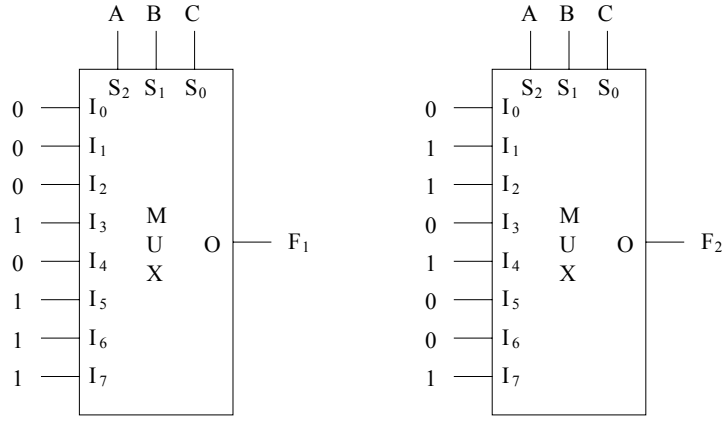


Figure 3.3 Two example implementations using an 8-to-1 multiplexer.

variable in the truth table to the most significant selection input of the multiplexer as shown in Figure 3.3.

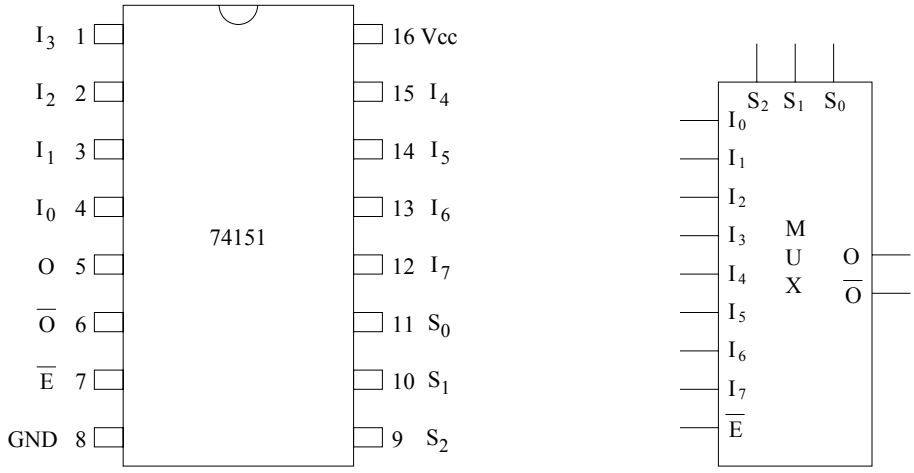
3.2.1 Implementation: A Multiplexer Chip

The 74151 is an example 8-to-1 multiplexer chip that is similar to the multiplexer we have used to implement the majority and even-parity functions. The connection diagram and the logic symbol are shown in Figure 3.4. The only additional input is the enable input ($\overline{\text{E}}$). This active-low input signal, after an internal inversion, goes as an additional input to all the AND gates in the multiplexer implementation shown in Figure 3.2. Thus, when this input is 1, output is forced to be high. For normal multiplexer operation, the enable input must be 0. Notice that the 74151 provides both the normal output (O) and its complement ($\overline{\text{O}}$). It is straightforward to see that we can implement the majority and even-parity functions using a single chip for each function.

A Note on the Notation: As we have just done, we often talk about low-active and high-active inputs. A low-active input means that a 0 should be applied to the input in order to activate the function. Similarly, a high-active means the input should be 1 to enable the function. We indicate a low-active input by using an overbar as in $\overline{\text{E}}$. There are several examples in this and later chapters.

3.2.2 Efficient Multiplexer Designs

We can do better than the naive design described in the last section. We can actually implement a b variable logical expression using a 2^{b-1} data input multiplexer. For some functions, we might need an additional inverter. The basic idea is to factor out one logical variable (say, X)



(a) Connection diagram (b) Logic symbol

Figure 3.4 The 74151 8-to-1 multiplexer chip.

from the truth table. This variable X or its complement may be required as a data input to the multiplexer (thus, the need for an additional inverter to get $\overline{X}$). In this design, the multiplexer data input set consists of $\{0, 1, X, \overline{X}\}$. Although any variable in the logical expression can be eliminated, it is most convenient to factor out the rightmost (as it appears in the truth table) logical variable.

The reduction process for the majority function is shown in Figure 3.5. On the left is the original truth table with three variables. We eliminate variable C from this table to get the new truth table with variables A and B. To derive the new truth table, we group two rows in which the values of A and B match. Then look at the output of these two rows: if both row outputs are zero (one), the output in the new table is zero (one); otherwise, find the relation between the output values of these two rows in the original table and the values of the variable C (you will get this to be either C or $\overline{C}$). For example, when A = 0 and B = 0, the output is zero independent of the value of C. So the new truth table has zero as output. On the other hand, when A = 0 and B = 1, the output is equal to the value of C. Thus, the output in the new table is C. Once we have derived this reduced truth table, implementation is straightforward as shown in Figure 3.5.

The corresponding truth table reduction process for the even-parity function is shown in Figure 3.6. The implementation of this function requires an inverter to get $\overline{C}$ (this inverter is not shown in the figure).

3.2.3 Implementation: A 4-to-1 Multiplexer Chip

The 74153 is a dual 4-to-1 multiplexer chip, shown in Figure 3.7, that can be used to implement our reduced truth tables. Even though the 74153 provides two 4-to-1 MUXs, the two are not

Original truth table

A	B	C	F_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

New truth table

A	B	F_1
0	0	0
0	1	C
1	0	C
1	1	1

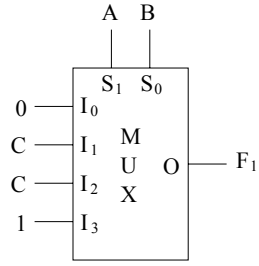


Figure 3.5 Derivation of the reduced truth table and its implementation for the majority function.

Original truth table

A	B	C	F_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

New truth table

A	B	F_1
0	0	C
0	1	$\overline{C}$
1	0	$\overline{C}$
1	1	C

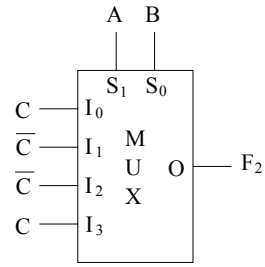
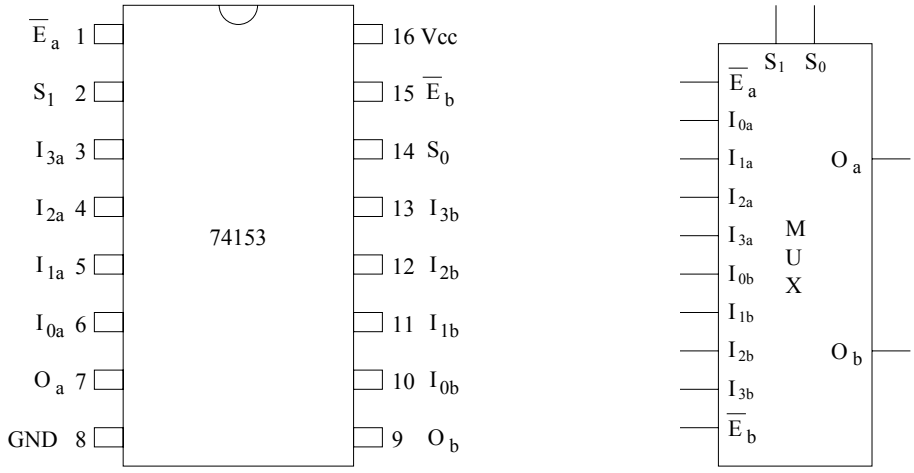


Figure 3.6 Derivation of the reduced truth table and its implementation for the even-parity function.

independent; they both use the same select lines. Each MUX is similar to the 74151 MUX we have seen before. The enable input can be used to disable or enable a MUX. We can use a single 74153 chip to implement both the majority function as well as the even-parity function on the same set of logical variables. By using the enable inputs, we can implement these functions on different logic variables at different times (but not concurrently). This chip, however, is more suitable for implementing two outputs of a multiple output function (see Section 2.9). For example, we can use a single 74153 chip to implement the two output functions given in Table 2.10 on page 74.



(a) Connection diagram (b) Logic symbol

Figure 3.7 The 74153 dual 4-to-1 multiplexer chip. The select lines are common to both multiplexers.

3.2.4 Demultiplexers

The demultiplexer (DeMUX) performs the complementary operation of a multiplexer. As in the multiplexer, a demultiplexer has n selection inputs. However, the roles of data input and output are reversed. In a demultiplexer with n selection inputs, there are 2^n data outputs and one data input. Depending on the value of the selection input, the data input is connected to the corresponding data output. The block diagram and the implementation of a 4-data out demultiplexer is shown in Figure 3.8.

A Demultiplexer Chip: Figure 3.9 shows the connection diagram and logic symbol for the 74138 demultiplexer chip. As we show in the next section, this chip acts as a decoder as well. The logic symbol in Figure 3.9*b* does not have an explicit data input; instead, it has three enable inputs (two low-active, $\overline{\mathrm{E}}_0$ and $\overline{\mathrm{E}}_1$, and one high-active, E_2). These three enable inputs are ANDed as shown in Figure 3.9*c*, and the output of this AND gate is connected to all the AND gates (similar to the data input line in Figure 3.8). We can use any of these enable inputs as a data input, while holding the other two enable inputs at their active level. For example, you can connect data input to E_2 while holding $\overline{\mathrm{E}}_0$ and $\overline{\mathrm{E}}_1$ low. The inputs I_0, I_1, and I_2 are used when the chip functions as a decoder.

3.3 Decoders and Encoders

The decoder is another basic building block that is useful in selecting one-out-of-N lines. The input to a decoder is an I-bit binary (i.e., encoded) number and the output is 2^I bits of decoded

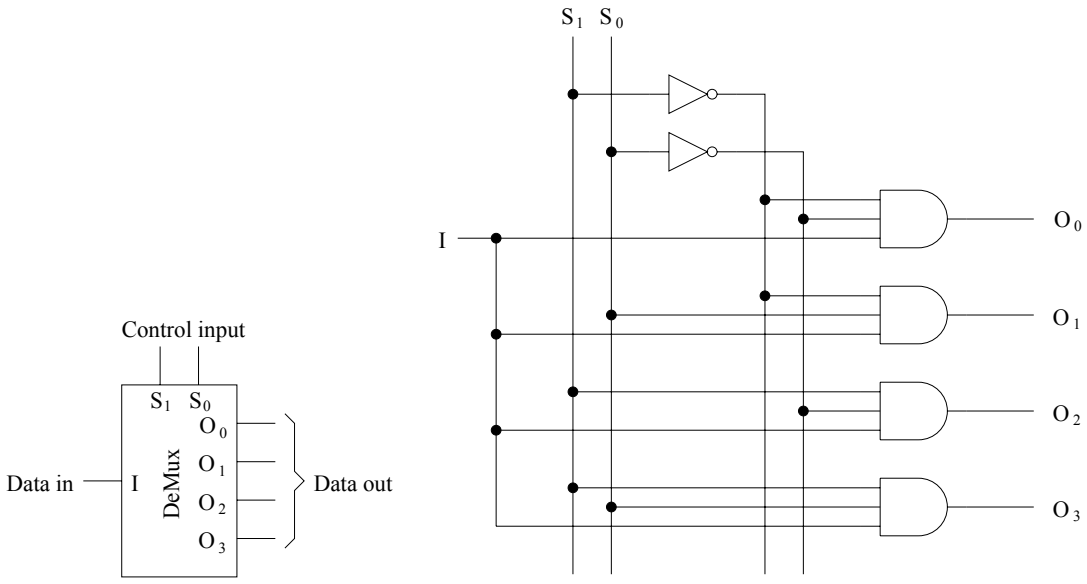


Figure 3.8 Demultiplexer block diagram and its implementation.

data. Figure 3.10 shows a 2-to-4 decoder and its logical implementation. Among the 2^I outputs of a decoder, only one output line is 1 at any time as shown in the truth table (Figure 3.10).

Decoders are also useful in implementing logic functions. Using a single decoder and a set of OR gates, we can implement any logical function. The key idea is that each decoder output is essentially a product term. Thus, if we OR those product terms for which the logical function output is 1, we implement the sum-of-products expression for the logical function. As an example, Figure 3.11 shows how the two logical functions shown in Table 2.10 on page 74 can be implemented using a decoder-OR combination. In Figure 3.11, we have relabeled F_1 as Sum and F_2 as $\mathrm{C}_{\mathrm{out}}$.

We can add decoder and OR as another set of logic devices to our universal set of NAND gates, NOR gates, MUXs, and so on (see the discussion on complete sets on page 45).

3.3.1 Decoder Chips

The 74138 chip can be used as a decoder by activating the three enable inputs. If we connect the two low-active enable inputs $\overline{\mathrm{E}}_0$ and $\overline{\mathrm{E}}_1$ to 0 and E_3 to high, this chip acts as a 3-to-8 decoder.

Figure 3.12 shows details about another decoder that provides two independent 2-to-4 decoders. Some key differences from our decoder implementation shown in Figure 3.10 are the enable input as in the previous multiplexers and the low-active outputs. This means that decoder outputs are high except for the selected line. With these low-active outputs, we need to use NAND gates instead of OR gates to implement a logical function. For example, a single

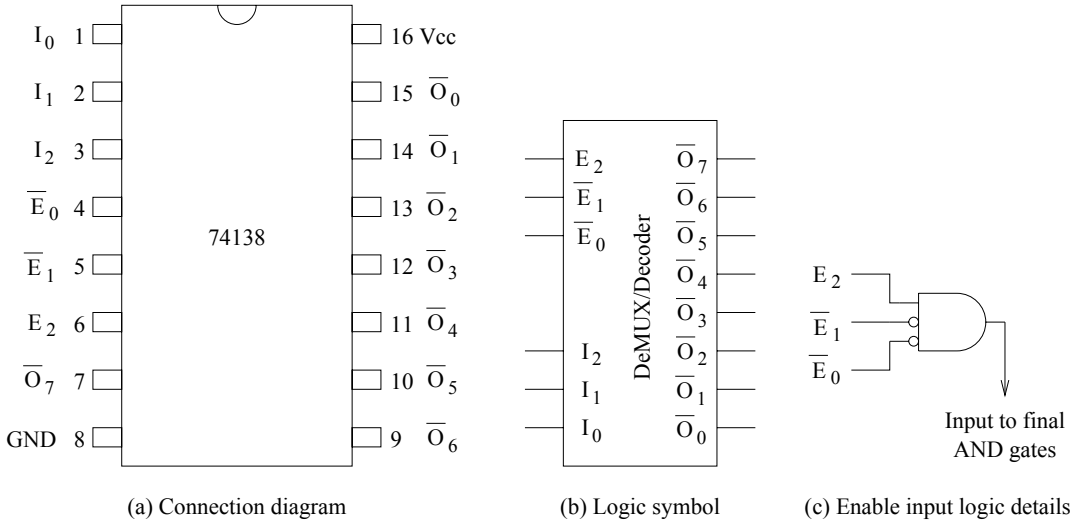


Figure 3.9 The 74138 chip can be used both as a demultiplexer and a decoder.

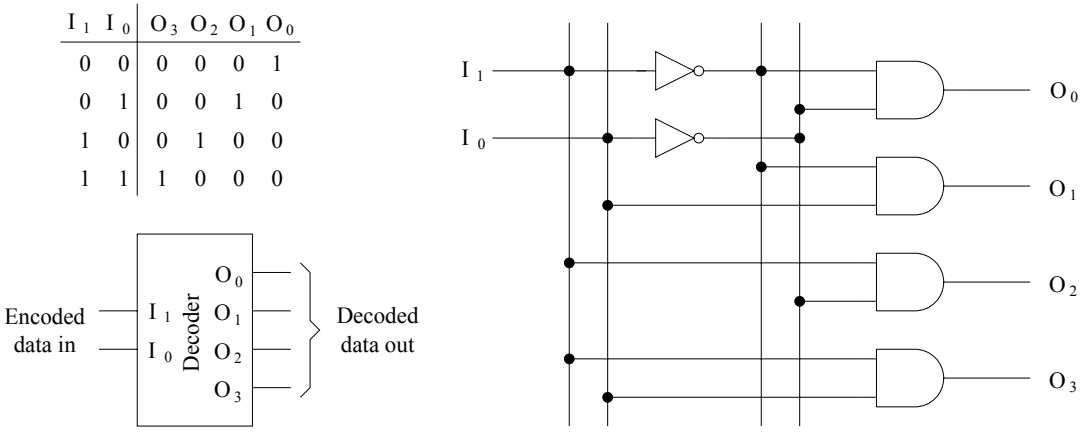


Figure 3.10 Decoder block diagram and its implementation.

74139 chip can be used along with NAND gates to implement two independent multiple output logical functions, each with two variables.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

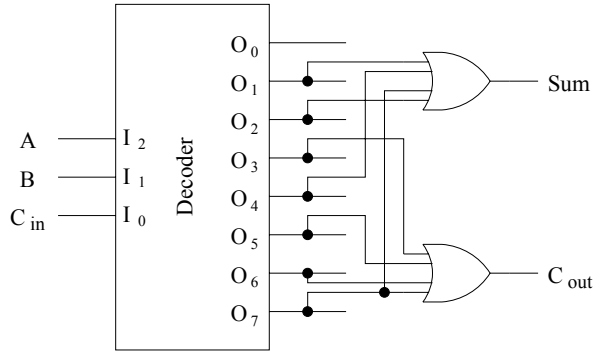
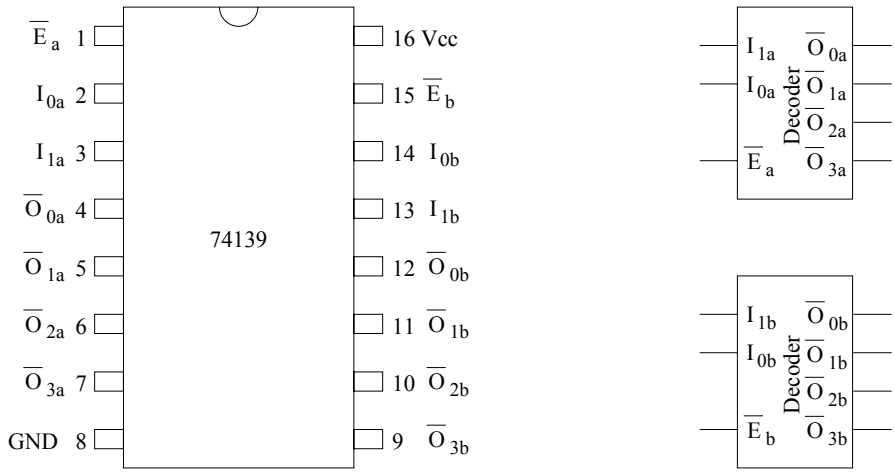


Figure 3.11 Implementation of logical functions using decoder and OR gates.



(a) Connection diagram (b) Logic symbol

Figure 3.12 Dual decoder chip 74139.

3.3.2 Encoders

Encoders perform the reverse operation of decoders. Encoders take 2^B input lines and generate a B-bit binary number on B output lines. The basic truth table of an encoder is similar to the decoder with the inputs and outputs interchanged. The output of an encoder is valid as long as only one of the inputs is 1. Encoders need a way to indicate when the output is valid so that the presence of invalid inputs can be conveyed to the output side. This is typically done by an additional output control line as shown in Figure 3.13. This figure also shows how an enable

Enable input	I_3	I_2	I_1	I_0	O_1	O_0	Input active control signal
0	X	X	X	X	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	1	1
1	0	1	0	0	1	0	1
1	1	0	0	0	1	1	1

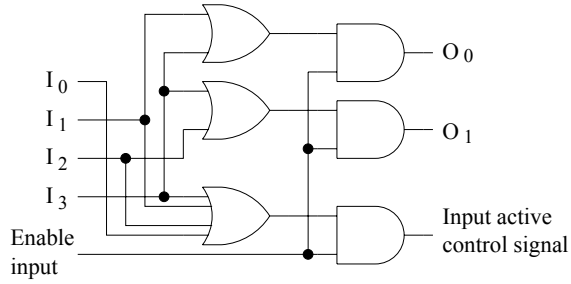


Figure 3.13 A 4-to-2 encoder.

Enable input	I_3	I_2	I_1	I_0	O_1	O_0	Input active control signal
0	X	X	X	X	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	X	0	1	1
1	0	1	X	X	1	0	1
1	1	X	X	X	1	1	1

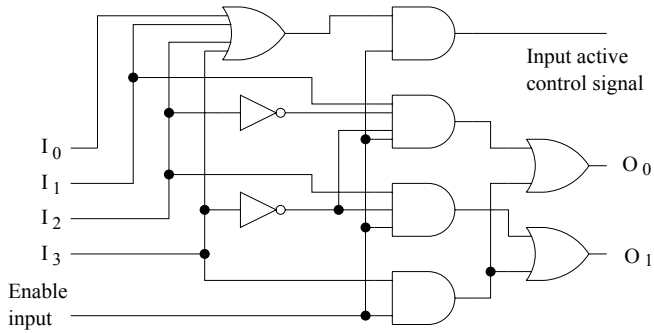


Figure 3.14 A 4-to-2 priority encoder ("X" represents a don't care input).

input can be incorporated into the logic. Notice that the output is valid only when the enable input is high. Furthermore, the output is 00 when the input line $I_3I_2I_1I_0 = 0001$ or 0000. To distinguish these two cases, we can use the valid-input control signal, which is one whenever at least one of the four input signals is one.

The logic circuit shown in Figure 3.13 handles the binary case of no input or some valid input present on the input lines. However, it does not handle the situations where more than one input is high. For example, if $I_3I_2I_1I_0 = 0110$, 11 is the output of the encoder. Clearly this is wrong. One way out of this situation is to assign priorities to the inputs and if more than one input is high, the highest priority input is encoded. Priorities are normally assigned such that I_0 has the lowest priority and I_3 has the highest. In our example, with $I_3I_2I_1I_0 = 0110$, the encoder should output 10, as the highest priority input that has a one is I_2. Such encoders are called *priority encoders*. Figure 3.14 shows a 4-to-2 priority encoder.

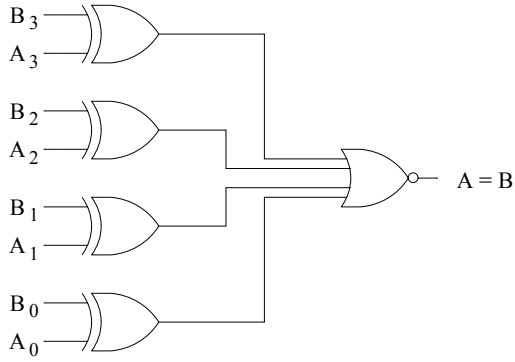


Figure 3.15 A 4-bit comparator implementation using XOR gates.

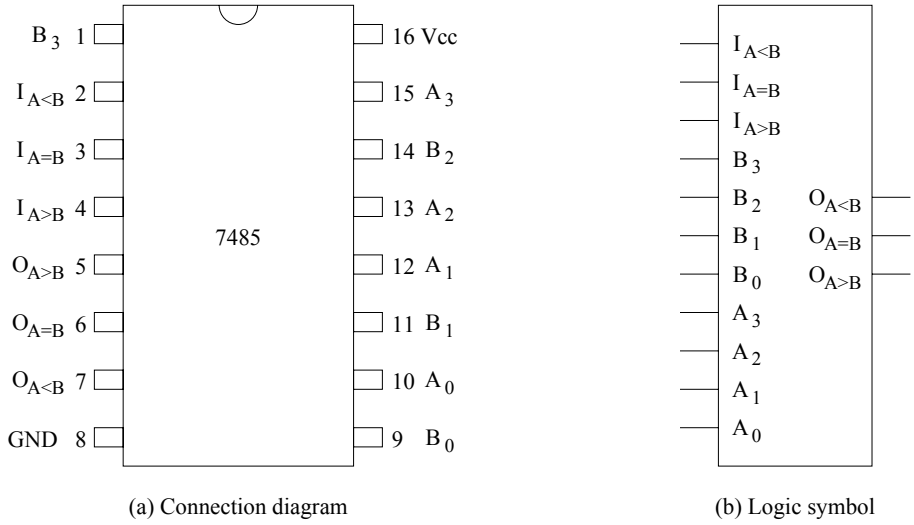
3.4 Comparators

Comparators are useful for implementing relational operations such as =, <, >, and so on. For example, we can use XOR gates to test whether two numbers are equal. Figure 3.15 shows a 4-bit comparator that outputs 1 if the two 4-bit input numbers A = $A_3A_2A_1A_0$ and B = $B_3B_2B_1B_0$ match. However, implementing $<$ and $>$ is more involved than testing for equality. Although equality can be established by comparing bit by bit, positional weights must be taken into consideration when comparing two numbers for $<$ and $>$. We leave it as an exercise to design such a circuit.

3.4.1 A Comparator Chip

Figure 3.16 shows the connection diagram and logic symbol for the 7485 magnitude comparator chip. It compares two 4-bit numbers and provides three outputs: $O_{A=B}$, $O_{A<B}$, and $O_{A>B}$. An interesting feature of this chip is that it takes three expander inputs: $I_{A=B}$, $I_{A<B}$, and $I_{A>B}$. The functionality of this chip is that, when the two 4-bit numbers are not the same, it disregards the expander inputs and the output is either A $>$ B or A $<$ B depending on the relationship between the two numbers. When the two numbers are equal, it essentially copies the expander inputs $I_{A=B}$, $I_{A<B}$, and $I_{A>B}$ to the outputs $O_{A=B}$, $O_{A<B}$, and $O_{A>B}$ (for the truth table and implementation, see the data sheet on the Web).

The expander inputs $I_{A=B}$, $I_{A<B}$, and $I_{A>B}$ can be used to build larger magnitude comparators. For example, we can use a serial (ripple) expansion by connecting outputs of a chip as the expander inputs of the next chip. Figure 3.17 shows how we can build an 8-bit magnitude comparator using two 7485 chips. We can use a similar construction technique to expand to any word length. Assuming that each chip introduces a delay of 10 ns, a W-bit comparator with serial expansion introduces a delay of $(W/4)*10 = 2.5W$ ns. For example, the 8-bit comparator shown in Figure 3.17 introduces a delay of 20 ns. We can also use a parallel construction that reduces this delay substantially. For more details, see the 7485 data sheet available on the Web.



(a) Connection diagram (b) Logic symbol

Figure 3.16 The 7485 four-bit magnitude comparator chip.

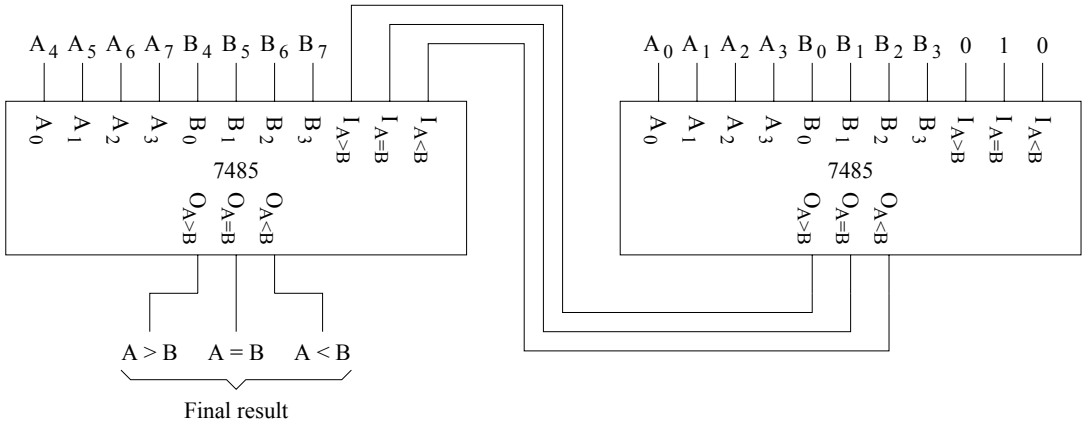
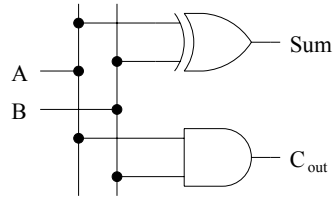


Figure 3.17 Serial construction of an 8-bit comparator using two 7485 chips.

3.5 Adders

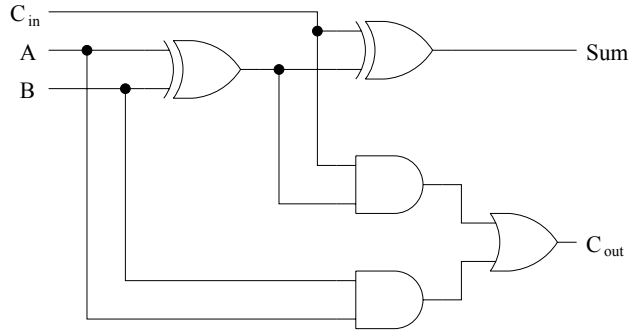
We now look at adder circuits that provide the basic capability to perform arithmetic operations. The simplest of the adders is called a *half-adder*, which adds two bits and produces a sum and carry output as shown in Figure 3.18*a*. From the truth table it is straightforward to see that the carry output C_{out} can be generated by a single AND gate and the sum output by a single XOR gate.

A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



(a) Half-adder truth table and implementation

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



(b) Full-adder truth table and implementation

Figure 3.18 Full- and half-adder truth tables and implementations.

The problem with the half-adder is that we cannot use it to build adders that can add more than two 1-bit numbers. If we want to use the 1-bit adder as a building block to construct larger adders that can add two N-bit numbers, we need an adder that takes the two input bits and a potential carry generated by the previous bit position. This is what the *full-adder* does. A full adder takes three bits and produces two outputs as shown in Figure 3.18*b*. The full-adder implementation follows the design shown in Section 2.10.2 (see Figure 2.27 on page 77).

Using full adders, it is straightforward to build an adder that can add two N-bit numbers. An example 16-bit adder is shown in Figure 3.19. Such adders are called *ripple-carry adders* as the carry ripples through bit positions 1 through 15. Let us assume that this ripple-carry adder is using the full adder shown in Figure 3.18*b*. If we assume a gate delay of 5 ns, each full adder takes three gate delays (=15 ns) to generate C_{out}. Thus, the 16-bit ripple-carry adder shown in Figure 3.19 takes $16 \times 15 = 240$ ns. If we were to use this type of adder circuit in a system, it cannot run more than 1/240 ns = 4 MHz with each addition taking about a clock cycle.

How can we speed up multibit adders? If we analyze the reasons for the "slowness" of the ripple-carry adders, we see that carry propagation is causing the delay in producing the final N-bit output. If we want to improve the performance, we have to remove this dependency and

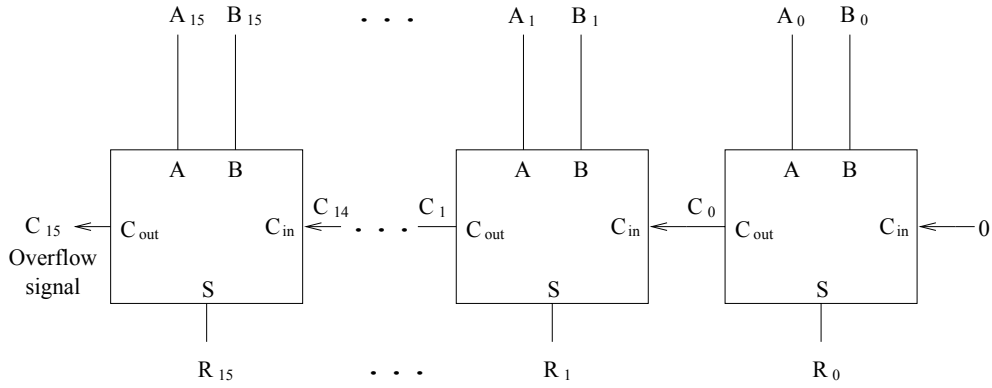


Figure 3.19 A 16-bit ripple-carry adder using the full adder building blocks.

determine the required carry-in for each bit position independently. Such adders are called *carry lookahead adders*. The main problem with these adders is that they are complex to implement for long words. To see why this is so and also to give an idea of how each full adder can generate its own carry-in bit, let us look at the logical expression that should be implemented to generate the carry-in. Carry-out from the rightmost bit position C_0 is obtained as

$$C_0 = A_0 B_0 .$$

C_1 is given by

$$C_1 = C_0 (A_1 + B_1) + A_1 B_1 .$$

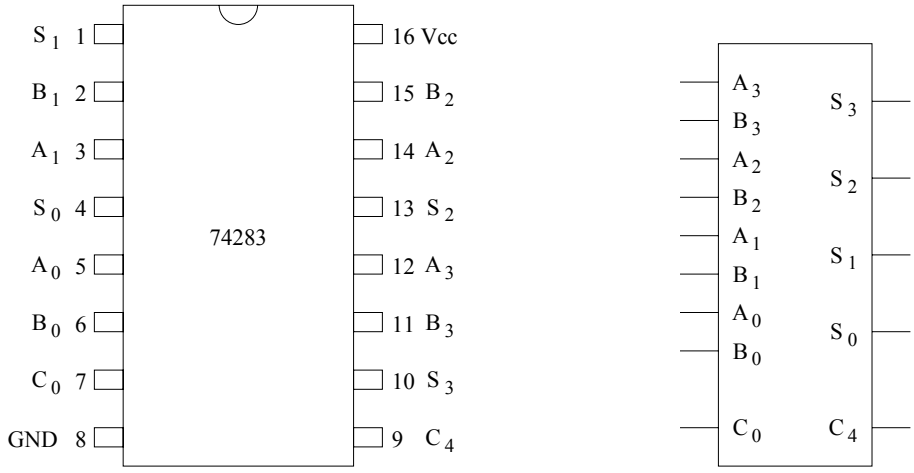
By substituting $A_0 B_0$ for C_0, we get

$$C_1 = A_0 B_0 A_1 + A_0 B_0 B_1 + A_1 B_1 .$$

Similarly, we get C_2 as

$$\begin{aligned} C_2 &= C_1 (A_2 + B_2) + A_2 B_2 \\ &= A_2 A_0 B_0 A_1 + A_2 A_0 B_0 B_1 + A_2 A_1 B_1 \\ &\quad + B_2 A_0 B_0 A_1 + B_2 A_0 B_0 B_1 + B_2 A_1 B_1 + A_2 B_2 . \end{aligned}$$

Using this procedure, we can generate the necessary carry-in inputs independently. The logical expression for C_i is a sum-of-products expression involving only A_k and B_k, $i \leq k \leq 0$. Thus, independent of the length of the word, only two gate delays are involved, since each product term can be implemented by a single gate. The complexity of implementing such a circuit makes it impractical for more than 8-bit words. Typically, carry lookahead is implemented at the 4- or 8-bit level. We can apply our ripple-carry method of building higher word length adders by using these 4- or 8-bit carry lookahead adders. We illustrate the advantage of this scheme next.



(a) Connection diagram (b) Logic symbol

Figure 3.20 The 74283 adder is a 4-bit carry lookahead adder.

3.5.1 An Example Adder Chip

The 74283 is a 4-bit binary adder with internal carry lookahead. Figure 3.20 shows the connection diagram and logic symbol for the 74283 chip. The data on a faster version of this chip available from Motorola indicate that the maximum delay is about 10 ns to complete the addition. If we use four 74283 chips to build a 16-bit adder, it only takes about 40 ns to complete the 16-bit addition (six times faster than our 240 ns ripple-carry adder).

3.6 Programmable Logic Devices

We have seen several ways of implementing sum-of-products expressions. Programmable logic devices provide yet another way to implement these expressions. There are two types of these devices that are very similar to each other. The next two subsections describe these devices.

3.6.1 Programmable Logic Arrays (PLAs)

PLA is a field programmable device to implement sum-of-product expressions. It consists of an AND array and an OR array as shown in Figure 3.21. A PLA takes N inputs and produces M outputs. Each input is a logical variable. Each output of a PLA represents a logical function output. Internally, each input is complemented, and a total of $2N$ inputs is connected to each AND gate in the AND array through a fuse. The example PLA, shown in Figure 3.21, is a 2×2 PLA with two inputs and two outputs. Each AND gate receives four inputs: I_0, $\overline{I}_0$, I_1, and $\overline{I}_1$. The fuses are shown as small white rectangles. Each AND gate can be used to implement a product term in the sum-of-products expression.

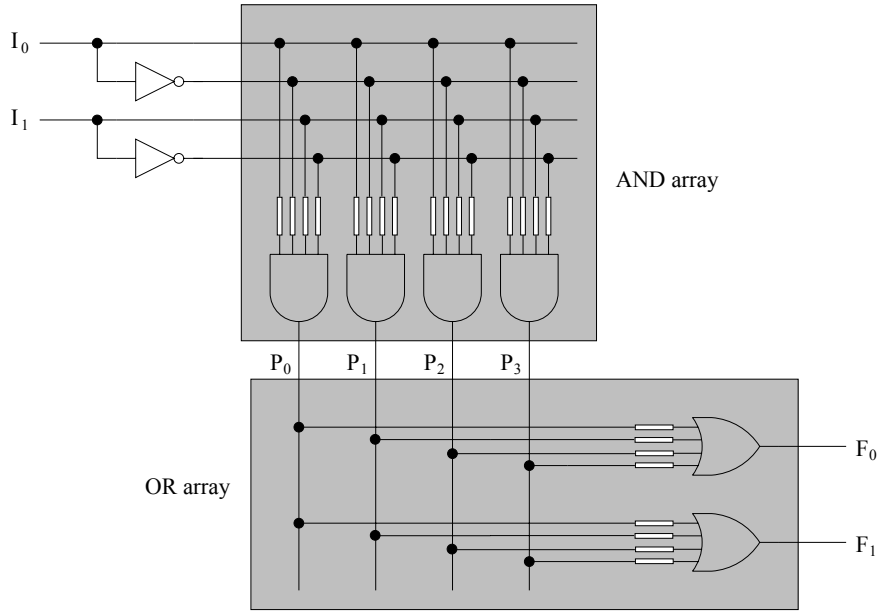


Figure 3.21 An example PLA with two inputs and two outputs.

The OR array is organized similarly except that the inputs to the OR gates are the outputs of the AND array. Thus, the number of inputs to each OR gate is equal to the number of AND gates in the AND array. The output of each OR gate represents a function output.

When the chip is shipped from the factory, all fuses are intact. We program the PLA by selectively blowing some fuses (generally by passing a high current through them). The chip design guarantees that an input with a blown fuse acts as 1 for the AND gates and as 0 for the OR gates.

Figure 3.22 shows an example implementation of functions F_0 and F_1. The rightmost AND gate in the AND array produces the product term $A\,B$. To produce this output, the inputs of this gate are programmed by blowing the second and fourth fuses that connect inputs $\overline{A}$ and $\overline{B}$, respectively. Programming a PLA to implement a sum-of-products function involves implementing each product term by an AND gate. Then a single OR gate in the OR array is used to obtain the final function. In Figure 3.22, we are using two product terms generated by the middle two AND gates (P_1 and P_2) as inputs to both OR gates as these two terms appear in both F_0 and F_1.

To simplify specification of the connections, the notation shown in Figure 3.23 is used. Each AND and OR gate input is represented by a single line. A $\times$ is placed if the corresponding input is connected to the AND or OR gates as shown in the figure.

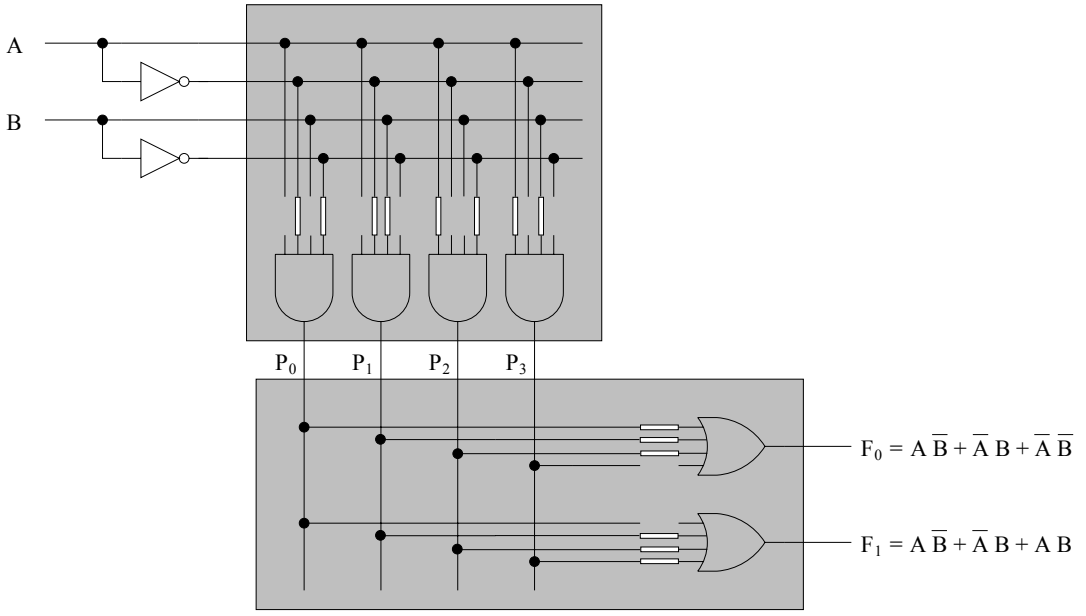


Figure 3.22 Implementation of functions F_0 and F_1 using the example PLA.

3.6.2 Programmable Array Logic Devices (PALs)

PLAs are very flexible in implementing sum-of-products expressions. However, the cost of providing a large number of fuses is high. For example, a 12×12 PLA with a 50-gate AND array and 12-gate OR array requires $24 \times 50 = 1200$ fuses for the AND array and $50 \times 12 = 600$ fuses for the OR array for a total of 1800 fuses. We can reduce this complexity by noting that we can retain most of the flexibility by cutting down the set of fuses in the OR array. This is the rationale for PALs. Due to their cost advantage, most manufacturers produce only PALs.

PALs are very similar to PLAs except that there is no programmable OR array. Instead, OR connections are fixed. Figure 3.24 shows a PAL with the bottom OR gate connected to the leftmost two product terms and the other OR gate connected to the other two product terms. As a result of these connections, we cannot implement the two functions shown in Figure 3.23. This is the loss of flexibility that sometimes may cause problems but in practice is not such a big problem. Next we discuss an example PAL to give you an idea about what is available commercially. But the advantage of PAL devices is that we can cut down all the OR array fuses that are present in a PLA. In the last example, we reduce the number of fuses by a third (from 1800 fuses to 1200).

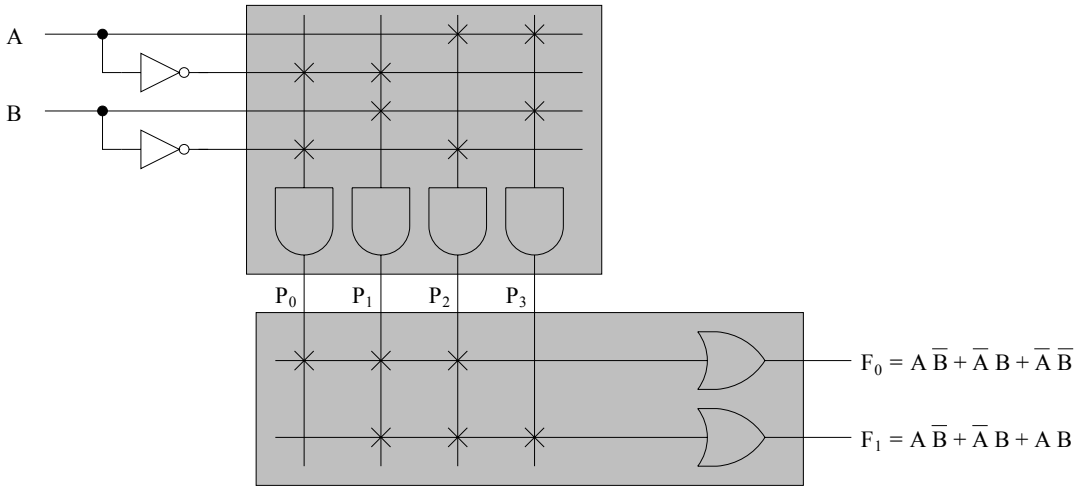


Figure 3.23 A simplified notation to show implementation details of a PLA.

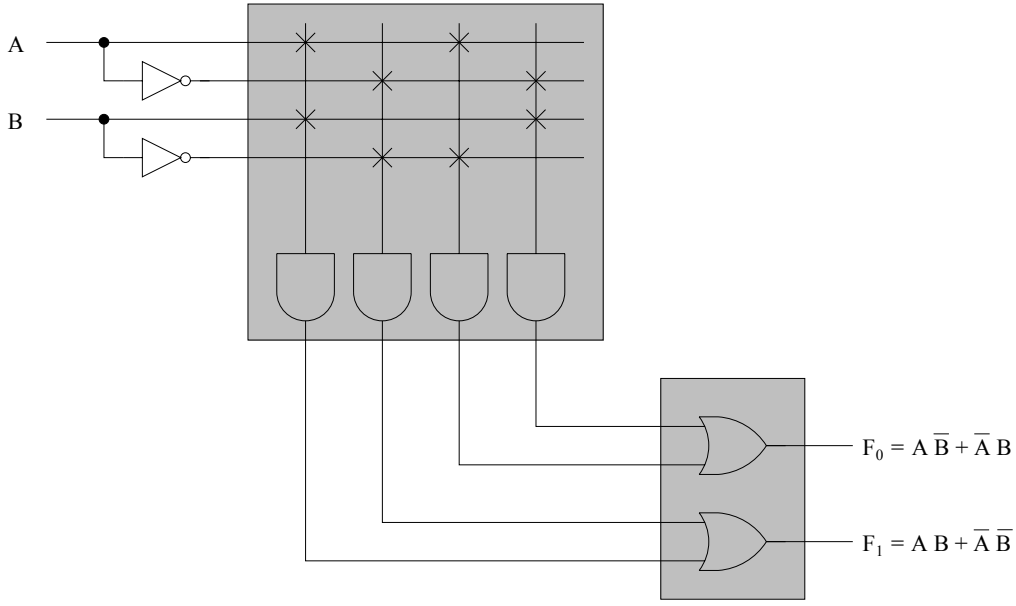


Figure 3.24 Programmable array logic device with fixed OR gate connections. We have used the simplified notation to indicate the connections in the AND array.

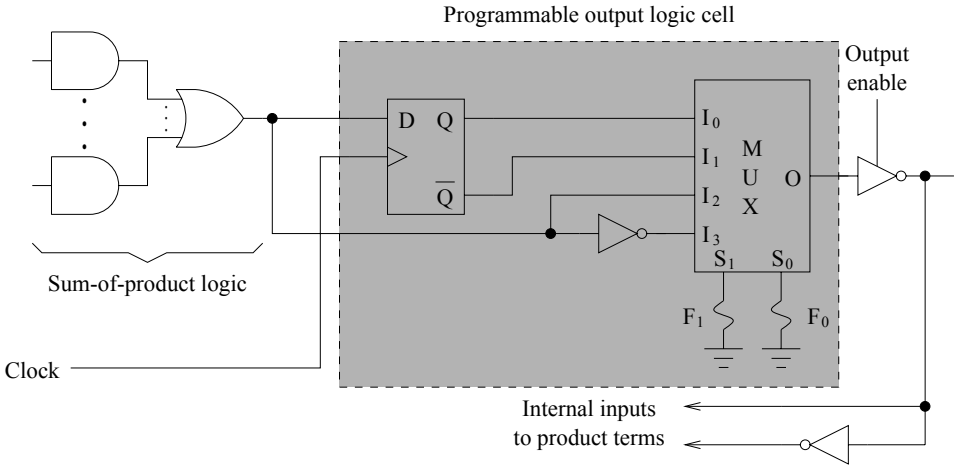


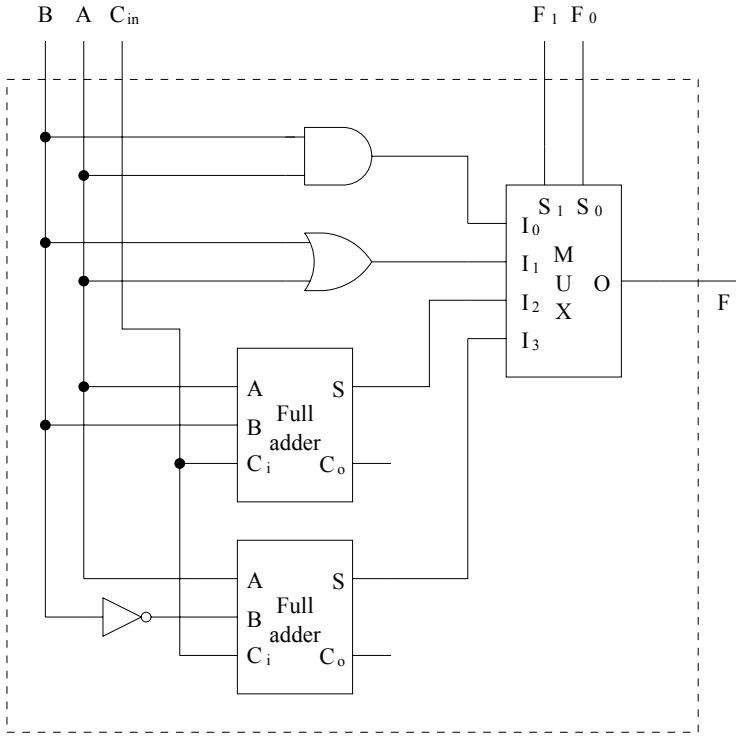
Figure 3.25 A typical programmable output cell of PAL devices.

An Example PAL

As an example of a PAL, we briefly describe the Texas Instruments TIBPAL22V10-10C PAL. This is a 22×10 PAL device that uses titanium–tungsten fuses. It is packaged as a 24-pin DIP. Internally, it consists of a 120-gate AND array and 10-gate OR array. As it takes 22 inputs, the number of fuses in this chip is $44 \times 120 = 5280$! All this just for the AND array. Being a PAL device, the OR array has fixed connections. This chip provides a variable number of connections for the OR gates. There are two each of 8-, 10-, 12-, 14-, and 16-input OR gates to provide flexibility in implementing logical functions. This means we can implement sum-of-product expressions from 8 to 16 product terms.

An additional feature of this device is that it allows internal feedback through a programmable output cell. A simplified version of this cell is shown in Figure 3.25. As shown in this figure, to increase flexibility, it provides both registered (through a D flip-flop) and unregistered output of an OR gate and its complement as output. The select-inputs S_0 and S_1 of the output multiplexer are programmable through two fuses F_0 and F_1. The output of the multiplexer goes through a tristate inverter. We discuss tristate devices in Chapter 16. For now it is sufficient to know that this inverter acts as an open circuit when the enable input is low, and acts as a normal inverter if the enable input high.

How can we have a 24-pin chip that takes 22 inputs and produces 10 outputs? Furthermore, we need two pins for Vcc and ground connections. The trick is to multiplex pins in time. That is, 10 of these pins act both as input and output (for more details, see the data sheet).



F_1	F_0	F
0	0	A and B
0	1	A or B
1	0	A + B
1	1	A − B

Figure 3.26 A simple 1-bit ALU that can perform addition, subtraction, AND, and OR operations. The carry output of the circuit is incomplete in this figure as a better and more efficient circuit is shown in the next figure. Note: "+" and "−" represent arithmetic addition and subtraction operations, respectively.

3.7 Arithmetic and Logic Units

We are now ready to design our own arithmetic and logic unit. The ALU forms the computational core of a processor, performing basic arithmetic and logical operations such as integer addition, subtraction, and logical AND and OR functions. Figure 3.26 shows an example ALU that can perform two arithmetic functions (addition and subtraction) and two logical functions (AND and OR). We use a multiplexer to select one of the four functions. The implementation is straightforward except that we implement the subtractor using a full adder by negating the B input.

To implement the subtract operation, we first convert B to $-$B in 2's complement representation. We get the 2's complement representation by complementing the bits and adding 1. We need an inverter to complement. The required 1 is added via C_{in}.

Since the difference between the adder and subtractor is really the negation of the one input, we can get a better circuit by using a programmable inverter. Figure 3.27 shows the final design with the XOR gate acting as a programmable inverter. Remember that the XOR gate acts as an

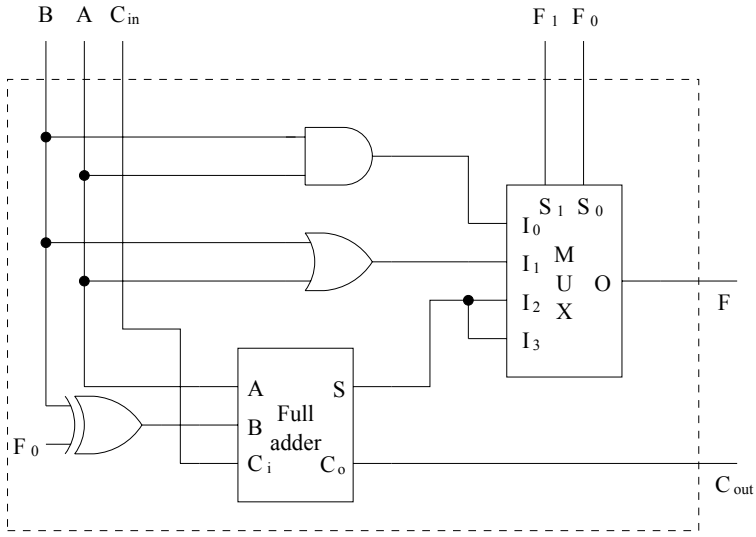


Figure 3.27 A better 1-bit ALU that uses a single full adder for both addition and subtraction operations.

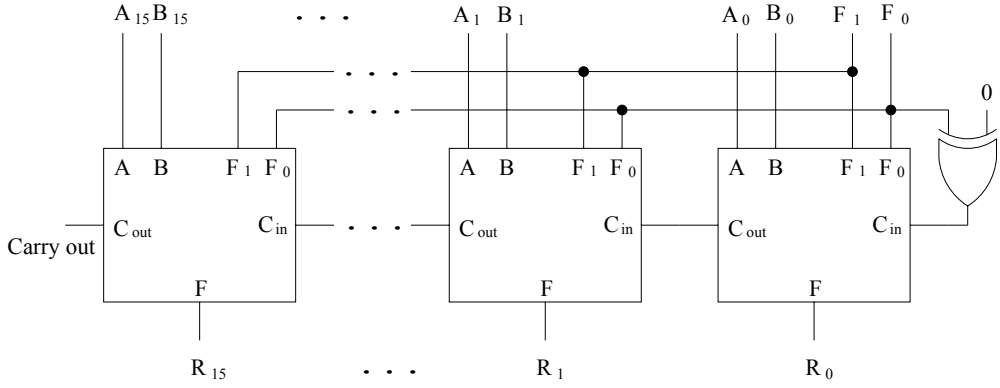


Figure 3.28 A 16-bit ALU built with the 1-bit ALU: The XOR gate sets C_{in} to 1 for the subtract operation. Logical operations ignore the carry bits.

inverter when one of the inputs is one. We can use these 1-bit ALUs to get word-length ALUs. Figure 3.28 shows an implementation of a 16-bit ALU using the 1-bit ALU of Figure 3.27.

To illustrate how the circuit in Figure 3.28 subtracts two 16-bit numbers, let us consider an example with A = 1001 1110 1101 1110 and B = 0110 1011 0110 1101. Since B is internally complemented, we get $\overline{B}$ = 1001 0100 1001 0010. Now we add A and $\overline{B}$ with the carry-in to the rightmost bit set to 1 (through the XOR gate):

$1 \leftarrow$ carry-in from the XOR gate

$$\begin{array}{rcl} A & = & 1001\ 1110\ 1101\ 1110 \\ \overline{B} & = & \underline{1001\ 0100\ 1001\ 0010} \\ A - B & = & 0011\ 0011\ 0111\ 0001 \end{array}$$

which is the correct value. If B is larger than A, we get a negative number. In this case, the result will be in 2's complement form. Also, with 2's complement representation, we ignore any carry generated out of the most significant bit.

3.7.1 An Example ALU Chip

Figure 3.29 shows an example 4-bit ALU chip. It supports 16 different functions. The exact functions supported depend on whether we are using high-active (positive logic) or low-active inputs (negative logic). The functions supported include arithmetic as well as logic functions. When the mode control input (M) is high, it performs all 16 possible logic operations. Recall from Section 2.2.2 on page 44, with two inputs, we can have $2^{2^2} = 16$ different logical functions. When this input is low, it performs 16 different arithmetic operations on two 4-bit numbers. For more details, see the data sheet for this chip.

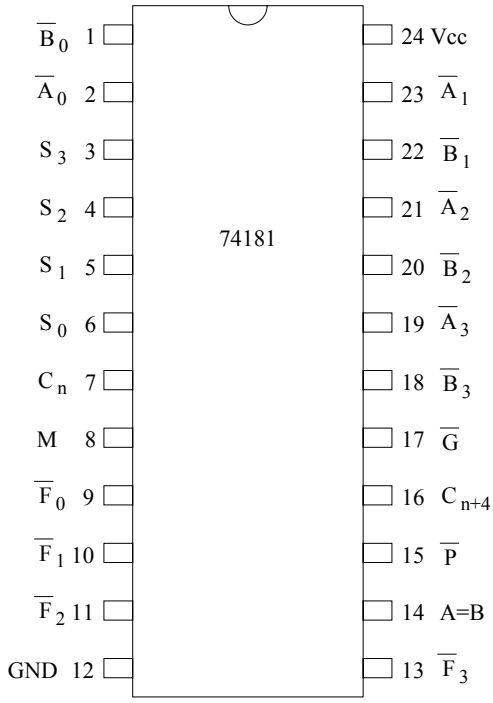
3.8 Summary

Combinational circuits provide a higher level of abstraction than the basic circuits discussed in the last chapter. Higher-level logical functionality provided by these circuits helps in the design of complex digital circuits.

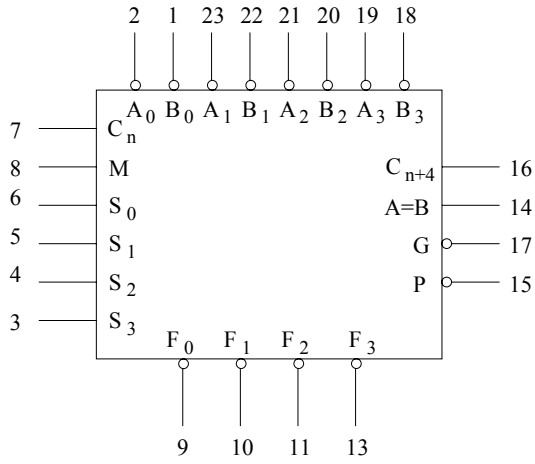
We have discussed several commonly used combinational circuits including multiplexers and demultiplexers, decoders and encoders, comparators, adders, and ALUs. We have shown how multiplexers can be used to implement any logical function. This proves that multiplexers are universal just like the NAND and NOR gates. Similarly, decoders along with OR gates are also universal.

We have also presented details about two types of programmable logic devices: PLAs and PALs. These devices can also be used to implement any logical function. Both these devices use internal fuses that can be selectively blown to program the device to implement a given logical function. PALs reduce the complexity of the device by using fewer fuses than PLAs. As a result, most commercial implementations of programmable logic devices are PALs.

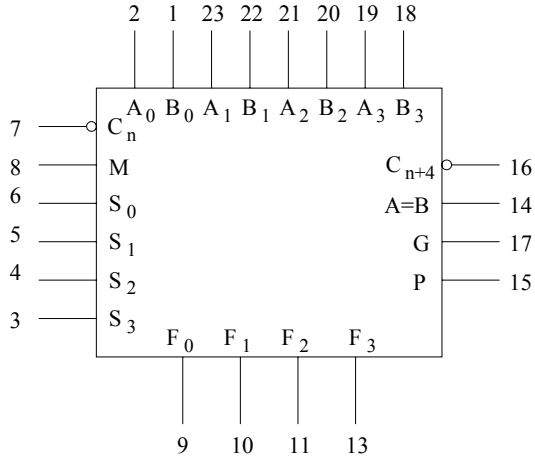
Our discussion of ALU design suggests that complex digital circuit design can be simplified by using the higher level of abstraction provided by the combinational circuits.



(a) Connection diagram



(b) Active low operands



(c) Active high operands

Figure 3.29 An example 4-bit ALU chip.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Adders
- ALUs
- Carry lookahead adders
- Comparators
- Decoders
- Demultiplexers
- Encoders
- Full adder
- Half-adder
- Multiplexers
- Programmable array logic devices (PALs)
- Programmable logic arrays (PLAs)
- Ripple-carry adders

3.9 Exercises

3–1 In the two multiplexer implementations shown in Figure 3.3 does it matter how we connect the inputs A, B, and C to the control inputs S_0, S_1, and S_2?

3–2 Prove that the 4-data input multiplexer is universal by showing how 2-input AND and NOT gates can be implemented using this multiplexer.

3–3 Prove that the 2-data input multiplexer is universal by showing how 2-input AND and NOT gates can be implemented using this multiplexer.

3–4 We have discussed even-parity implementation in Figure 3.3. Show how an odd-parity function can be implemented using an 8-to-1 multiplexer.

3–5 Implement the odd-parity function of the last exercise using a 4-to-1 multiplexer. If you need to, you can use one additional inverter in your implementation.

3–6 Implement the majority function described in Exercise 2–9 on page 80 using a 4-to-1 multiplexer. If you need to, you can use one additional inverter in your implementation.

3–7 Implement the majority function described in Exercise 2–10 on page 80 using a 4-to-1 multiplexer. If you need to, you can use one additional inverter in your implementation.

3–8 In Exercise 2–24 on page 81, you have designed an even-parity generator for the 7-bit ASCII character using XOR gates. In this exercise, we want to generate the same using multiplexers. Design a circuit to generate this parity using two 74151 multiplexer chips and a 2-input XOR gate.

3–9 What is the major problem with ripple-carry adders? Despite this problem, why do we still use them in practice?

3–10 Discuss the advantages and disadvantages of carry lookahead adders over ripple-carry adders.

3–11 Show an implementation of the full adder by using two half-adders and a 2-input OR gate.

3–12 Implement the full adder, shown in Figure 3.18, using a 74153 multiplexer chip.

3–13 Implement the priority encoder, shown in Figure 3.14, using a 74153 multiplexer chip.

3–14 Give the truth table for a full subtractor with three input bits (two data bits and a borrow bit) analogous to the full adder. Implement it using a 74153 multiplexer chip.

3–15 Consider the seven-segment display shown in Figure 2.20 on page 65. Each segment is a light emitting diode that is turned on (gives out light) if a logical 1 is given as input to the driver circuit of the LED. Typically, a logical circuit converts a 4-bit BCD number to display digits 0 through 9. Assume that the LED **d** is on to display digit 9. You can assume that inputs 10 through 15 are not given to the circuit.

(a) Give the truth table for the logical circuit that drives LED **d**.

(b) Implement your truth table in part (a) using a single 8-to-1 multiplexer. Your solution should not use any additional logic gates.

3–16 Design a 2-bit comparator that outputs 1 when $A > B$, where A and B are 2-bit unsigned numbers.

3–17 Design a 2-bit comparator that outputs 1 when $A < B$, where A and B are 2-bit unsigned numbers.

3–18 Design a 2-bit comparator that outputs 1 when $A > B$, where A and B are 2-bit signed numbers.

3–19 Give the truth table for a 2-bit by 2-bit multiplier. Assume that both 2-bit inputs are unsigned numbers. Show an implementation using 74151 multiplexer chips.

3–20 Give the truth table for a 2-bit by 2-bit divider. Assume that both 2-bit inputs are unsigned numbers. Show an implementation using 74151 multiplexer chips.

3–21 What is the difference between PLAs and PALs?

3–22 Show an implementation of a full adder and a full subtractor using a PLA (with three inputs and two outputs). Use the simplified notation to express your design.

3–23 Implement the priority encoder, shown in Figure 3.14, using a PLA (with four inputs and two outputs). Use the simplified notation to express your design.

3–24 Why is F_0 connected to the XOR gate in the 1-bit ALU shown in Figure 3.27?

3–25 What is the purpose of the XOR gate in the 16-bit ALU shown in Figure 3.28?

3–26 Show how the two output functions given in Table 2.10 (page 74) are implemented using a single 74153 chip.

Chapter 4

Sequential Logic Circuits

Objectives

- To bring out the differences between combinational and sequential circuits;
- To introduce the basic building blocks—latches and flip-flops—that can store a single bit;
- To describe how simple flip-flops can be used to design more complex sequential circuits such as counters and shift registers;
- To provide a methodology for designing general sequential circuits.

Our focus in the last two chapters was on the design of combinational circuits. In this chapter, we concentrate on sequential circuits. In sequential circuits, current output depends on the current inputs as well as the past history of inputs. This aspect makes the design of sequential circuits more complex. We start the chapter with a high-level overview of sequential circuits (Section 4.1). Since the behavior of these circuits depends on the past input sequence, we have to introduce the notion of time. We discuss clock signals, which provide this timing information, in Section 4.2. We start our discussion of sequential circuits with some simple circuits that can store a single bit of information. These circuits include latches (Section 4.3) and flip-flops (Section 4.4). The following section discusses more complex sequential circuits such as counters and shift registers. We introduce the sequential circuit design process in Section 4.6. We conclude the chapter with a summary.

4.1 Introduction

We discussed combinational circuits in the last two chapters. The defining characteristic of a combinational circuit is that its output depends only on the current inputs applied to the circuit. The output of a sequential circuit, on the other hand, depends both on the current input values as well as the past inputs. This dependence on past inputs gives the property of "memory" for sequential circuits.

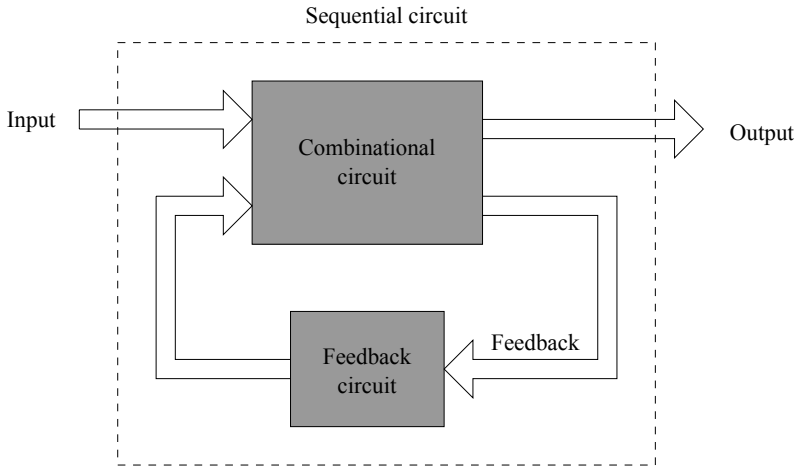
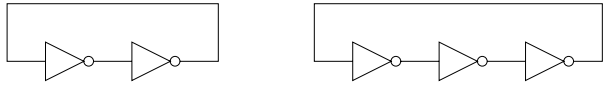


Figure 4.1 Main components of a sequential circuit.



(a) Stable circuit (b) Unstable circuit

Figure 4.2 Stable and unstable circuits.

In general, the sequence of past inputs is encoded into a set of state variables. There is a feedback path that feeds these variables to the input of a combinational circuit as shown in Figure 4.1. Sometimes, this feedback consists of a simple interconnection of some outputs of the combinational circuit to its inputs. For the most part, however, the feedback circuit consists of elements such as flip-flops that we discuss later in this chapter. These elements themselves are sequential circuits that can remember or store the state information.

Once we introduce feedback, we also introduce potential instability into the system. As a simple example, consider the circuit shown in Figure 4.2*a*. This circuit is stable in the sense that the output of each inverter can stay at a particular level. However, this circuit is indeterministic as we cannot say what the output level of each inverter is. Outputs of the first and second inverters could be 0 and 1, respectively. The outputs could also be 1 and 0, instead. In contrast, you can verify that the circuit in Figure 4.2*b* is unstable.

We have discussed several methods to design a combinational circuit. For example, see the design process described in Section 2.5 on page 55. The major step in designing a combinational circuit is the simplification of logical expressions. We have discussed several methods including intuition-based designs as well as more formal methods such as the Karnaugh map and Quine–McCluskey methods.

For sequential circuits, the process is not as simple due to their dependence on the past inputs. This makes it even more imperative to develop a theoretical foundation for designing arbitrary sequential circuits. Towards the end of the chapter we discuss a method to give you an idea of the process involved in designing sequential circuits. However, before we present this method, we develop our intuition by looking at some simple sequential circuits such as latches and flip-flops.

4.2 Clock Signal

Digital circuits can operate in *asynchronous* or *synchronous* mode. Circuits that operate in asynchronous mode are independent of each other. That is, the time at which a change occurs in one circuit has no relation to the time a change occurs in another circuit. Asynchronous mode of operation causes serious problems in a typical digital system in which the output of one circuit goes as input to several others. Similarly, a single circuit may receive outputs of several circuits as inputs. Asynchronous mode of operation implies that all required inputs to a circuit may not be valid at the same time.

To avoid these problems, circuits are operated in synchronous mode. In this mode, all circuits in the system change their state at some precisely defined instants. The clock signal in a digital system provides such a global definition of time instants at which changes can take place. Implicit in this definition is the fact that the clock signal also specifies the speed at which a circuit can operate.

A clock is a sequence of 1s and 0s as shown in Figure 4.3. We refer to the period during which the clock is 1 as the ON period and the period with 0 as the OFF period. Even though we normally use symmetric clock signals with equal ON and OFF periods as in Figure 4.3*a*, clock signals can take asymmetric forms as shown in Figures 4.3*b* and *c*.

The clock signal edge going from 0 to 1 is referred to as the *rising edge* (also called the positive or leading edge). Analogously, we can define a *falling edge* as shown in Figure 4.3*a*. The falling edge is also referred to as a negative or trailing edge.

A clock cycle is defined as the time between two successive rising edges as shown in Figure 4.3. You can also treat the period between successive falling edges as a clock cycle.

Clock rate or frequency is measured in number of cycles per second. This number is referred to as Hertz (Hz). The abbreviation MHz refers to millions of cycles per second. The clock period is defined as the time represented by one clock cycle. All three clock signals in Figure 4.3 have the same clock period.

$$\text{Clock period} \;=\; \frac{1}{\text{Clock frequency}}\,.$$

For example, a clock frequency of 100 MHz yields a clock period of

$$\frac{1}{100 \times 10^6} \;=\; 10 \text{ ns.}$$

Note that one nanosecond (ns) is equal to 10^{-9} second.

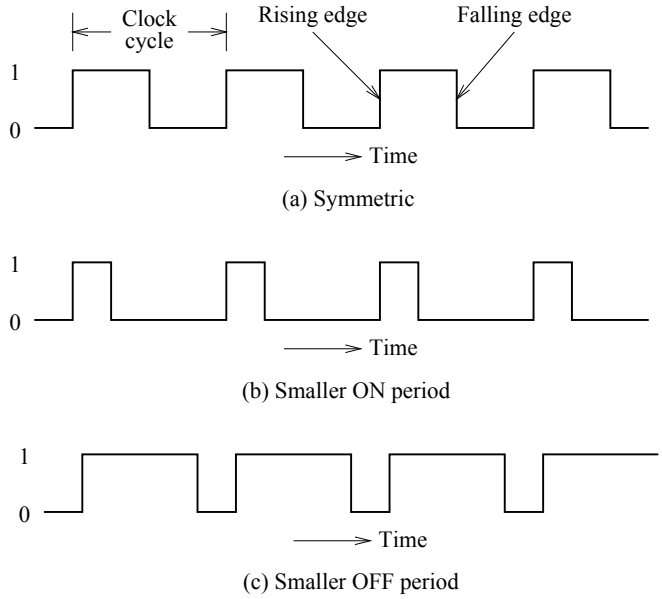


Figure 4.3 Three types of clock signals with the same clock period.

The clock signal serves two distinct purposes in a digital circuit. It provides the global synchronization signal for the entire system. Each clock cycle provides three distinct epochs: start of a clock cycle, end of a clock cycle, and an intermediate point at which the clock signal changes levels. This intermediate point is in the middle of a clock cycle for symmetric clock signals. The other equally important purpose is to provide timing information in the form of a clock period (e.g., time to complete an operation such as logical AND).

We have mentioned in Chapter 2 that propagation delay represents the delay involved from input to output (see page 49). A typical logic gate such as the AND gate has a propagation delay of about 10 ns. To see how propagation delay affects timing signals, look at the circuit shown in Figure 4.4*a*. Assuming that ΔT = 10 ns, the input clock signal is inverted and delayed at input X to the AND gate. When we AND the two signals at the input of the AND gate, we get a signal whose ON period is ΔT (see the signal labeled "X AND Input"). However, assuming that the AND gate also has a propagation delay of ΔT, this signal appears at the output of the AND gate with ΔT time delay as shown in Figure 4.4.

This discussion may lead you to believe that the propagation delay determines the maximum clock frequency that a circuit can handle. A careful look at the propagation delay definition convinces us that this is not true. The characteristic that limits the frequency is the input hold time. Input *hold time* refers to the minimum time that the input signal should be held constant for proper operation of the device. For simple gates, this input signal hold time is approximately equal to the propagation delay. That is, the input should be held steady during the time of its

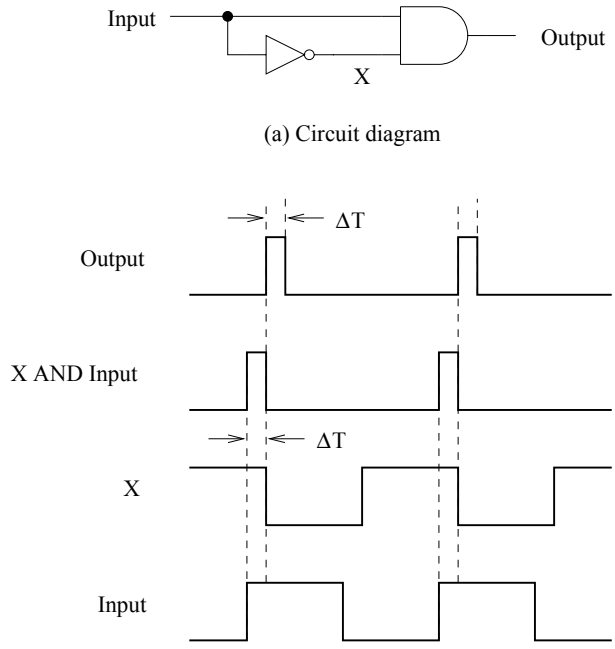


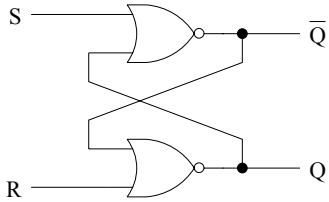
Figure 4.4 Effect of propagation delay on logic circuit output.

propagation. That's why propagation delay appears to control the maximum frequency at which a gate can operate.

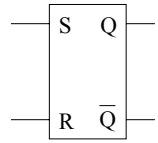
For the sequential circuits that require a clock signal, data sheets also specify the minimum value for clock ON and/or OFF periods. These periods are referred to as HIGH and LOW periods. For example, the data sheet from Motorola for the 74F164 shift register chip specifies that the ON and OFF periods should be at least 4 ns. This specification implies that this chip cannot operate at frequencies higher than 1/8 ns = 125 MHz. The actual maximum frequency rating for this chip is at least 80 MHz. Other conditions such as signal setup, hold times, and the like contribute to the lower maximum frequency than what we got as 125 MHz.

4.3 Latches

It is time to look at some simple sequential circuits that can remember a single bit value. We discuss latches in this section. Latches are level-sensitive devices in that the device responds to the input signal levels (high or low). In contrast, flip-flops are edge-triggered. That is, output changes only at either the rising or falling edge. We look at flip-flops in the next section.



(a) Circuit diagram



(b) Logic symbol

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
~~1~~	~~1~~	~~0~~

(c) Truth table

Figure 4.5 A NOR gate implementation of the SR latch.

4.3.1 SR Latch

The SR latch is the simplest of the sequential circuits that we consider. It requires just two NOR gates. The feedback in this latch is a simple connection from the output of one NOR gate to the input of the other NOR gate as shown in Figure 4.5*a*. The logic symbol for the SR latch is shown in Figure 4.5*b*.

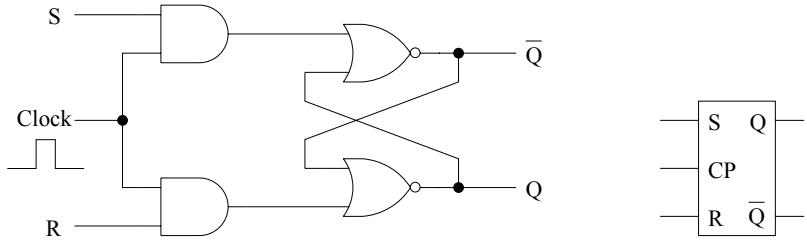
A simplified truth table for the SR latch is shown in Figure 4.5*c*. The outputs of the two NOR gates are labeled Q and $\overline{Q}$ because these two outputs should be complementary in normal operating mode. We use the notation Q_n to represent the current value (i.e., current state) and Q_{n+1} to represent the next value (i.e., next state).

Let us analyze the truth table. Consider first the two straightforward cases. When S = 0 and R = 1, we can see that independent of the current state, output Q is forced to be 0 as R is 1. Thus, the two inputs to the upper NOR gate are 0. This leads $\overline{Q}$ to be 1. This is a stable state. That is, Q and $\overline{Q}$ can stay at 0 and 1, respectively. You can verify that when S = 1 and R = 0, another stable state Q = 1 and $\overline{Q}$ = 0 results.

When both S and R are zero, the next output depends on the current output. Assume that the current output is Q = 1 and $\overline{Q}$ = 0. Thus, when you change inputs from S = 1 and R = 0 to S = R = 0, the next state Q_{n+1} remains the same as the current state Q_n. Now assume that the current state is Q = 0 and $\overline{Q}$ = 1. It is straightforward to verify that changing inputs from S = 0 and R = 1 to S = R = 0, leaves the output unchanged. We have summarized this behavior by placing Q_n as the output for S = R = 0 in the first row of Figure 4.5*c*.

What happens when both S and R are 1? As long as these two inputs are held high, both outputs are forced to take 0. We struck this state from the truth table to indicate that this input combination is undesirable. To see why this is the case, consider what happens when S and R inputs are changed from S = R = 1 to S = R = 0. It is only in theory that we can assume that both inputs change simultaneously. In practice, there is always some finite time difference between the two signal changes. If the S input goes low earlier than the R signal, the sequence of input changes is SR = 11 $\rightarrow$ 01 $\rightarrow$ 00. Because of the intermediate state SR = 01, the output will be Q = 0 and $\overline{Q}$ = 1.

If, on the other hand, the R signal goes low before the S signal does, the sequence of input changes is SR = 11 $\rightarrow$ 10 $\rightarrow$ 00. Because the transition goes through the SR = 10 intermediate



(a) Circuit diagram (b) Logic symbol

Figure 4.6 Clocked SR latch.

state, the output will be Q = 1 and $\overline{Q}$ = 0. Thus, when the input changes from 11 to 00, the output is indeterminate. This is the reason we want to avoid this state.

The inputs S and R stand for "Set" and "Reset," respectively. When the set input is high (and reset is low), Q is set (i.e., Q = 1). On the other hand, if set is 0 and reset is 1, Q is reset or cleared (i.e., Q = 0).

From this discussion, it is clear that this latch is level sensitive. The outputs respond to changes in input levels. This is true for all the latches.

We notice that this simple latch has the capability to store a bit. To write 1 into this latch, set SR as 10; to write 0, use SR = 01. To retain a stored bit, keep both S and R inputs at 0. In summary, we have the capacity to write 0 or 1 and retain it as long as there is power to the circuit. This is the basic 1-bit cell that static RAMs use. Once we have the design to store a single bit, we can replicate this circuit to store wider data as well as multiple words. We look at some of these design issues in Chapter 16.

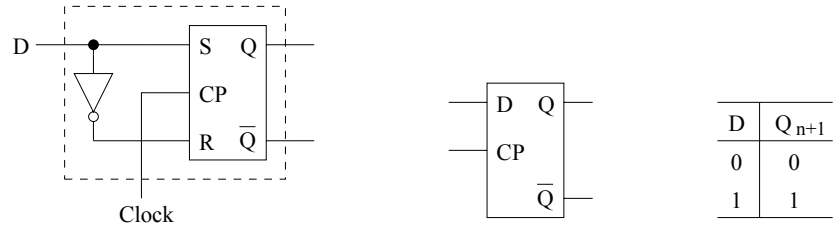
4.3.2 Clocked SR Latch

A basic problem with the SR latch is that the output follows the changes in the input. If we want to make the output respond to changes in the input at specific instants in order to synchronize with the rest of the system, we have to modify the circuit as shown in Figure 4.6*a*. The main change is that a clock input is used to gate the S and R inputs. These inputs are passed onto the original SR latch only when the clock signal is high. The inputs have no effect on the output when the clock signal is low. When the clock signal is high, the circuit implements the truth table of the SR latch given in Figure 4.5*c*.

This latch is level sensitive as well. As long as the clock signal is high, the output responds to the SR inputs. We show in the next section that we can design flip-flops that are edge-triggered.

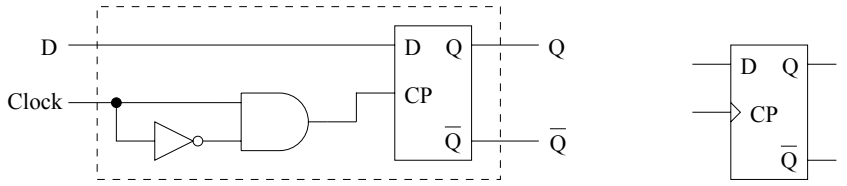
4.3.3 D Latch

A problem with both versions of SR latches is that we have to avoid the SR = 11 input combination. This problem is solved by the D latch shown in Figure 4.7*a*. We use a single inverter



(a) Circuit diagram (b) Logic symbol (c) Truth table

Figure 4.7 D latch uses an inverter to avoid the SR = 11 input combination.



(a) Circuit diagram (b) Logic symbol

Figure 4.8 Positive edge-triggered D flip-flop.

to provide only complementary inputs at S and R inputs of the clocked SR latch. To retain the value, we maintain the clock input at 0. The logic symbol and the truth table for the D latch clearly show that it can store a single bit.

4.4 Flip-Flops

We have noted that flip-flops are edge-triggered devices whereas latches are level sensitive. Unfortunately, even some manufacturers do not follow this distinction. Several books on digital logic also propagate this confusion. We strictly follow the distinction between latches and flip-flops. In this section, we look at two flip-flops: the D flip-flop and the JK flip-flop. We also show a logic symbol notation to clearly distinguish flip-flops from latches.

4.4.1 D Flip-Flops

We show how one can convert the level-sensitive D latch of Figure 4.7 into an edge-triggered D flip-flop. The necessary clue comes from Figure 4.4. We can see from this figure that the output of this simple circuit is a small pulse, equal to the propagation delay of the inverter (typically 10 ns), at the rising edge of the input clock signal. Thus, if we feed this signal instead of the original clock, we convert our D latch into a rising or positive edge-triggered D flip-flop.

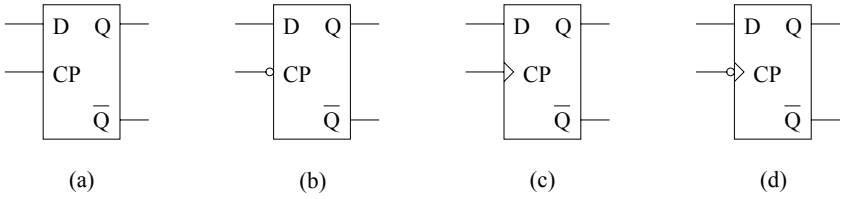


Figure 4.9 Logic symbol notation for latches and flip-flops: (a) high level-sensitive latch; (b) low level-sensitive latch; (c) positive edge-triggered flip-flop; (d) negative edge-triggered flip-flop.

In the logic symbol, we use an arrowhead on the clock input to indicate a positive edge-triggered flip-flop as shown in Figure 4.8*b*. The absence of this arrowhead indicates a high level-sensitive latch (see Figure 4.9*a*). We add a bubble in front of the clock input to indicate a negative edge-triggered flip-flop (Figure 4.9*d*) or a low level-sensitive latch (Figure 4.9*b*).

As is obvious from the bubble notation, we can convert a high level-sensitive latch to a low level-sensitive one by feeding the clock signal through an inverter. Recall that the bubble represents an inverter (see page 76). Similarly, we can invert the clock signal to change a negative edge-triggered flip-flop to a positive edge-triggered one.

4.4.2 JK Flip-Flops

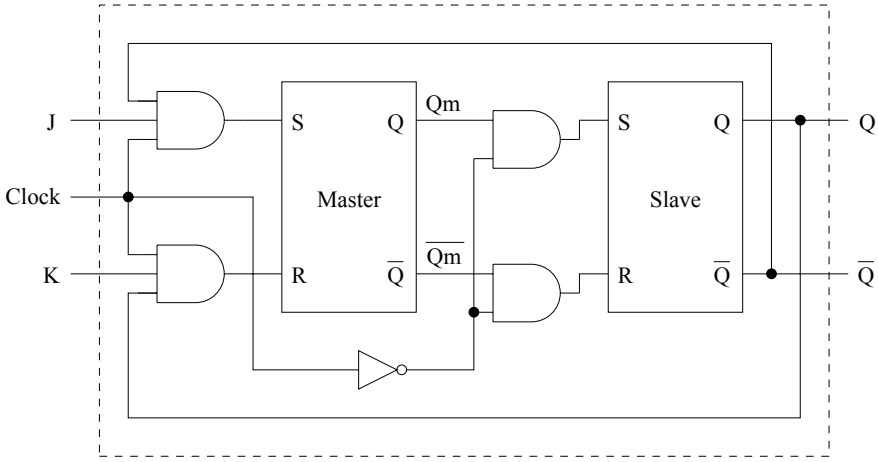
We can design an edge-triggered flip-flop by using a master–slave design as shown in Figure 4.10*a*. This design uses two SR latches. The basic idea is to activate the master SR latch during the clock ON period and transfer the output of the master latch to the final output during the clock OFF period. The circuit shown in Figure 4.10*a* is a negative edge-triggered flip-flop. The logic symbol shown in Figure 4.10*b* follows our notation to indicate this fact.

The feedback connections from the slave SR latch output to the JK input AND gates transform the SR latch into the JK latch to avoid indeterminacy associated with 11-input combinations in the SR latch.

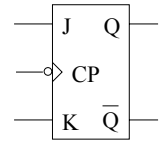
To illustrate the working of this flip-flop, we have shown the timing diagram (Figure 4.10*c*) that covers all six possibilities: JK = 01, 10, JK = 11 with $\mathrm{Q_n}$ = 0 and 1, and JK = 00 with $\mathrm{Q_n}$ = 0 and 1. For clarity, we have not included $\overline{\mathrm{Q}}$ and $\overline{\mathrm{Qm}}$ in Figure 4.10*c* as these two signals are complementary to Q and Qm signals. We can see from this timing diagram that output changes only at falling edges of the clock signal. You can also verify that changes to J and K inputs during other times do not affect the output.

The JK flip-flop truth table is shown in Table 4.1. Unlike the SR latch, the JK flip-flop allows all four input combinations. When JK = 11, the output toggles. This characteristic is used to build counters, as we show in the next section.

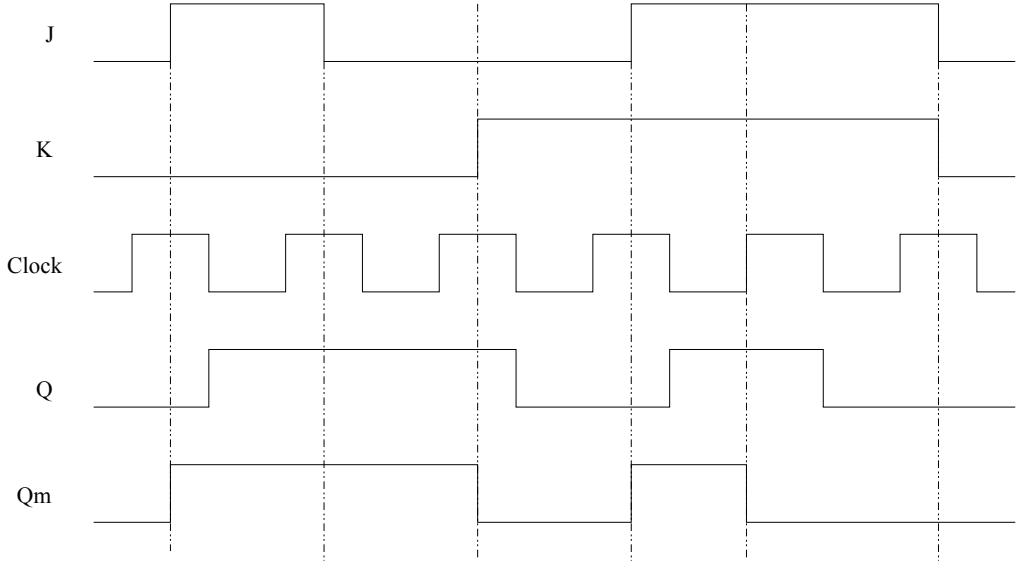
For proper operation, the JK inputs should be applied at least $\mathrm{t_s}$ time units before the falling edge of the clock. The time $\mathrm{t_s}$ is referred to as the *setup time*. A typical value for $\mathrm{t_s}$ is about 20 ns for the 7476 chip, which is described next.



(a) Circuit diagram



(b) Logic symbol



(c) Timing diagram

Figure 4.10 JK master–slave flip-flop.

Table 4.1 JK flip-flop truth table

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q}_n$

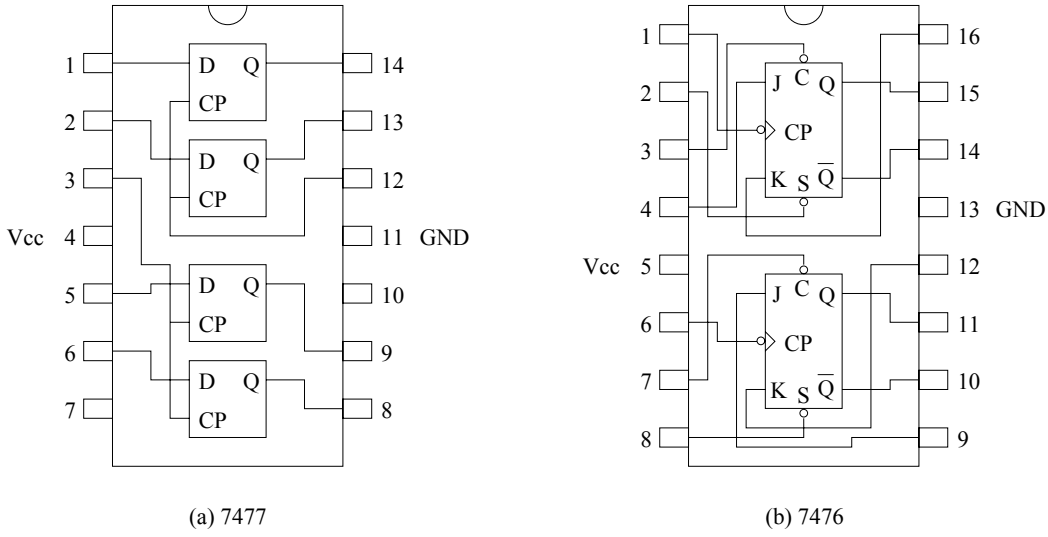


Figure 4.11 Two example chips.

4.4.3 Example Chips

Several commercial latch and flip-flop chips are available. As with the basic gates we have seen in Figure 2.8 on page 50, several units are packaged into a single chip. Figure 4.11 shows two example chips.

The 7477 chip uses a 14-pin DIP package and provides four D latches, grouped into two sets. Each set of two D latches shares their clock signal. The logic symbol indicates that this is a high level-sensitive D latch. Such latches are useful in providing temporary storage. For example, if we want to store an 8-bit datum from a peripheral device such as a printer, we can use two 7477 chips in parallel to construct an 8-bit latch. All we have to do is tie the four clock input signals together to have a single clock that latches the 8-bit datum into the latch.

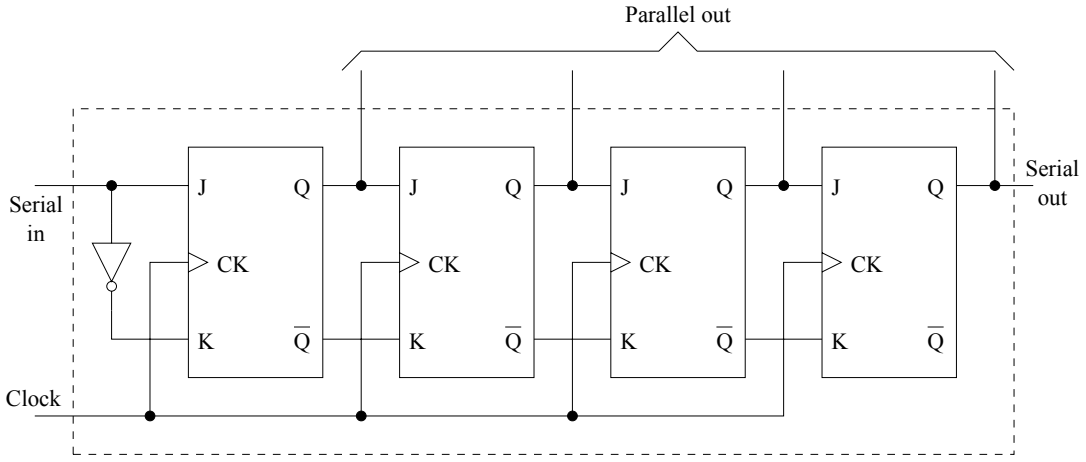


Figure 4.12 A 4-bit shift register using JK flip-flops.

As the 7477 is level sensitive, the Q output follows the changes in the corresponding D input while the clock signal is high. In practice, when the data are ready to be stored, a short pulse is applied to the clock input.

The 7476 chip provides two completely independent JK flip-flops. From the logic symbol we can observe that this is a negative edge-triggered flip-flop. The 7476 JK flip-flop also provides set (S) and clear (C) inputs. The set input, when active, sets the Q output to 1. An active clear input clears the Q output (i.e., Q = 0). Because of the bubble associated with both S and C inputs, at the chip level, both are low-active signals. For example, when a low level is applied to pin 2, Q output of the top flip-flop (pin 15) will be set (Q = 1). Holding both S and C inputs at low level results in an indeterminate state (similar to the 11-input combination scenario for the SR latch of Figure 4.5) and should be avoided. We show some sample applications for the flip-flops in the next section.

4.5 Example Sequential Circuits

4.5.1 Shift Registers

Shift registers, as the name suggests, shift data left or right with each clock pulse. Designing a shift register is relatively straightforward as shown in Figure 4.12. This shift register, built with positive edge-triggered JK flip-flops, shifts data to the right. For the first JK flip-flop, we need an inverter so that the K input is the complement of the data coming in ("Serial in" input). The data out, taken from the Q output of the rightmost JK flip-flop, is a copy of the input serial signal except that this signal is delayed by four clock periods. This is one of the uses of the shift registers.

We can also use a shift register for serial-to-parallel conversion. For example, a serial signal, given as input to the shift register in Figure 4.12, produces a parallel 4-bit output (taken from the four Q outputs of the JK flip-flops) as shown in Figure 4.12. Even though we have not shown it here, we can design a shift register that accepts input in parallel (i.e., parallel load) as well as serial form. Shift registers are also useful in implementing logical bit shift operations in the ALU of a processor.

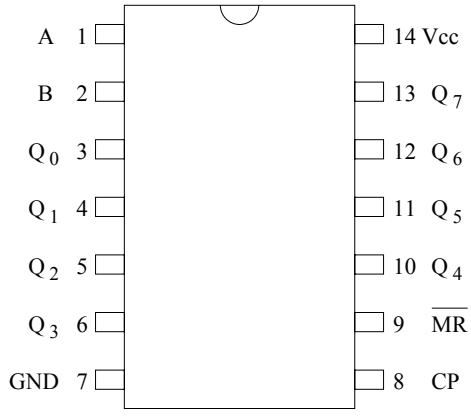
Although we can build shift registers using flip-flops, several shift register chips are commercially available. Figure 4.13 shows an 8-bit serial-in, parallel-out shift register. This shift register is implemented using D flip-flops. Notice that the clock signal applied to pin 8 goes through an internal inverter. Thus, even though the D flip-flops are negative edge-triggered, the shift register itself is positive edge-triggered at the pin level. This shift register also has a master reset ($\overline{\text{MR}}$) signal that clears all eight outputs Q0 through Q7. The AND gate at the input side allows us to use one of the two inputs (A or B) as an enable signal and the other for the data input.

4.5.2 Counters

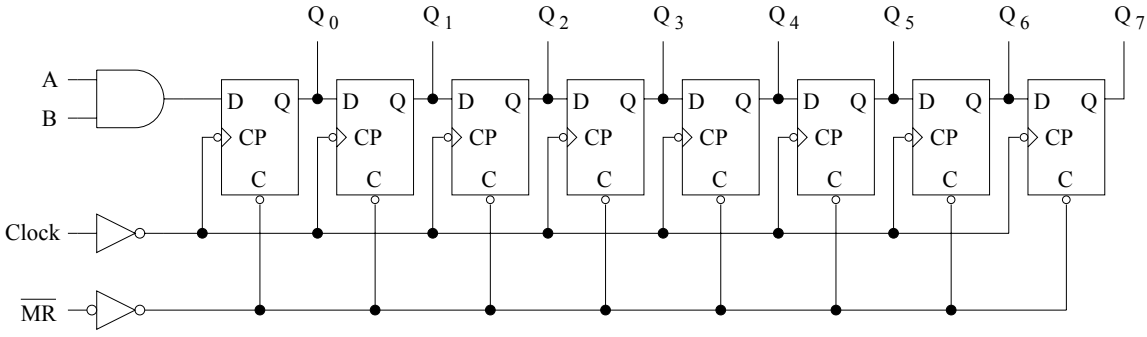
A counter is another example of a sequential circuit that is often used in digital circuit design. To see how we can build a counter, let us consider the simplest of all counters: the binary counter. A binary counter with B bits can count from 0 to $2^B - 1$. For example, a binary counter with $B = 3$ can count from 0 to 7. After counting eight (with a count value of 7), the count value wraps around to zero. Such a counter is called a modulo-8 counter. Another counter that is useful with the decimal number system is the modulo-10 counter. Because designing binary counters is simple, we first focus on the design of a modulo-8 counter.

We know that a modulo-8 counter requires 3 bits to represent the count value. In general, a modulo-2^B counter requires B bits (i.e., $\log_2 2^B$ bits). To develop our intuition, it is helpful to look at the values 0 through 7, written in the binary form in that sequence. If you look at the rightmost bit, you will notice that it changes with every count. The middle bit changes whenever the rightmost bit changes from 1 to 0. The leftmost bit changes whenever the middle bit changes from 1 to 0. These observations can be generalized to counters that use more bits. There is a simple rule that governs the counter behavior: a bit changes (flips) its value whenever its immediately preceding right bit goes from 1 to 0. This observation gives the necessary clue to design our counter. Suppose we have a negative edge-triggered JK flip-flop. We know that this flip-flop changes its output with every negative edge on the clock input, provided we hold both J and K inputs high. Well, that is the final design of our 3-bit counter as shown in Figure 4.14.

We operate the JK flip-flops in the "toggle" mode with JK = 11. The Q output of one flip-flop is connected as the clock input of the next flip-flop. The input clock, which drives our counter, is applied to FF0. When we write the counter output as $\text{Q}_2\text{Q}_1\text{Q}_0$, the count value represents the number of clock negative edges. For example, the dotted line in Figure 4.14*b* represents $\text{Q}_2\text{Q}_1\text{Q}_0$ = 011. This value matches the number of falling edges to the left of the dotted line in the input clock.



(a) Connection diagram

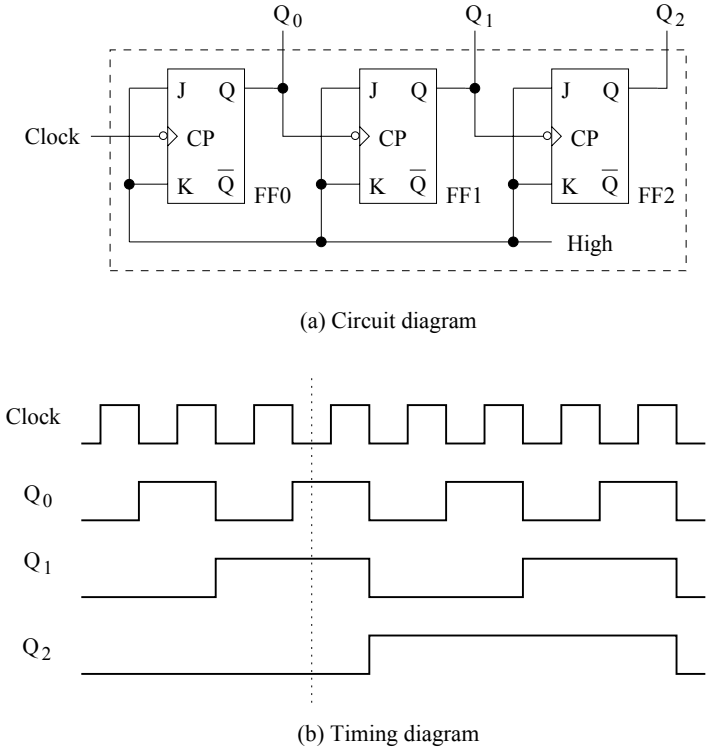


(b) Logic diagram

Figure 4.13 Details of the 74164 shift register.

Counters are also useful in generating clocks with different frequencies by dividing the input clock. For example, the frequency of the clock signal at Q_0 output is half of the input clock. Similarly, frequencies of the signals at Q_1 and Q_2 are one-fourth and one-eighth of the counter input clock frequency.

The counter design shown in Figure 4.14 is called a *ripple counter* as the count bits ripple from the rightmost to the leftmost bit (i.e., in our example, from FF0 to FF2). A major problem with ripple counters is that they take a long time to propagate the count value. To give a concrete example, suppose that we are using the JK flip-flops provided by the 7476 chip to build a 16-bit binary counter. The data sheet from Motorola for a version of this chip (74LS76A) states that the maximum delay from clock to output is 20 ns. That means it takes $16 \times 20 = 320$ ns. Thus,



(a) Circuit diagram

(b) Timing diagram

Figure 4.14 A binary ripple counter implementation using negative edge-triggered JK flip-flop.

a system that uses this 16-bit counter cannot operate faster than about 1/320 ns $\approx$ 3 MHz. We have had a similar discussion about ripple carry adders in Section 3.5 on page 95.

How can we speed up the operation of the ripple binary counters? We apply the same trick that we used to derive the carry lookahead adder in Section 3.5. We can design a counter in which all output bits change more or less at the same time. These are called *synchronous counters*. We can obtain a synchronous counter by manipulating the clock input to each flip-flop. We observe from the timing diagram in Figure 4.14*b* that a clock input should be applied to a flip-flop if all the previous bits are 1. For example, a clock input should be applied to FF1 whenever the output of FF0 is 1. Similarly, a clock input for FF2 should be applied when the outputs of FF0 and FF1 are both 1. A synchronous counter based on this observation is shown in Figure 4.15. The maximum delay in this circuit is one flip-flop delay (about 20 ns) and an AND gate delay (about 10 ns). Thus, our synchronous counter can operate at 1/30 ns $\approx$ 33 MHz, an 11-fold increase from the ripple counter design.

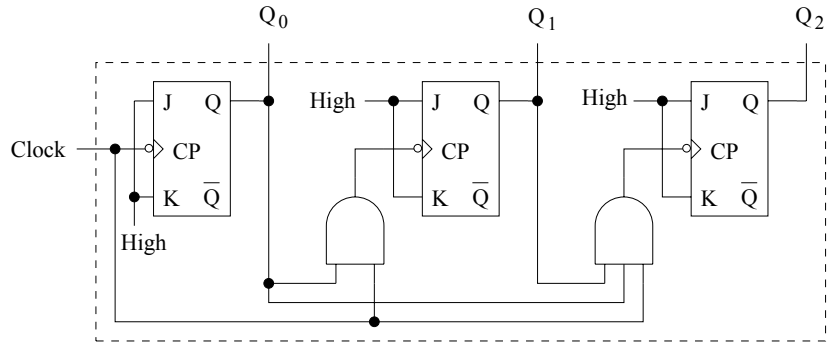


Figure 4.15 A synchronous modulo-8 counter.

Table 4.2 Function table for the counter chips

$\overline{\text{MR}}$	$\overline{\text{PE}}$	CET	CEP	Action on clock rising edge
L	X	X	X	Clear
H	L	X	X	Parallel load ($P_n \rightarrow Q_n$)
H	H	H	H	Count (increment)
H	H	L	X	No change (hold); TC is low
H	H	X	L	No change (hold)

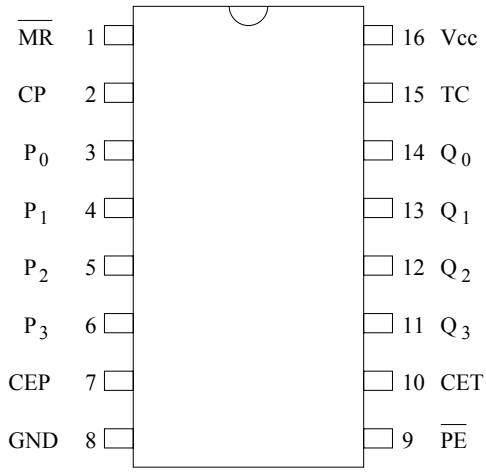
H = High voltage level; L = Low voltage level; X = Don't care.

Some Counter Chips

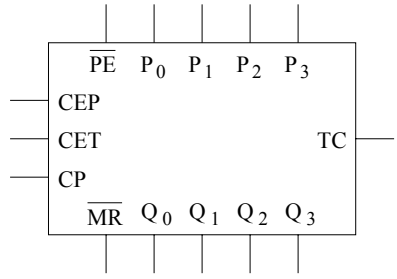
We present two example counter chips: the 74161 and 74160. The 74161 chip is a binary modulo-16 synchronous counter that can operate at the 120 MHz count frequency. The other is a modulo-10 synchronous counter. Modulo-10 counters are also called *decade counters*. Both chips use the same connection layout and logic symbol as shown in Figure 4.16.

The 74161 is a 4-bit binary counter that cycles through 0 to 15 (Figure 4.16*c*). On the other hand, the 74160 can be used in applications that require counting in the decimal system. As shown in Figure 4.16*d*, the count value goes from 9 to 0 to implement the modulo-10 function.

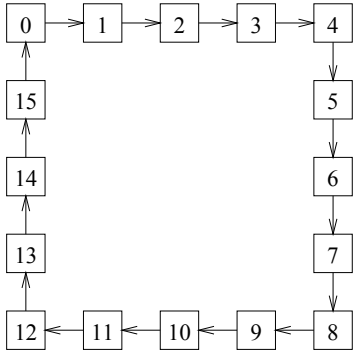
These chips provide a way to initialize the counter. The $\overline{\text{MR}}$, when held low, will clear the output (equivalent to making the count value 0). We can also initialize the counter to a preset value other than zero by using the preset enable ($\overline{\text{PE}}$) signal. As shown in the second row of



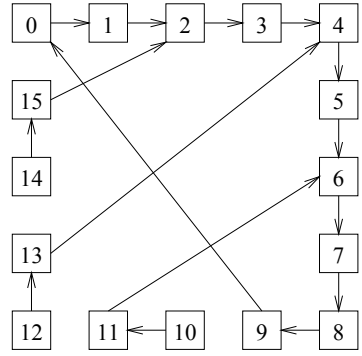
(a) Connection diagram



(b) Logic symbol



(c) State diagram of 74161



(d) State diagram of 74160

Figure 4.16 Two synchronous counter chips: 74161 is a modulo-16 binary counter and 74160 is a modulo-10 decade counter.

the function table (Table 4.2), we can load the four P inputs to the corresponding outputs by applying a low level to $\overline{\text{PE}}$ while holding $\overline{\text{MR}}$ high. The other two clock enable inputs CET and CEP should be high in order for the counter to count. If either of these two signals is low, the counter clock is disabled and the current count holds.

The terminal count (TC) output is high when the CET is high and the counter is in its maximum count state (15 for the binary counter and 9 for the decade counter). The TC output

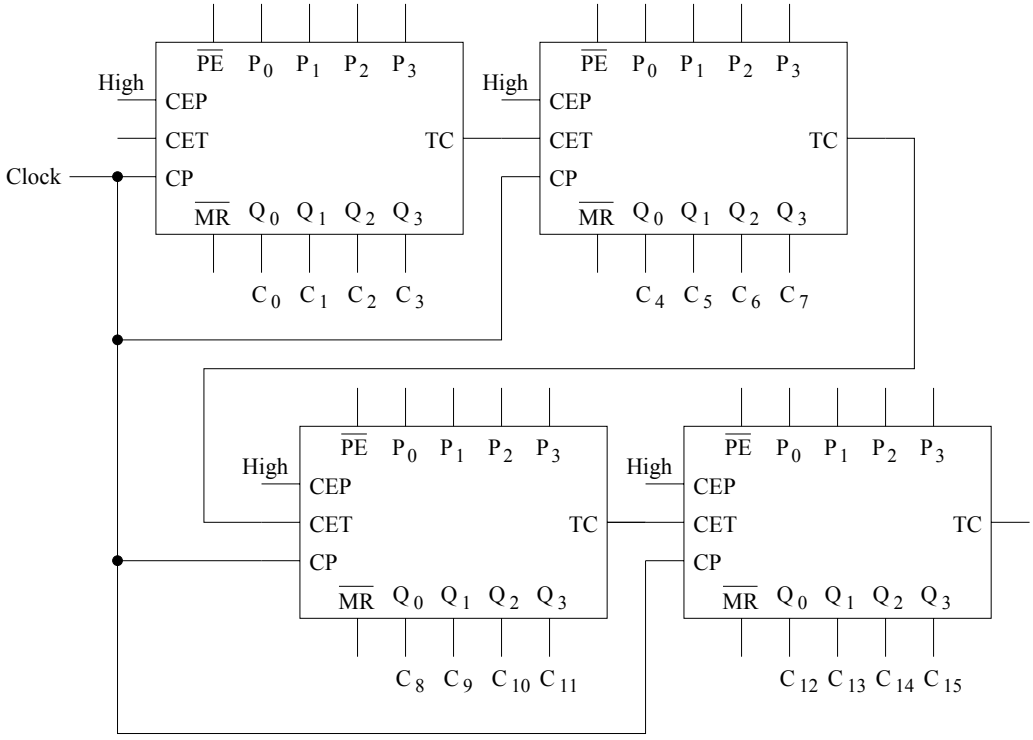


Figure 4.17 A 16-bit multistage counter using four 4-bit synchronous counters.

can also be high for the decade counter chip when the counter is preset to one of the illegal states 11, 13, or 15. The decade counter could also be in an illegal state when power is applied. In any case, the counter returns to a legal counting sequence within two counts as shown in Figure 4.16*d*.

The TC output signal is useful in implementing synchronous multistage counters. We can use the TC output of a previous stage to enable the clock input of the next chip as shown in Figure 4.17. Using four 74161 chips, we can build a 16-bit binary counter. The data sheet guarantees maximum delay of 15 ns from clock CP to TC output. Since the stages are connected in ripple counter fashion, the total delay is $4 \times 15 = 60$ ns. This gives us a guaranteed operating frequency for our 16-bit counter as 1/60 ns $\approx$ 16.7 MHz. Note that this is the guaranteed minimum value. The typical value will be higher than this value. The data sheet discusses a better way to implement a multistage synchronous counter using the CET and CEP inputs.

4.6 Sequential Circuit Design

We now have the necessary background and intuition to start working on the sequential circuit design process. To illustrate the concepts we focus on sequential circuits using JK flip-flops. Recall that a sequential circuit consists of a combinational circuit that produces output and a feedback circuit for state variable feedback (Figure 4.1). We use JK flip-flops for the feedback circuit design. The design process, however, can be generalized to other devices such as SR latches.

In the next two subsections, we consider designing counters using JK flip-flops. The first counter example is the 3-bit binary counter we have discussed in the last section, except that we derive a synchronous design. We use this example to introduce the design process. The following section describes how we can use this process to design a general counter.

Counters are a special case of more general sequential circuits. Section 4.6.2 describes the design process for general sequential circuits.

4.6.1 Binary Counter Design with JK Flip-Flops

We start our discussion with a simple 3-bit synchronous counter design example. In analyzing a sequential circuit, we go from input to output. For example, we might know JK inputs and we want to know the next state outputs. We use the JK flip-flop truth table for this purpose.

In designing a sequential circuit, we know the output state and we have to find the input combination. For this purpose, we build an excitation table, which can be derived from the truth table. Table 4.3*a* shows the truth table for the JK flip-flop. In this table, we have explicitly included the present output Q_n. The excitation table consists of Q_n and Q_{n+1} as well as J and K inputs as shown in Table 4.3*b*. In this table, "d" indicates a don't care input. For example, to change the output from 0 to 1, the J input has to be 1, but the K input doesn't matter (second row in Table 4.3*b*). We use this excitation table extensively in designing sequential circuits in the remainder of this chapter.

We can now write the combination of JK inputs that take the circuit from the current state to the next state. To facilitate this process, we create a new table with several groups of columns (Table 4.4). The first group is the current state output, the second group is the next state output, and the last group consists of the JK inputs for each flip-flop in the circuit. For our 3-bit counter example, we have three bits A B C to represent the current state output and another three bits to represent the values of the next state as shown in Table 4.4. Since each bit is stored in a JK flip-flop, we have three pairs of JK inputs corresponding to the three flip-flops A, B, and C.

Filling the entries of this table is straightforward. In our example, the next state is given by (current state + 1) modulo 8. Then we will fill the JK inputs. To do this, look at the corresponding bits in the current state and next state and write down the JK input combination from the excitation table that gives the required transition from Q_n to Q_{n+1}. For example, for the first row, current state and next state for the A bit is 0. Thus, we write $J_A = 0$ and $K_A = d$ for flip-flop A. As another example, in the last row, the A bit changes from 1 to 0. Thus, JK inputs will have to be d1 (third row of the excitation table). This is the entry you see in the last row for the JK inputs of the A flip-flop.

Table 4.3 Excitation table derivation for the JK flip-flops

(a) JK flip-flop truth table

J	K	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(b) Excitation table for JK flip-flops

Q_n	Q_{n+1}	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

Table 4.4 Design table for the binary counter example

Present state			Next state			JK flip-flop inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	d	0	d	1	d
0	0	1	0	1	0	0	d	1	d	d	1
0	1	0	0	1	1	0	d	d	0	1	d
0	1	1	1	0	0	1	d	d	1	d	1
1	0	0	1	0	1	d	0	0	d	1	d
1	0	1	1	1	0	d	0	1	d	d	1
1	1	0	1	1	1	d	0	d	0	1	d
1	1	1	0	0	0	d	1	d	1	d	1

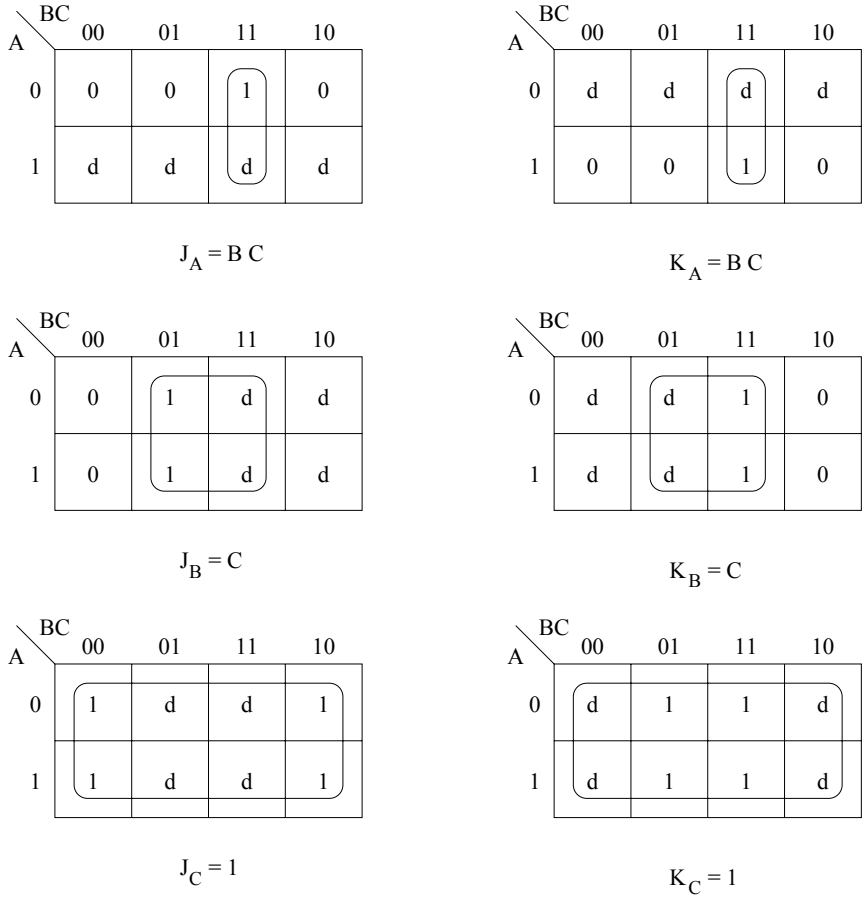


Figure 4.18 Karnaugh maps to derive simplified logic expressions for the JK inputs.

Once we have identified the JK inputs for each flip-flop, all we have to do is to see if we can simplify the logical expression for each input. This step is similar to the logical expression simplification we have done in combinational circuit design. We can use any of the methods discussed in Chapter 2. Here, we use the Karnaugh map method to simplify the logical expressions for the six inputs of the three JK flip-flops. Figure 4.18 shows the six Karnaugh maps for the JK inputs. Note that the cells in these Karnaugh maps represent the present state values for A, B, and C.

As a final step, we write the logic circuit diagram based on these logical expressions. In addition to the three JK flip-flops, we just need a 2-input AND gate to implement the 3-bit synchronous counter as shown in Figure 4.19. Contrast this design with the synchronous counter design shown in Figure 4.15. The main difference is that our new design uses the common clock

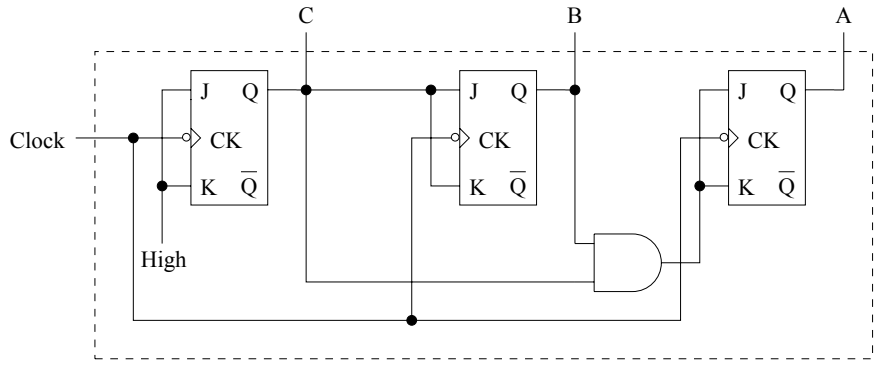


Figure 4.19 Another 3-bit synchronous counter design.

while manipulating the JK inputs. The previous design, on the other hand, manipulates the clock inputs while holding JK inputs high.

A More General Counter Design Example

The previous counter example is simple in the sense that the next state is always an increment of the current state and there is a natural wraparound when the count reaches the maximum value. We know that counters can behave differently. An example we have seen before is the decade counter. In this counter, the next state after 9 is 0. How do we design such counters? We go even a step further and show a counter that jumps states in no specific order. Once we show how such a general counter can be designed, decade counter design becomes a special case of this design process.

For our example, we consider designing a 3-bit synchronous counter with the state transitions shown in Figure 4.20. The counter goes through the following states in a cycle:

$$0 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 0.$$

This example counter is different from our binary counter example in certain respects. First, not all states are present. This is similar to the decade counter example, which skips states 10 through 15. The other feature is that next state seems to be an arbitrary state (i.e., not a state that can be obtained by incrementing the current state).

Despite these differences, the design process we have used for the binary counter example applies. The design table to derive the JK inputs is shown in Table 4.5. One significant difference from the table used for the previous counter example and this is the presence of don't care values in the next state columns for states 1, 2, and 4 (shown by a dash in Table 4.5). Because the next state for these three states is a don't care, all JK inputs also assume don't care inputs. This would help us in simplifying the logical expressions for the JK inputs.

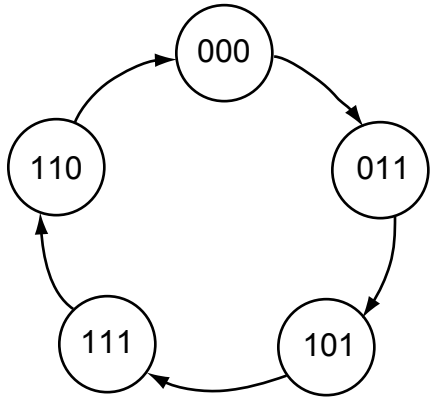


Figure 4.20 State diagram for the general counter example.

Table 4.5 Design table for the general counter example

Present state			Next state			JK flip-flop inputs					
A	B	C	A	B	C	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	1	0	d	1	d	1	d
0	0	1	–	–	–	d	d	d	d	d	d
0	1	0	–	–	–	d	d	d	d	d	d
0	1	1	1	0	1	1	d	d	1	d	0
1	0	0	–	–	–	d	d	d	d	d	d
1	0	1	1	1	1	d	0	1	d	d	0
1	1	0	0	0	0	d	1	d	1	0	d
1	1	1	1	1	0	d	0	d	0	d	1

Figure 4.21 shows the Karnaugh maps to derive the simplified logical expressions for the JK inputs. The final logical circuit for this counter is shown in Figure 4.22.

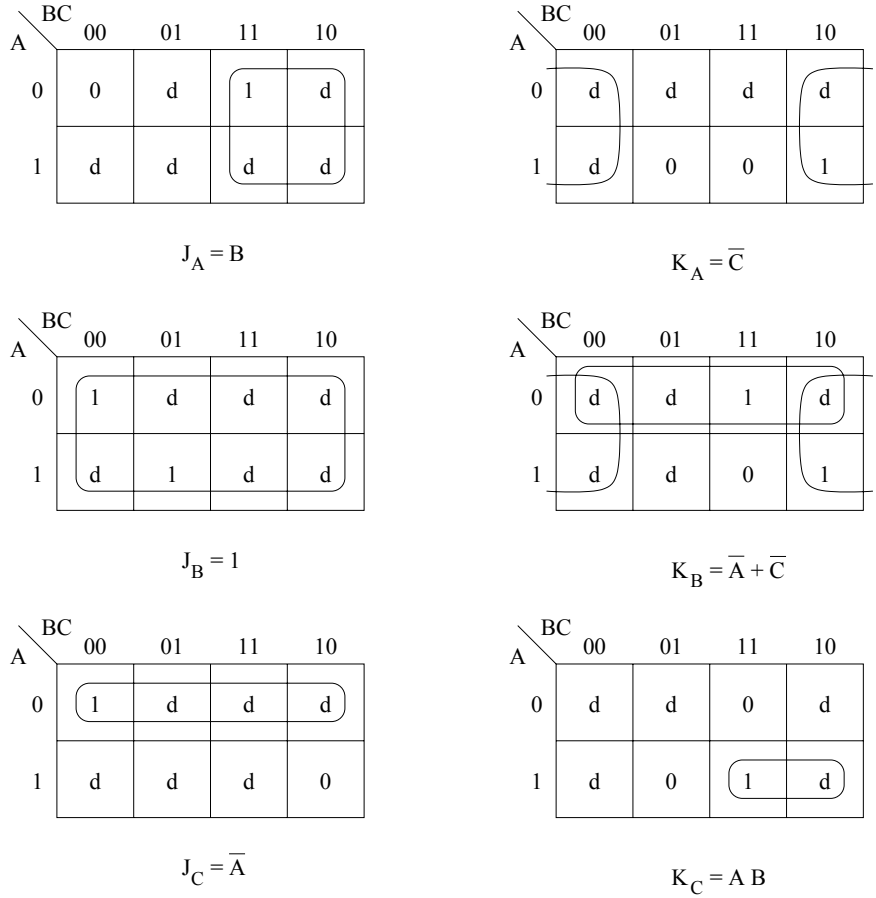


Figure 4.21 Derivation of JK logical expressions for the general counter example.

4.6.2 General Design Process

The counter design example discussed in the last section is a special case of the more general sequential circuit design process. A finite state machine can express the behavior of a sequential circuit. A finite state machine consists of a set of (finite number of) states that the system can take. State transitions are indicated by arrows with labels X/Y, where X represents the inputs that cause the change in the system state and Y represents the output generated while moving from the current state to the next state. Note that X and Y could represent multiple bits.

Example 4.1: *Even-parity checker.* Parity is used to provide rudimentary error detection capability. For example, we can associate a parity bit for each 7-bit ASCII character transmitted. Even parity means that the total number of 1's in the 8-bit group (7 data bits + 1 parity bit)

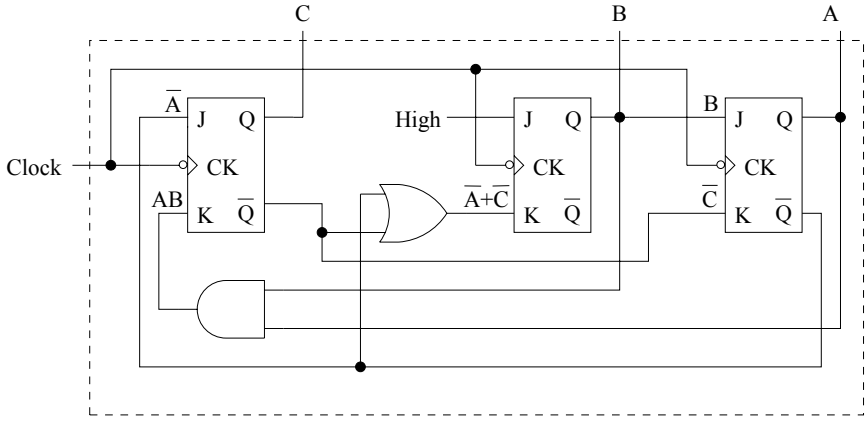


Figure 4.22 Logic circuit for the general counter example.

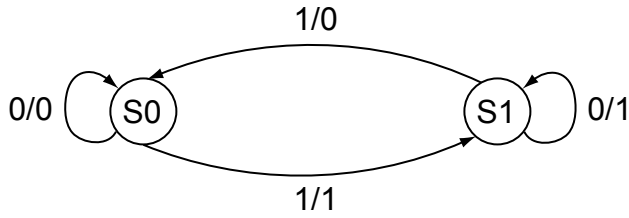


Figure 4.23 State diagram for the even-parity checker.

should be an even number. The receiver of the data also computes the parity and checks if the data have been received properly. For a properly received 8-bit datum, the computed parity bit should be 0. Parity provides single-bit error detection capability.

We have seen in Section 2.10.2 (see Figure 2.27 on page 77) that we can use XOR gates to generate even parity. However, this circuit is not useful for us if the data are coming serially, one bit at a time as on a modem line. If we want to use the XOR implementation, we need to convert the serial input data to parallel form. The other alternative is to design a sequential circuit that receives data in serial form and generates the parity bit.

A simple analysis leads us to the conclusion that the FSM needs to remember only one of two facts summarizing the past input sequence: whether the number of 1's is odd or even. Thus, our FSM needs just two states as shown in Figure 4.23. In the FSM, state S0 represents the fact that the input sequence so far has an even number of 1's. The odd number of 1's is represented by state S1.

Now we have to look at the possible transitions between these two states. When the machine is in state S0, if a 1 is received, the machine should move to S1. Also, it should output a 1 as

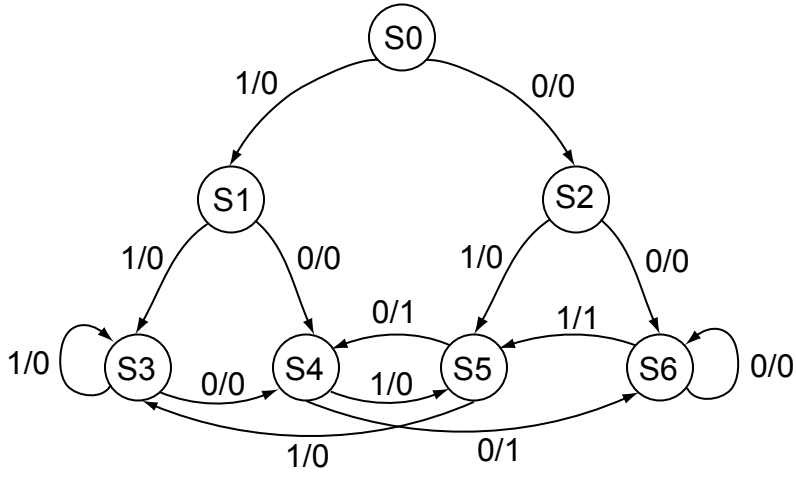


Figure 4.24 State diagram for the pattern recognition problem.

the parity bit for the data it has received so far. This transition is shown in the figure with label 1/1 to represent the fact that the input is 1 and the output is also 1. On the other hand, if a 0 is received in state S0, it remains in that state with output 0.

Similarly, when in state S1, if a 0 is received as input, it remains in state S1 with a 1 as output because the number of 1's in the input so far is odd. A 1 input takes the machine from S1 to S0 with 0 output.

If we want to carry this design forward, we need a single flip-flop to represent the two states in the FSM. We leave the complete design of this machine as an exercise to be done after reading this section (see Exercise 4–13). We show the design process on the next example, which is more complex.

Example 4.2: *Pattern recognition.* As another example, consider designing a system for recognizing a specific bit pattern in the input bit sequence, which enters the system serially. The particular system we wish to design outputs a 1 whenever the input bit sequence has exactly two 0s in the last three input bits.

The system maintains a memory of the last two bits and looks at the next bit coming in to determine the output. For example, the input sequence 010100101001000 produces 101111011111000 as the output sequence. In this example, we assume that the rightmost bit is the first bit to enter the system.

The FSM for this example is shown in Figure 4.24. State S0 represents the initial state (i.e., no input has been received yet). From this state, whether the input is 0 or 1, the output is 0. However, we need to remember the first input bit. Thus, we create two states S1 and S2. From both S1 and S2, whatever the input is, the output is 0 as we have seen only two bits so far. Notice that we visit these three states—S0, S1, and S2—only once during the initial phase.

After receiving the first two input bits, we can be in one of the four states—S3, S4, S5, or S6—depending on the values of these two bits. For example, we will be in S6 if the first two bits of the input sequence are 00. If the input is 01, we will be in state S4. We can summarize the state information captured by these four states as follows:

S3: Last two bits are 11;
S4: Last two bits are 01;
S5: Last two bits are 10;
S6: Last two bits are 00.

Having this summary information helps us in adding the transitions to the FSM among these states. For example, consider state S3. That means the last two bits are 11. Suppose the next bit is 0. The last three bits have two 1's and a 0. Therefore, the output should be 0. And what should be the new state after receiving 0? Since the last two bits are 01, the new state should be S4 as shown in Figure 4.24.

What if, when in state S3, the next bit is 1? In this case, the last three bits are 111 and hence the output should be 0. Next state is the state that represents 11, which is S3 itself. This is shown as a loop with a 1/0 label in the state diagram.

You can apply this method to complete all the transitions among the four states of the FSM that represents the pattern recognition problem.

Steps in the Design Process

Our design process consists of the following steps:

1. *Derive FSM:* From the problem specification, derive a finite state machine representation of the problem. Make sure that the FSM truly and accurately represents the problem specification.
2. *State Assignment:* Assign binary states of the flip-flops to the FSM states. This step is necessary to get an efficient implementation. This problem did not arise in our counter design examples.
3. *Design Table Derivation:* Derive a design table for the state assignment selected in the last step. This step is similar to the design tables used in the design of counters. However, in a general sequential circuit, both input and output may be present.
4. *Logical Expression Derivation:* Derive logical expressions for the JK inputs and the logical expression for the combinational circuit to generate the output. This step is similar to the process described in the last section.
5. *Implementation:* As a final step, implement the logical expressions obtained in the last step.

Since we already derived the FSM for our example, we look at the remaining steps next.

State Assignment

The state diagram for the pattern recognition problem (Figure 4.24) consists of seven states—S0 through S6. That means we need three flip-flops to keep the state information. In general, the number of flip-flops required is $\lceil \log_2 N \rceil$, where N is the number of states in the state diagram. The problem we have now is: How do we assign the state diagram states to the flip-flop states? Of course, we can randomly assign a bit pattern for each state or use a sequence based on the state number. For example, we could assign 000 for S0, 001 for S1, and so on. The problem is that such a state assignment might not lead to an efficient design in that the logic required to drive the JK inputs as well as the logic required to generate the output might not be minimal.

To obtain a good design, we preset three heuristics that help us select a good state assignment. We consider two states adjacent if the number of bit positions in which they differ is exactly one. For example, 000 and 010 are adjacent whereas 010 and 001 are not. If you are familiar with Hamming distance, adjacent states are states with a Hamming distance of one. The three heuristics are summarized below:

1. States that have the same next state for a given input should be assigned adjacent states.
2. States that are the next states of the same state should be assigned adjacent states.
3. States that have the same output for a given input should be assigned adjacent states.

Why do these heuristics make sense? What we are suggesting by these heuristics is that there can be only one variable change between adjacent states. The reason is that these states will end up as neighbors in the Karnaugh map we use later (as we did with the counter examples) to simplify the logical expressions. Having them together simplifies our logical expressions for the JK inputs and the output.

Heuristic 1 says that all input states of a state should be assigned adjacent states so that they form an area in the Karnaugh map. Similarly, all output states of a state should also be assigned adjacent states. In heuristics 1 and 2, we fix the input (i.e., for the same input). These two heuristics are useful in obtaining simplified expressions for the JK inputs. The last heuristic is useful in simplifying the logical expression for the output.

Of course, it is not always possible to satisfy all three heuristics simultaneously. We try our best in assigning adjacent states by giving priority, for example, to the frequency of adjacency requirement.

We illustrate the process of state assignment with the pattern recognition example. We can write the state table, shown in Table 4.6, from the state diagram of Figure 4.24. We can apply our three heuristics to this table. Application of heuristic 1 suggests that states S1, S3, and S5 should be assigned adjacent states as they all have S4 as the next state when the input is 0 (i.e., X = 0). Similarly, when X = 0, S2, S4, and S6 should be assigned adjacent states as they all have S6 as their next state. In our example, we get the same groupings even when X = 1. To indicate that each group is applicable twice, we use superscript 2. When there is a state assignment conflict, we use this weight to resolve the conflict.

Heuristic 1 Groupings: (S1, S3, S5)^2 (S2, S4, S6)^2.

Table 4.6 State table for the pattern recognition example

	Next state		Output	
Present state	X = 0	X = 1	X = 0	X = 1
S0	S2	S1	0	0
S1	S4	S3	0	0
S2	S6	S5	0	0
S3	S4	S3	0	0
S4	S6	S5	1	0
S5	S4	S3	1	0
S6	S6	S5	0	1

Applying Heuristic 2 leads us to the following groupings:

Heuristic 2 Groupings: (S1, S2) $(S3, S4)^3$ $(S5, S6)^3$.

Notice that the (S3, S4) group occurs three times: states S1, S3, and S5 lead to the same set of states. Similarly, states S2, S4, and S6 lead to the (S5, S6) group.

When applying Heuristic 3, if the output is a single bit, it is sufficient to group the states with a 1 output. Thus, we have

Heuristic 3 Groupings: (S4, S5).

We can use a Karnaugh map to get an assignment that satisfies as many adjacency requirements as possible. The Karnaugh map used for the state assignment is shown in Figure 4.25. We start with the assignment of 000 to S0. This initial assignment is arbitrary. From the groupings, we see that there is a strong requirement for assigning adjacent states for S3 and S4 as well as for S5 and S6. Each occurs three times in the groupings obtained by applying Heuristic 2. S1 and S2 should also be assigned adjacent states.

For each assignment, we can use a column in the Karnaugh map to satisfy the adjacency requirement. With this information, the assignment can be arbitrary within each column. Furthermore, placement of these columns in the Karnaugh map can be done arbitrarily. However, if we look at the groupings obtained with Heuristic 1, we see that S1, S3, and S5 should be adjacent. Similarly, S2, S4, and S6 should be adjacent. Well, we cannot satisfy this requirement completely. We can only have two of these three states adjacent. Although this additional

A \ BC	00	01	11	10
0	S0	S3	S5	S1
1		S4	S6	S2

Figure 4.25 Karnaugh map for the state assignment.

Table 4.7 State assignment

State		A	B	C
S0	=	0	0	0
S1	=	0	1	0
S2	=	1	1	0
S3	=	0	0	1
S4	=	1	0	1
S5	=	0	1	1
S6	=	1	1	1

information restricts our state assignment within a column, it still gives us freedom as to the placement of columns relative to S0. Heuristic 3 suggests that S4 and S5 should be assigned adjacent states. We cannot satisfy this requirement as the adjacency requirements from Heuristics 1 and 2 are much stronger.

This leads us to the final assignment shown in Table 4.7. Note that there are other equally good state assignments for this example. In Exercise 4–14, you are asked to experiment with other assignments.

Remaining Design Steps

The remaining steps mentioned in our design process are similar to the design process we have used in the counter design examples. The only difference is that we have to add the inputs and outputs to the design table as shown in Table 4.8. This table has three groups of variables. The first group consists of the current state information A B C and the current input X. Because we have a single bit input, we create two rows with the same A B C values, but with a different

Table 4.8 Design table for the pattern recognition example

Present state			Present state	Next state			Present state	JK flip-flop inputs					
A	B	C	X	A	B	C	Y	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	1	0	0	1	d	1	d	0	d
0	0	0	1	0	1	0	0	0	d	1	d	0	d
0	0	1	0	1	0	1	0	1	d	0	d	d	0
0	0	1	1	0	0	1	0	0	d	0	d	d	0
0	1	0	0	1	0	1	0	1	d	d	1	1	d
0	1	0	1	0	0	1	0	0	d	d	1	1	d
0	1	1	0	1	0	1	1	1	d	d	1	d	0
0	1	1	1	0	0	1	0	0	d	d	1	d	0
1	0	1	0	1	1	1	1	d	0	1	d	d	0
1	0	1	1	0	1	1	0	d	1	1	d	d	0
1	1	0	0	1	1	1	0	d	0	d	0	1	d
1	1	0	1	0	1	1	0	d	1	d	0	1	d
1	1	1	0	1	1	1	0	d	0	d	0	d	0
1	1	1	1	0	1	1	1	d	1	d	0	d	0

X value. For example, see the first two rows: the first row is used to indicate the next state transition when A B C = 000 and the input X = 0; the second row identifies the next state if the input is 1 instead. The number of rows we add with the same current state value is 2^n, where n is the number of bits in the input.

The second group consists of the next state values and the current output. This output is the one associated with the transition from the current state to the next. The third group consists of the JK inputs as in the counter examples. These JK input values are filled using the excitation table for the JK flip-flop (see Table 4.3*b* on page 128).

We use the Karnaugh map method to derive simplified expressions for the JK inputs. The Karnaugh maps along with the simplified logic expressions for the JK inputs of the three flip-flops are shown in Figure 4.26. At this point, we have all the information to design the feedback

circuit of the final sequential circuit. Figure 4.28*a* shows this circuit consisting of three JK flip-flops.

To complete the design of the sequential circuit, we need to design the combinational circuit that generates the output Y. We again use a Karnaugh map whose cells are filled with values from the Y column in Table 4.8. The Karnaugh map along with the simplified expression for Y is shown in Figure 4.27. The logical expression for Y

$$Y = \overline{A}BC\overline{X} + ABCX + A\overline{B}\,\overline{X}$$

can be rewritten as

$$Y = BC(\overline{\overline{A}X + A\overline{X}}) + A\overline{B}\,\overline{X}.$$

An implementation of this expression is shown in Figure 4.28*b*.

Figure 4.28 clearly shows the two main components—the combinational logic circuit and the sequential feedback circuit—of the sequential circuit block diagram shown in Figure 4.1 on page 110.

4.7 Summary

In the last two chapters, we have focused on combinational circuits. In combinational circuits, the output depends only on the current inputs. In contrast, output of a sequential circuit depends both on the current inputs as well as the past history. In other words, sequential circuits are state-dependent whereas the combinational circuits are stateless. The state-dependent behavior of a sequential circuit can be described by means of a finite state machine.

Design of a sequential circuit is relatively more complex than designing a combinational circuit. In sequential circuits, we need a notion of time. We have introduced a clock signal that provides this timing information. Clocks also facilitate synchronization of actions in a large, complex digital system that has both combinational and sequential circuits.

We have discussed two basic types of circuits: latches and flip-flops. The key difference between these two devices is that latches are level sensitive whereas flip-flops are edge-triggered. These devices can be used to store a single bit of data. Thus, they provide the basic capability to design memories. We discuss memory design in Chapter 16.

We have presented some example sequential circuits—shift registers and counters—that are commonly used in digital circuit design. There are several other sequential circuit building blocks that are commercially available.

We have given a detailed description of the sequential circuit design process. We have presented the design process in two steps. To develop your intuition, we have discussed a simple counter design using JK flip-flops. We have then presented a more complex, general sequential circuit design process by using two examples.

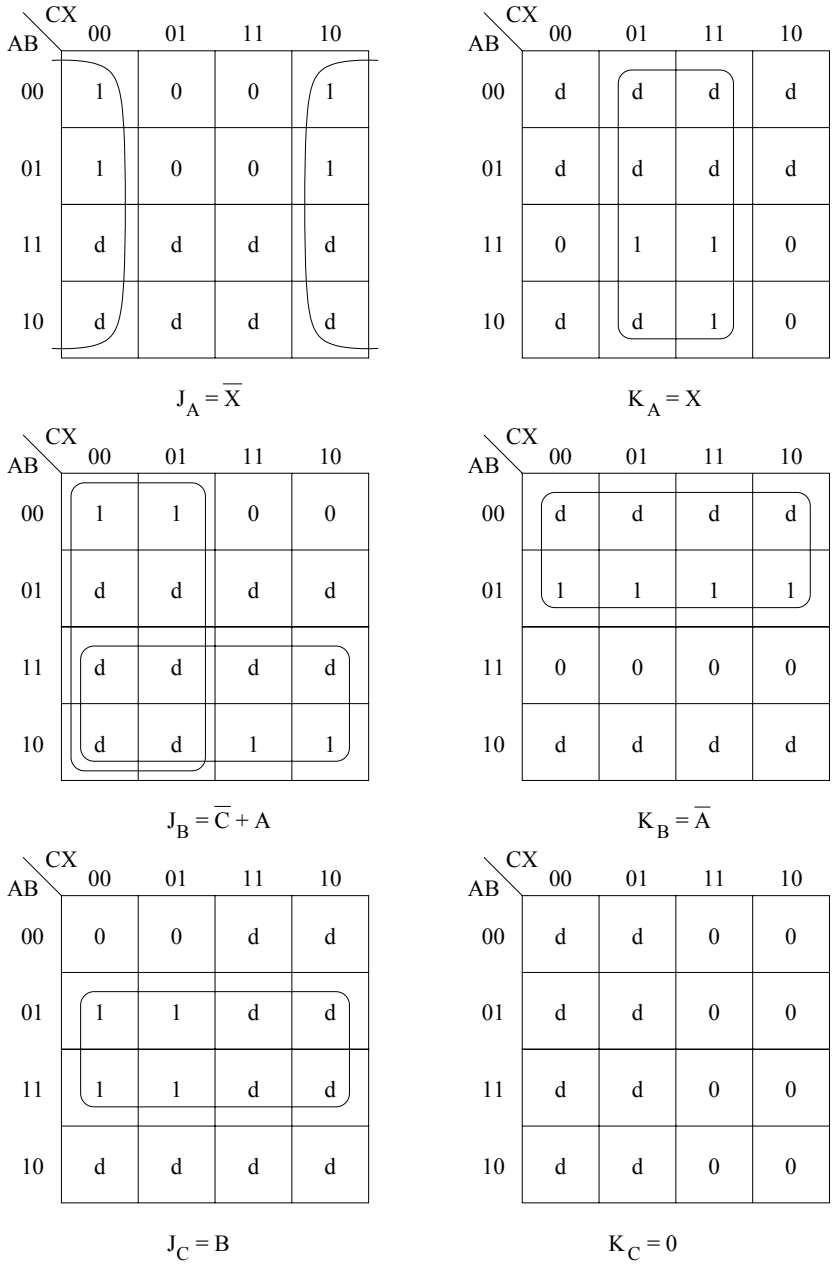


Figure 4.26 Karnaugh maps for the JK inputs.

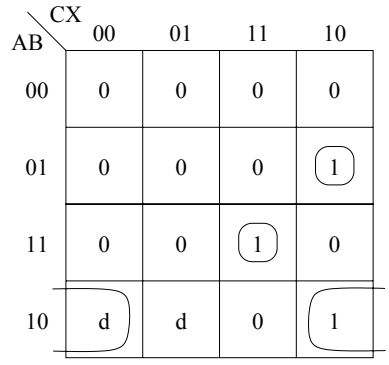


Figure 4.27 Karnaugh map for the output.

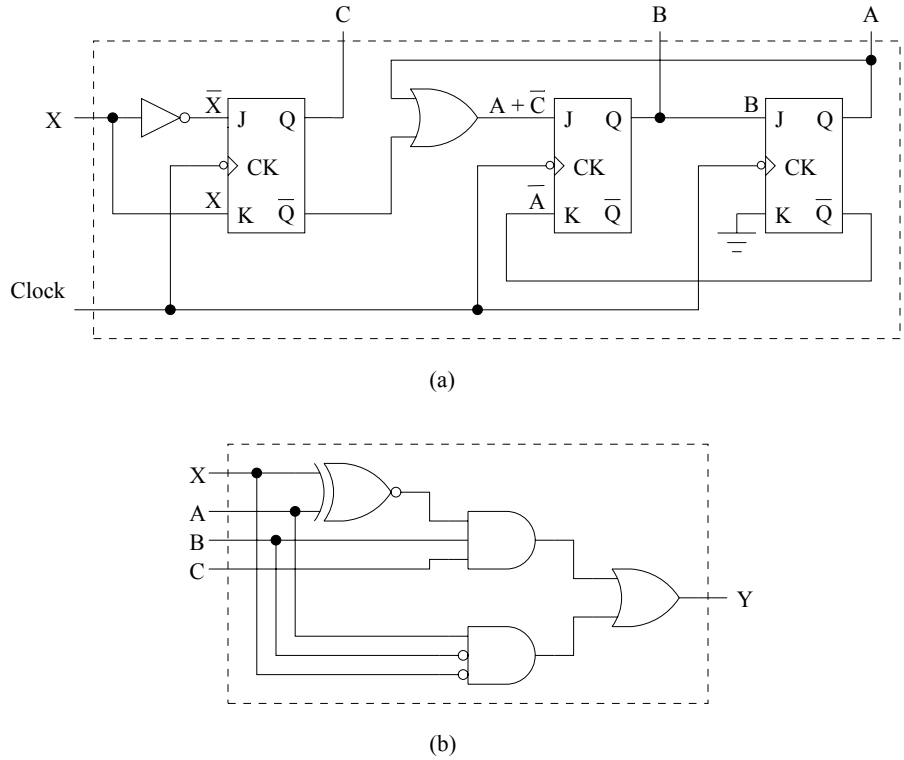


Figure 4.28 Sequential circuit for the pattern recognition example.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Binary counter design
- Clock cycle
- Clock frequency
- Clock period
- Clock signal
- Clocked SR latch
- Counters
- D flip-flops
- D latch
- Falling or trailing edge
- Flip-flops
- General counter design
- JK flip-flops
- Latches
- Rising or leading edge
- Sequential circuit design steps
- Shift registers
- SR latch

4.8 Exercises

4–1 What is main difference between a combinational circuit and a sequential circuit?

4–2 What is the purpose of the feedback circuit in a sequential circuit?

4–3 If the propagation delay is zero, what is the output of the circuit shown in Figure 4.4a?

4–4 Explain the reason for not allowing the input combination S = 1 and R = 1 in Figure 4.5.

4–5 We have shown an implementation of an SR latch in Figure 4.5a using two NOR gates. Suppose we replace the NOR gates by NAND gates. Give the truth table for this new circuit. Can you argue that this new circuit is essentially equivalent to the one in Figure 4.5a?

4–6 How is the D latch avoiding the input combination S = 1 and R = 1?

4–7 What is the difference between a latch and a flip-flop?

4–8 Using the 7476 JK flip-flop chips, construct a 4-bit binary counter. This circuit should have a reset input to initialize the counter to 0.

4–9 Extend the 3-bit synchronous counter shown in Figure 4.15 to implement a 5-bit synchronous counter.

4–10 Extend the 3-bit synchronous counter shown in Figure 4.19 to implement a 4-bit synchronous counter.

4–11 Suppose that we are interested in a 4-bit binary counter that counts through only the even values. That is, the counter starts at 0 and goes through $0 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 12 \rightarrow 14 \rightarrow 0$ and so on. Design such a counter using JK flip-flops.

4–12 In this exercise, implement a 3-bit *gray code* counter. A gray code is a sequence of numbers in which successive numbers differ in only one bit position. An example 3-bit gray code sequence is: 000, 001, 011, 010, 110, 111, 101, and 100. The code is cyclic,

which means that the sequence repeats after 100. Design a counter to implement this gray code sequence.

4–13 We have used the even-parity example to illustrate the general sequential circuit design process. But we did not complete the design (see Example 4.1 on page 132). In this exercise, complete the design of this circuit.

4–14 We have discussed several heuristics to assign states on page 136. Try a different state assignment (possibly the worst) and carry the design through. Compare the final design of your assignment with the one shown in Figure 4.28.

4–15 We have designed a sequential circuit to recognize exactly two zeros in the last three input bits. Can you easily modify the sequential circuit to recognize exactly two ones in the last three inputs?

4–16 Design a sequential circuit that recognizes the bit pattern 010 in an input bit stream. This circuit outputs a 1 whenever the pattern 010 appears. Here is an example input bit stream and the corresponding output of the sequential circuit:

Input: `11101010010101010101`;
Output: `00010100101010100100`0.

As in our pattern recognition example on page 134, assume that the rightmost bit is the first bit to enter the system.

4–17 This is a variation on the last exercise. We are interested in identifying the bit pattern 010 but not on a continuous basis. Once a 010 pattern is recognized, it skips these three bits and looks for another 010 pattern as shown in the following example:

Input: `11101010010101010101`;
Output: `00000100001000100100`0.

Design a sequential circuit for this problem.

4–18 Consider a vending machine that accepts nickels (5 cents) and dimes (10 cents). All products cost 25 cents. The machine requires exact change (i.e., it will not give out change). Design a sequential circuit that accepts 25 cents and activates a selection circuit. Activating the selection circuit enables the user to select an item from the selection panel. Since the machine will not give change, if someone gives three dimes, it will allow the user to select an item (and keeps the extra 5 cents). *Hint:* You can represent the input using two bits: one for the nickel and the other for dime. Then the input combination 01 represents insertion of a nickel; 10 represents insertion of a dime. If the input is 00, it means that no coins have been inserted into the machine. Obviously, input 11 cannot occur, as the machine does not allow insertion of two coins at the same time. You can represent the output by a bit (selection circuit activated or not).

4–19 Modify the last exercise to make the machine give the change. Note that you need to take care of the case when someone inserts three dimes. Do not worry if someone inserts six or more nickels. You simply activate the selection circuit after inserting five nickels.

Chapter 5

System Buses

Objectives

- To discuss bus design issues;
- To illustrate the principles of synchronous and asynchronous buses;
- To describe the concepts of bus arbitration;
- To provide an overview of several example buses including the ISA, PCI, AGP, PCI-X, and PCMCIA.

We have seen in Chapter 1 how the components of a system are connected by using a memory bus consisting of address, data, and control lines. In this chapter, we expand on that discussion and present several ways of interconnecting various system components such as the CPU, memory, and I/O units. We categorize buses into internal and external buses. An external bus is mainly used as an I/O device interface. Consequently, we defer our discussion of these buses to Chapter 19, which discusses I/O issues. After a brief introduction to buses, we present some basic issues in bus design (Section 5.2). Operationally, buses can be classified into two basic types: synchronous and asynchronous. Most internal buses are of the synchronous type, discussed in Section 5.3. Asynchronous buses are briefly described in Section 5.4. A typical bus system may have several devices, called bus masters, competing for the bus use. We need a mechanism to arbitrate requests from several bus masters. We illustrate bus arbitration principles in Section 5.5. The next section describes several example buses. These are the ISA, PCI, AGP, PCI-X, and PCMCIA buses. We conclude the chapter with a summary.

5.1 Introduction

A bus connects various components in a computer system. We use the term *system bus* to represent any bus within a processor system. These buses are also referred to as *internal buses*.

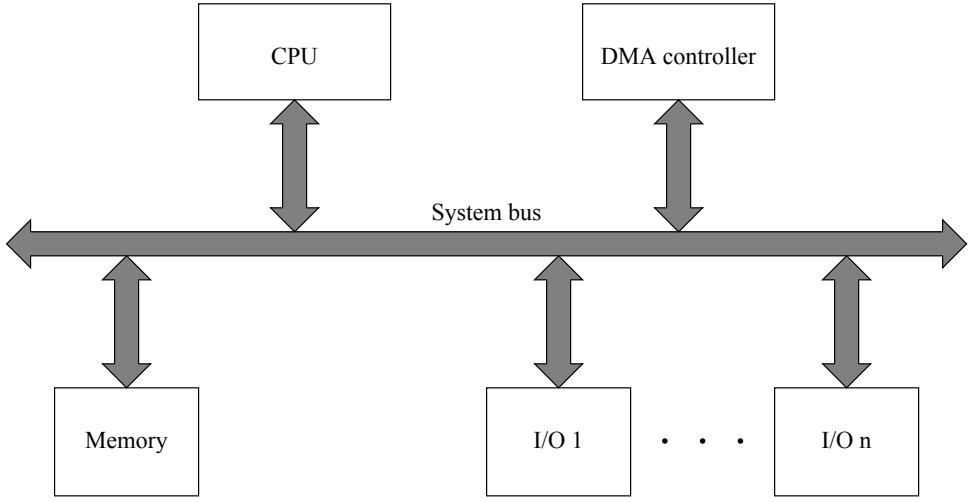


Figure 5.1 A simplified system block diagram showing components of the system bus. An example system bus architecture, which gives more details, is shown in Figure 5.14.

As discussed in Chapter 1, a high-level view of a bus consists of an address bus for addressing information, a data bus to carry data, and a control bus for various control signals.

External buses, on the other hand, are used to interface with the devices outside a typical processor system. By our classification, serial and parallel interfaces, Universal Serial Bus (USB), and FireWire belong to the external category. These buses are typically used to connect I/O devices. We, therefore, defer a description of these external buses to Chapter 19. In this chapter, we focus on internal buses.

Since the bus is a shared resource, we need to define how the devices connected to the bus will use it. For this purpose, we define a *bus transaction* as a sequence of actions to complete a well-defined activity. Some examples of such activities are memory read, memory write, I/O read, burst read, and so on. During a bus transaction, a master device will initiate the transaction, and a slave device will respond to the master's request. In a memory read/write transaction, the CPU is the master and memory is the slave. Some devices such as the memory can only act as slaves. Other devices can act both as a master and slave (but not at the same time). A bus transaction may perform one or more *bus operations*. For example, a Pentium burst read transfers four words. Thus this bus transaction consists of four memory read operations. Each operation may take several bus cycles. A bus cycle is the clock cycle of the bus clock.

Figure 5.1 shows dedicated buses connecting the major components of a computer system. The system bus consists of address, data, and control buses. One problem with the dedicated bus design is that it requires a large number of wires. We can reduce this count by using multiplexed buses. For example, a single bus may be used for both address and data. In addition, the address and data bus widths play an important role in determining the address space and data transfer rate, respectively. We discuss these issues in the next section.

The control bus carries transaction-specific control information. Some typical control signals are given below:

- *Memory Read* and *Memory Write:* These two control signals are used to indicate that the transaction is a memory read or write operation.
- *I/O Read* and *I/O Write:* These signals indicate that the transaction involves an I/O operation. The I/O read is used to read data from an I/O device. The I/O write, on the other hand, is used for writing to an I/O device.
- *Ready:* A target device that requires more time to perform an operation typically uses this signal to relate this fact to the requestor. For example, in a memory read operation, if the memory cannot supply data within the CPU-specified time, it can let the CPU know that it needs more time to complete the read operation. The CPU responds by inserting wait states to extend the read cycle. We discuss this issue in detail later.
- *Bus Request* and *Bus Grant:* Since several devices share the bus, a device should first request the bus before using it. The bus request signal is used to request the bus. This signal is connected to a bus arbiter that arbitrates among the competing requests to use the bus. The bus arbiter conveys its decision to allocate the bus to a device by sending the bus grant signal to that device. Bus arbitration is discussed in Section 5.5.
- *Interrupt* and *Interrupt Acknowledgment:* These two signals are used to facilitate interrupt processing. A device requesting an interrupt service will raise the interrupt signal. For example, if you depress a key, the keyboard generates an interrupt request asking the processor to read the key. When the processor is ready to service the interrupt, it sends the interrupt acknowledgment signal to the interrupting device. A computer system has several devices that require interrupt processing. Therefore, as with the bus arbitration, we need a mechanism to arbitrate among the different interrupt requests. This arbitration is usually done by assigning priorities to interrupts. We discuss interrupt processing in detail in Chapter 20.
- *DMA Request* and *DMA Acknowledgment:* These two signals are used to transfer data between memory and an I/O device in direct memory access mode. The normal mode of data transfer between memory and an I/O device is via the processor. As an example, consider transferring a block of data from a buffer in memory to an I/O device. To perform this transfer, the processor reads a data word from the memory and then writes to the I/O device. It repeats this process until all data are written to the I/O device. This is called programmed I/O. The DMA mode relieves the processor of this chore. The processor issues a command to the DMA controller (see Figure 5.1) by giving appropriate parameters such as the data buffer pointer, buffer size, and the I/O device id. The DMA controller performs the data transfer without any help from the processor. Thus, the processor is free to work on other tasks. Note that when the DMA transfer is taking place, the DMA controller acts as the bus master. We discuss DMA in Chapter 19.
- *Clock:* This signal is used to synchronize operations of the bus and also provides timing information for the operations. Section 5.3 gives more details on this topic.
- *Reset:* This signal initializes the system.

Buses can be designed as either synchronous or asynchronous buses. In synchronous buses, a bus clock provides synchronization of all bus operations. Asynchronous buses do not use a common bus clock signal; instead, these buses use handshaking to complete an operation by using additional synchronization signals. Synchronous buses are described in Section 5.3, and Section 5.4 discusses the asynchronous buses.

From this discussion, it is clear that the bus is shared on a transaction-by-transaction basis. The bus is acquired before the beginning of a transaction and released after its completion. We have skipped an important question: How do we allocate a bus for each bus transaction? A *bus arbiter* does this allocation. We discuss bus arbitration in Section 5.5.

Several example buses are described in Section 5.6. This section describes some important buses including the ISA, PCI, and PCMCIA buses. Looking at these buses gives you an idea of the basic concepts involved in designing buses for computer systems. We discuss external buses such as USB and FireWire in Chapter 19.

5.2 Bus Design Issues

Bus designers need to consider several issues to get the desired cost-performance tradeoff. We have already touched upon some of the issues in the last section. For completeness, here is a list of the bus design issues:

- *Bus Width:* Bus width refers to the *data* and *address* bus widths. System performance improves with a wider data bus as we can move more bytes in parallel. We increase the addressing capacity of the system by adding more address lines.
- *Bus Type:* As discussed in the last section, there are two basic types of buses: *dedicated* and *multiplexed*.
- *Bus Operations:* Bus systems support several types of operations to transfer data. These include the *read*, *write*, *block transfer*, *read-modify-write*, and *interrupt*.
- *Bus Arbitration:* Bus arbitration can be done in one of two ways: *centralized* or *distributed*.
- *Bus Timing:* As mentioned in the last section, buses can be designed as either *synchronous* or *asynchronous*.

We discuss bus arbitration and timing issues in detail in later sections. In the remainder of this section, we look at the first three design issues.

5.2.1 Bus Width

This design parameter mainly deals with the widths of the data and address buses. The control bus width is dictated by various other factors. Data bus width determines how the data are transferred between two communicating entities (e.g., CPU and memory). Although the instruction set architecture may have a specific size, the data bus need not correspond to this value. For example, the Pentium is a 32-bit processor. This simply means that the instructions can work on operands that are up to 32 bits wide. A natural correspondence implies that we should have

a 32-bit data bus. However, data bus width is a critical parameter in determining system performance: the wider the data bus, the higher the bandwidth. Bandwidth refers to the rate of data transfer. For example, we could measure bandwidth in number of bits transferred per second. To improve performance, processors tend to use a wider data bus. For example, even though the Pentium is a 32-bit processor, its data bus is 64 bits wide. Similarly, Intel's 64-bit Itanium processor uses a 128-bit wide data bus.

It is, however, not cheap to build systems with wider buses. They need more space on the motherboard or backplane, wider connectors, and more pins on the chip. For economical reasons, cheaper processor versions use smaller bus widths. For example, the IBM PC started with the 8088 CPU, which is a 16-bit processor just like the 8086. Although the 8086 CPU uses a 16-bit data bus, its cheaper cousin the 8088 uses only 8-bit data lines. Obviously, the 8088 needs two cycles to move a 16-bit value, 8-bits in each cycle.

While we are discussing system performance, we should mention that improvement could also be obtained by increasing the clock frequency. A 1 GHz Pentium moves data at a much faster rate than a 566 MHz Pentium. However, increasing the system clock leads to engineering problems such as *bus skew*. Bus skew is caused by the fact that signals on different lines travel at slightly different speeds. This difference in speed becomes a critical problem as we increase the clock frequency. A 1 GHZ Pentium uses a clock period of 1 ns whereas a 100 MHz Pentium can tolerate minor signal propagation delays as its clock period is 10 ns. Also, at these higher clock frequencies, the length of wire matters as the wire delay becomes comparable to the clock period.

The address bus determines the system memory addressing capacity. A system with n address lines can directly address 2^n memory words. In byte-addressable memories, that means 2^n bytes. With each new generation of processors, we see a substantial increase in memory addressing capacity. For example, the 8086 that was introduced in 1979 had 20 address lines. Thus, it could address up to 1 MB of memory. Later, in 1982, Intel added four more address lines when it introduced the 80286, which is also a 16-bit processor. This has increased the addressing capacity to 16 MB. Finally, in 1985, Intel introduced their 32-bit processor with a 32-bit data bus and 32-bit address bus.

Why did they not use a 32-bit address bus in the first place? The answer lies in the market economy and target application requirements. As mentioned, adding more address lines makes the system more expensive. Considering that in those days, even a 16 KB RAM was priced at about $500, it was important to keep the costs down. At these prices, having a large amount of memory is out of question.

You would think that the 4 GB address space of the Pentium is very large. After all, you and I have systems with less than 512 MB of memory. But there are applications (e.g., servers) that need more address space. For these and other reasons, Intel's 64-bit Itanium processor uses a 64-bit address bus. That's a lot of address space and guarantees that we will not run into address space problems for quite some time.

5.2.2 Bus Type

We have noted that we would like to have wider buses but they increase system cost. For example, a 64-bit processor with 64 data and address lines requires 128 pins just for these two buses. If you want to move 128 bits of data like the Itanium, we need 192 pins! Such designs are called *dedicated bus* designs because we have separate buses dedicated to carry data and address information. The obvious advantage of these designs is the performance we can get out of them. To reduce the cost of such systems we might use *multiplexed bus* designs. In these systems, buses are not dedicated to a function. Instead, both data and address information is time multiplexed on a shared bus. We refer to such a shared bus as an address–data (AD) bus.

To illustrate how multiplexed buses can be used, let us look at memory read and write operations. In a memory read, the CPU places the address on the AD bus. The memory unit reads the address and starts accessing the addressed memory location. In the meantime, the CPU removes the address so that the same lines can be used by memory to place the data. The memory write cycle operates similarly except that the CPU would have to remove the address and then place the data to be written on the AD lines. Obviously, multiplexed bus designs reduce the cost but they also reduce the system performance. In a later section, we discuss the PCI bus that uses multiplexed bus design.

5.2.3 Bus Operations

We have discussed the basic read and write operations. The slave in these operations can be either the memory or an I/O device. These are simple operations that transfer a single word. Processors support a variety of other operations. We mention a few of them here.

Processors provide block transfer operations that read or write several contiguous locations of a memory block. Such block transfers are more efficient than transferring each individual word. The cache line fill is an example that requires reading several bytes of contiguous memory locations. We discuss caches in Chapter 17 but it is sufficient to know that data movement between cache and main memory is in units of cache line size. If the cache line size is 32 bytes, each cache line fill requires 32 bytes of data from memory. The Pentium uses 32-byte cache lines. It provides a block transfer operation that transfers four 64-bit data from memory. Thus, by using this block transfer, we can fill a 32-byte cache line. More details on the block transfer are given in Section 5.3 on page 155.

Read-modify-write operations are useful in multiprocessor systems. In these systems, a shared data structure or a section of code, called the *critical section*, must be accessed on a mutually exclusive basis (i.e., one at a time). Read-modify-write operations support such accesses. Typically, a binary lock variable is used to keep track of the usage of the critical section. Its value is 0 if no one is in the critical section; 1 otherwise. A processor wanting to enter the critical section must first check the lock. If the lock value is 0, it sets the lock to 1 and enters the critical section. The problem is that reading the value of the lock and setting it to 1 must be done atomically. That is, no other operation should be allowed in between these read and write steps. Let us see what happens if we allow other operations. Suppose the lock value is 0. Processor 1 reads this value but before it can write 1 to the lock variable, processor 2 reads zero

as well. Thus, both processors enter the critical section. We can avoid this scenario by using the read-modify-write operation. In this case, a processor reads 0 and then updates the lock to 1 without allowing any other processor to intervene. Processors provide instructions such as test-and-set to implement this type of atomic operation. From the hardware point of view, it is implemented by a special bus lock signal (part of the control bus) that will not allow any other processor to acquire the bus.

Several other bus operations exist. We conclude this section by briefly discussing the interrupt operation. Interrupts are used to draw the attention of the processor for a service required by an I/O device. Since the type of service required depends on the interrupting device, the processor enters an interrupt cycle to get the identification of the interrupting device. From this information, the processor executes an appropriate service needed by the device. For example, when a key is depressed, the processor is interrupted, and the interrupt cycle is used to find that the keyboard wants service (i.e., reading a key stroke). We discuss interrupts in detail in Chapter 20.

5.3 Synchronous Bus

In synchronous buses, a bus clock signal provides the timing information for all actions on the bus. Change in other signals is relative to the falling or rising edge of the clock. In this section we discuss some sample memory read and write bus cycles. Our discussion in this section is loosely based on the Pentium memory read and write cycles.

5.3.1 Basic Operation

Simple bus operations are single read and write cycles. The memory read operation is shown in Figure 5.2. The basic read operation takes three clocks. The read cycle starts with the rising edge of clock T1. During the T1 clock cycle, the CPU places a valid address of the memory location to be read. Since the address is not a single line, we show the valid values by two lines, as some address lines can be 0 and others 1. When we don't care about the values, we show them shaded. For example, the address values at the beginning of T1 are shown shaded, as we don't need a value. After presenting a valid address on the address bus, the CPU asserts two control signals to identify the operation type:

- The IO/$\overline{\text{memory}}$ signal is made low to indicate a memory operation;
- The $\overline{\text{read}}$/write line is also turned low to indicate a read operation.

These two lines together indicate that the current bus transaction is a memory read operation. The CPU samples the $\overline{\text{ready}}$ line at the end of T2. This line is used by slower memories to indicate that they need more time. We discuss wait states later. For now, assume that the memory is able to access the data and does not need additional time. It asserts $\overline{\text{ready}}$ by making it low. The CPU then reads the data presented by the memory on the data bus, removes the address, and deactivates the IO/$\overline{\text{memory}}$ and $\overline{\text{read}}$/write control signals.

The memory write cycle is similar to the read cycle (see Figure 5.3). Since this is a write operation, the $\overline{\text{read}}$/write signal is held high. The difference is that the CPU places data during

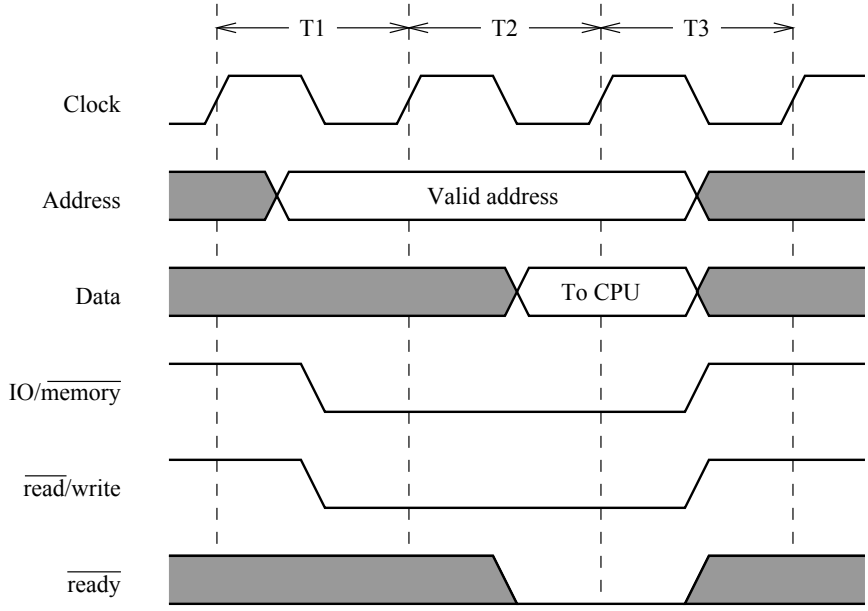


Figure 5.2 Memory read operation with no wait states.

the T2 clock cycle. If the memory is able to write the data without additional wait cycles, it asserts ready during the second clock cycle and writes the data. As in the read cycle, the CPU removes the address and the IO/memory and read/write control signals during the third clock cycle.

Reading from and writing to an I/O device are very similar to the corresponding memory read and write cycles. If the I/O device is memory mapped, it is similar to reading a memory location. In isolated I/O, we need a separate I/O line. This is the purpose of the IO/memory line, which specifies whether the operation is an I/O operation or a memory operation. For now, don't worry about the I/O mapping details. We discuss this topic in Chapter 19.

5.3.2 Wait States

The default timing allowed by the CPU is sometimes insufficient for a slow device to respond. For example, in a memory read cycle, if we have a slow memory it may not be able to supply data during the second clock cycle. In this case, the CPU should not presume that whatever is present on the data bus is the actual data supplied by memory. That is why the CPU always reads the value of the ready line to see if the memory has actually placed the data on the data bus. If this line is high, as in Figure 5.4, the CPU waits one more cycle and samples the ready line again. The CPU inserts wait states as long as the ready line is high. Once this line is low, it reads the data and terminates the read cycle. Figure 5.4 shows a read operation with one wait state.

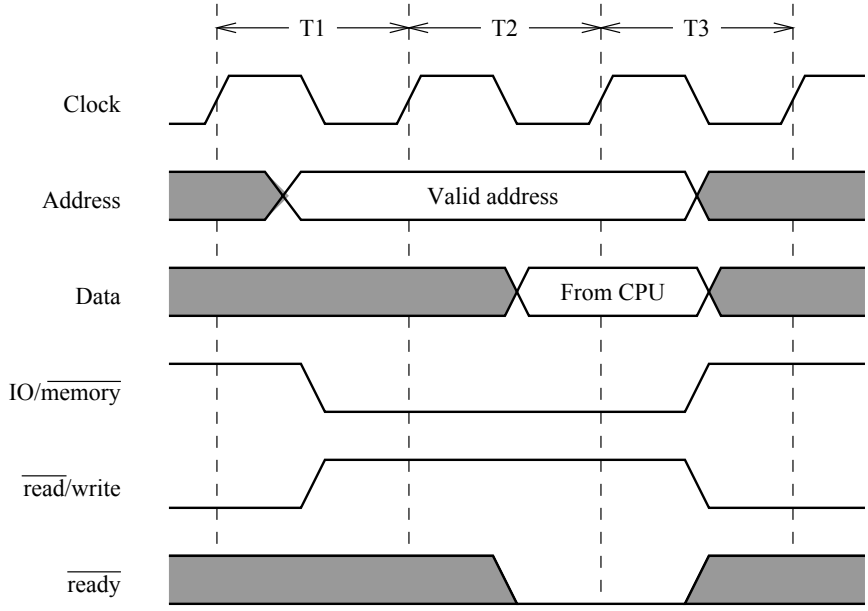


Figure 5.3 Memory write operation with no wait states.

The write cycle with a single wait state is shown in Figure 5.5. The CPU keeps the data on the data bus as long as the ready signal is not asserted. Other details remain the same as before.

5.3.3 Block Transfer

In block transfer mode, more than a single data item is transferred. As mentioned before, such transfers are useful for cache line fills. Figure 5.6, based on the Pentium burst mode transfer, shows a block data transfer of four reads. The processor places the address as in a single read cycle and asserts the IO/memory and read/write control signals. In addition, another control signal (block) is used to initiate the block transfer. In the Pentium, this control line is labeled cache to indicate block transfer of data for a cache line fill.

We assume that the memory does not need wait states. The processor places only the initial address of the data block. The address of the subsequent transfers must be calculated by external hardware. Block transfer of data makes sense for aligned data. Since the Pentium performs block transfers of 32 bytes, the initial address must be a multiple of 32. This means that the least significant five address bits are zeros. The external hardware supplies a counter that increments the address to cycle through the four addresses corresponding to the four data transfers. The Pentium simplifies block data transfer by fixing all data transfers as either single cycle or four cycles.

In general, block transfers may be of a varying number of cycles. In this case, the CPU places the value of the number of cycles on the data bus.

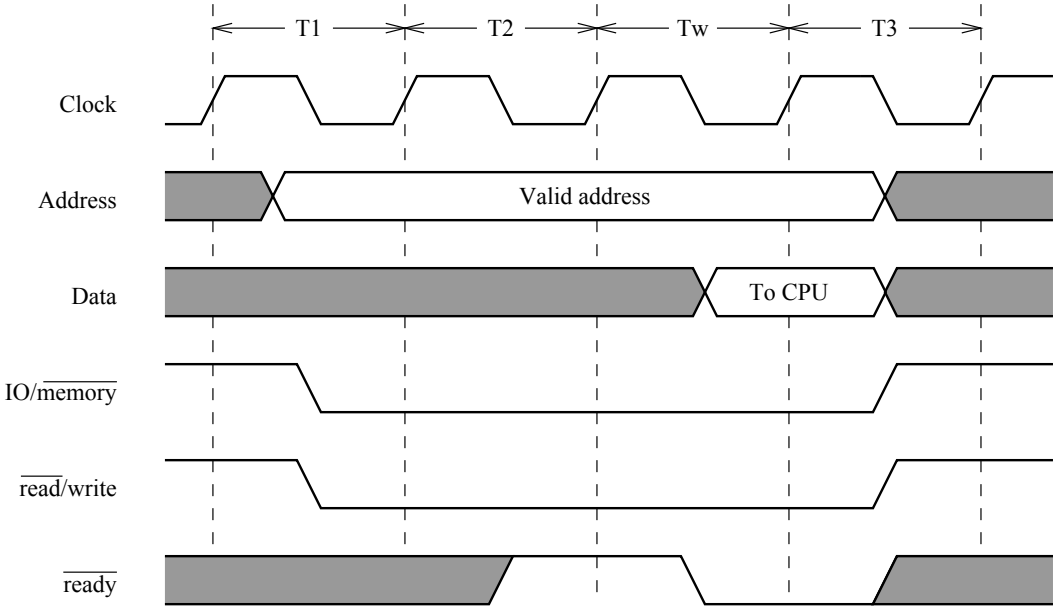


Figure 5.4 Memory read operation with a single wait state.

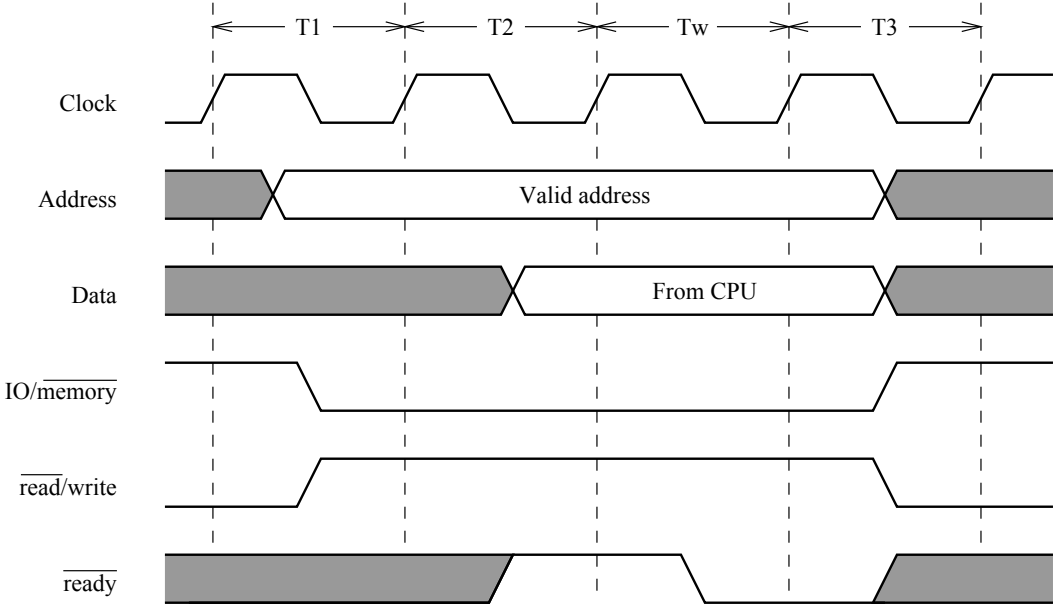


Figure 5.5 Memory write operation with a single wait state.

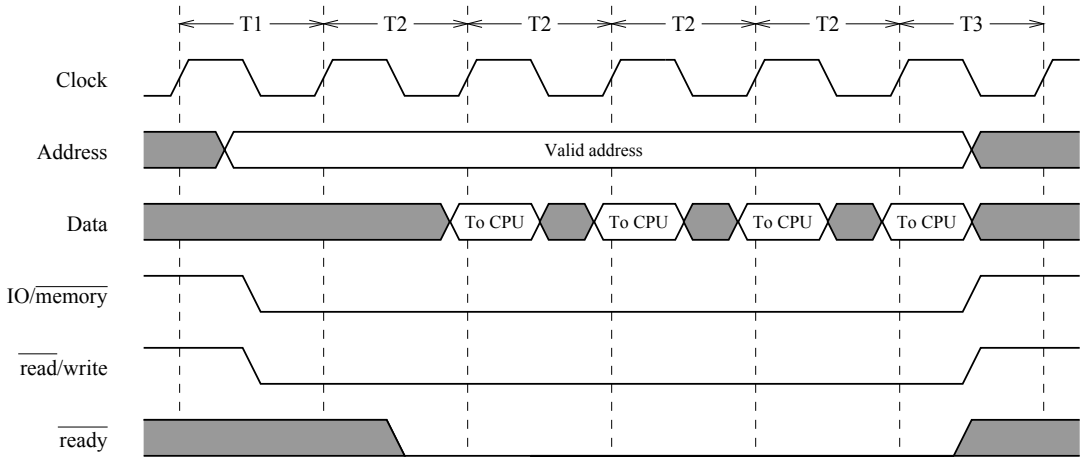


Figure 5.6 Block data transfer of data from memory.

5.4 Asynchronous Bus

In asynchronous buses, there is no clock signal. Instead, they use four-way handshaking to perform a bus transaction. This handshaking is facilitated by two synchronization signals: master synchronization (MSYN) and slave synchronization (SSYN). We can summarize the operation as follows:

1. Typically, the master places all the required data to initiate a bus transaction and asserts the master synchronization signal MSYN.
2. Asserting MSYN indicates that the slave can receive the data and initiate the necessary actions on its part. When the slave is ready with its reply, it asserts SSYN.
3. The master receives the reply and then removes the MSYN signal to indicate receipt. For example, in a memory read transaction, the CPU reads the data supplied by the memory.
4. Finally, in response to the master deasserting MSYN, the slave removes its own synchronization signal SSYN to terminate the bus transaction.

A sample asynchronous read transaction is shown in Figure 5.7. The CPU places the address and two control signals to indicate that this is a memory read cycle. This is done by asserting (i.e., making them low) IO/$\overline{\text{memory}}$ and $\overline{\text{read}}$/write lines. The CPU asserts $\overline{\text{MSYN}}$ after it has placed all the required information on the system bus (point A in the figure). This triggers the memory to initiate a memory read cycle. When the data are placed on the data bus, it asserts $\overline{\text{SSYN}}$ to tell the CPU that the data are available (point B). In response to this, the CPU reads the data from the data bus and removes the signals on all four lines. When the $\overline{\text{MSYN}}$ line is deasserted (point C), which indicates that the CPU has read the data, memory removes the data on the data lines. It also deasserts the $\overline{\text{SSYN}}$ (point D) to complete the read cycle.

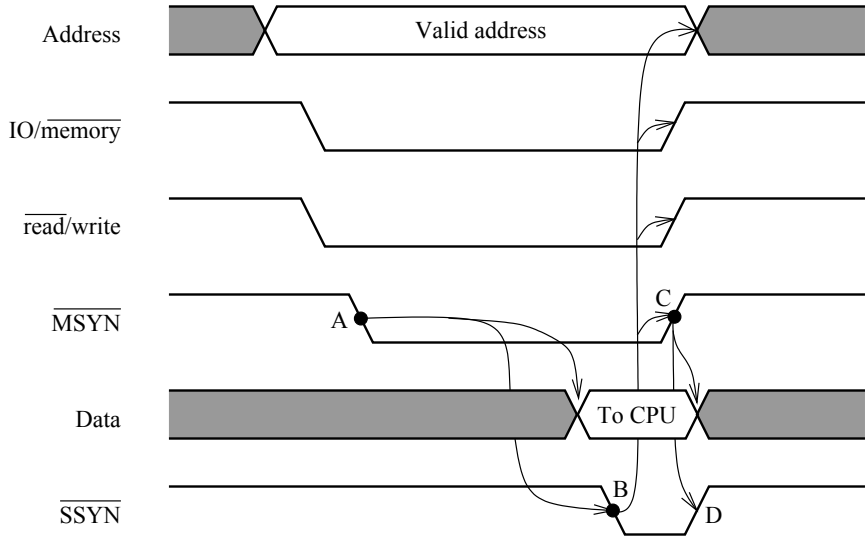


Figure 5.7 An asynchronous read bus transaction.

Asynchronous buses allow more flexibility in timing. In synchronous buses, all timing must be a multiple of the bus clock. For example, if memory requires slightly more time than the default amount, we have to add a complete bus cycle (see the wait cycle in Figure 5.4). Of course, we can increase the bus frequency to counter this problem. But that introduces problems such as bus skew, increased power consumption, the need for faster circuits, and so on.

Thus, choosing an appropriate bus clock frequency is very important for synchronous buses. In determining the clock frequency, we have to consider all devices that will be attached to the bus. When these devices are heterogeneous in the sense that their operating speeds are different, we have to operate the bus at the speed of the slowest device in the system. As the technology improves, bus clock frequency needs to be increased. For example, the PCI bus has two versions: one uses the 33 MHz clock and the other 66 MHz.

The main advantage of asynchronous buses is that they eliminate this dependence on the bus clock. However, synchronous buses are easier to implement, as they do not use handshaking. Almost all system buses are synchronous, partly for historical reasons. In the early days, the difference between the speeds of various devices was not so great as it is now. Since synchronous buses are simpler to implement, designers chose them. Current systems use different types of buses to interface with different components. For example, systems may have a local cache bus for cache memory, another local bus for memory, and another bus such as the PCI for other system components.

5.5 Bus Arbitration

Bus systems with more than one potential bus master need a *bus arbitration* mechanism to allocate the bus to a bus master. Although the CPU is the bus master most of the time, the DMA controller acts as the bus master during certain I/O transfers. In principle, bus arbitration can be done either statically or dynamically. In *static bus arbitration*, bus allocation among the masters is done in a predetermined way. For example, we might use a round-robin allocation that rotates the bus among the masters. The main advantage of a static mechanism is that it is easy to implement. However, since bus allocation follows a predetermined pattern rather than the actual need, a master may be given the bus even if it does not need it. This kind of allocation leads to inefficient use of the bus. Consequently, most implementations use a dynamic bus arbitration, which uses a demand-driven allocation scheme. The rest of our discussion focuses on dynamic bus arbitration.

5.5.1 Dynamic Bus Arbitration

In *dynamic bus arbitration*, bus allocation is done in response to a request from a bus master. To implement dynamic arbitration, each master should have a bus request and grant lines. A bus master uses the bus request line to let others know that it needs the bus to perform a bus transaction. Before it can initiate the bus transaction, it should receive permission to use the bus via the bus grant line. Dynamic arbitration consists of bus allocation and release policies.

Bus arbitration can be implemented in one of two basic ways: *centralized* or *distributed*. In the centralized scheme, a central arbiter receives bus requests from all masters. The arbiter, using the bus allocation policy in effect, determines which bus request should be granted. This decision is conveyed through the bus grant lines. Once the transaction is over, the master holding the bus would release the bus; the release policy determines the actual release mechanism.

In the distributed implementation, arbitration hardware is distributed among the masters. A distributed algorithm is used to determine the master that should get the bus. Figure 5.8 shows the design space for bus arbiter implementation. The following subsections discuss these issues in detail.

Bus Allocation Policies

When several bus masters compete for the bus, the arbiter uses a bus allocation policy to determine the winner. The arbiter can be implemented either in centralized or distributed manner; however, the policy itself is independent of the implementation. We can identify four types of policies:

- *Fixed Priority Policies:* In this policy, each master is assigned a unique fixed priority. When multiple masters request the bus, the highest priority master will get to use the bus. Since the priority is always fixed, careful assignment of priorities is important; otherwise, a higher-priority master can block bus requests from a lower-priority master forever, leading to starvation. However, starvation should not be a problem if the bus requests are intermittent. I/O device bus requests, mainly the DMA service, typically fall

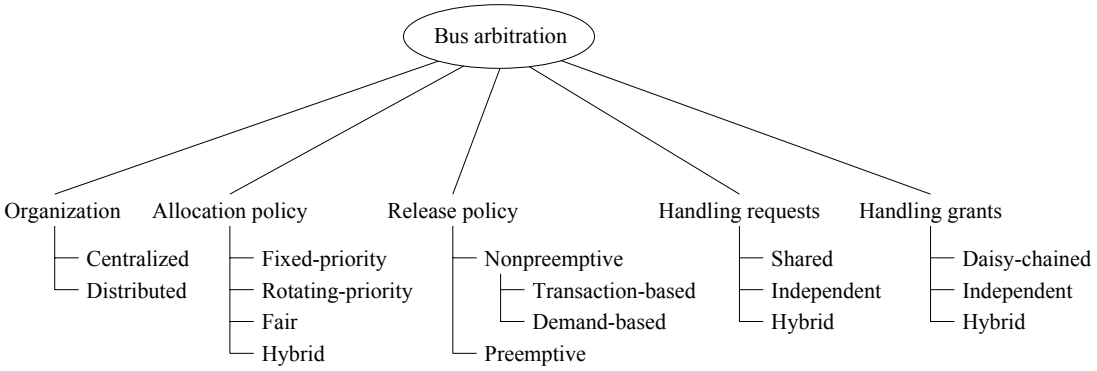


Figure 5.8 Design space for bus arbitration.

into this category. In such cases, fixed priority can be assigned based on the importance (urgency) of the required service.

- *Rotating Priority Policies:* In this policy, priority of a master is not fixed. For example, priority of a master can be a function of the time waiting to get the bus. Thus, the longer a master waits, the higher the priority. An advantage of this policy is that it avoids starvation. Another variation of this policy would reduce the priority of the master that just received service. We can turn this into a round-robin priority allocation by assigning each master that just received service the lowest priority. Thus, each master, after releasing the bus, waits for its turn by joining the queue.
- *Fair Policies:* Fairness is an important criterion in an allocation policy. In its basic form, fairness does not allow starvation. The rotating priority policies, for example, are fair policies. However, fair policies need not use priorities. Fairness can be defined in several ways. For example, fairness can be defined to handle bus requests within a priority class, or requests from several priority classes. Some examples of fairness are: (i) all bus requests in a predefined window must be satisfied before granting requests from the next window; (ii) a bus request should not be pending for more than M milliseconds. For example, in the PCI bus, we can specify fairness by indicating the maximum delay to grant a request.
- *Hybrid Policies:* We can incorporate both priority and fairness into a single policy. These policies are also referred to as *combined* policies. As we show later, PCI bus arbitration uses a hybrid policy.

Bus Release Policies

A bus release policy governs the conditions under which the current bus master releases the bus for use by other bus masters. We can classify the release policies into either nonpreemptive or preemptive policies:

- *Nonpreemptive Policies:* In these policies, the current bus master voluntarily releases the bus. We describe two variations in implementing this policy:
  - *Transaction-Based Release:* A bus master holding the bus releases the bus when its current transaction is finished. This is the simplest policy to implement. If it wants to use the bus for another transaction, it requests the bus again. By releasing the bus after every transaction, fairness can be ensured.
  - *Demand-Based Release:* A drawback with the previous release policy is that if there is only one master that is requesting the bus most of the time, we have to unnecessarily incur arbitration overhead for each bus transaction. This is typically the case in single-processor systems. In these systems, the CPU uses the bus most of the time; DMA requests are relatively infrequent. In demand-based release, the current master releases the bus only if there is a request from another bus master; otherwise, it continues to use the bus. Typically, this check is done at the completion of each transaction. This policy leads to more efficient use of the bus.
- *Preemptive Policies:* A potential disadvantage of the nonpreemptive policies is that a bus master may hold the bus for a long time, depending on the transaction type. For example, long block transfers can hold the bus for extended periods of time. This may cause problems for some types of services where the bus is needed immediately. Preemptive policies force the current master to release the bus without completing its current bus transaction.

5.5.2 Implementation of Dynamic Arbitration

Dynamic bus arbitration can be implemented either by a centralized arbiter or a distributed arbiter. We first discuss centralized arbitration and then present details on distributed arbitration.

Centralized Bus Arbitration

Centralized arbitration can be implemented in several ways. We discuss three basic mechanisms here: daisy-chaining, independent requests, and a hybrid scheme:

Daisy-Chaining: This scheme uses a single, shared bus request signal for all the masters as shown in Figure 5.9. The bus request line is "wired-ANDed" so that the request line goes low if one or more masters request the bus. See page 671 for a discussion of how multiple outputs can be tied together to get the wired-AND functionality.

When the central arbiter receives a bus request, it sends out a bus grant signal to the first master in the chain. The bus grant signals are chained through the masters as shown in Figure 5.9. Each master can pass the incoming bus grant signal to its neighbor in the chain if it does not want to use the bus. On the other hand, if a master wants to use the bus, it grabs the bus grant signal and will not pass it on to its neighbor. This master can then use the bus for its bus transaction. Bus release is done by the release policy in effect.

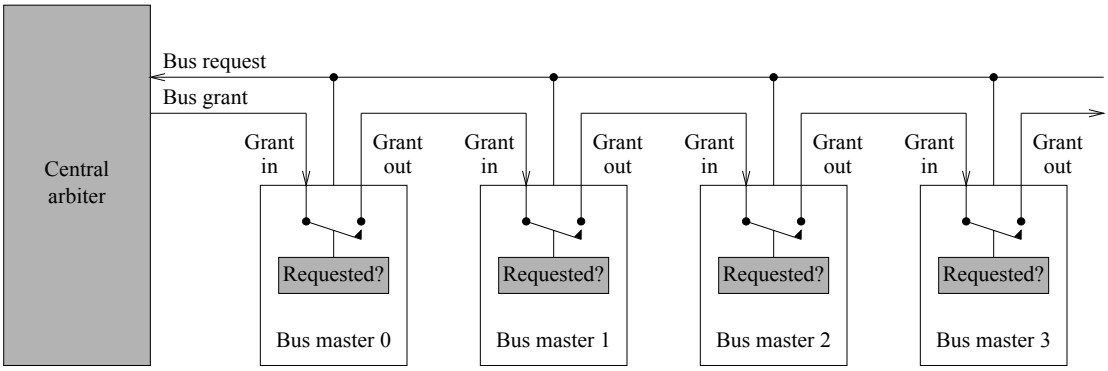


Figure 5.9 A daisy-chained centralized bus arbiter. The incoming grant signal is passed on to the neighbor in the chain only if it does not need the bus.

Daisy-chaining is simple to implement and requires only three control lines independent of the number of hosts. Adding a new bus master is straightforward: simply add to the chain by connecting the three control lines. This scheme has three potential problems:

- It implements a fixed-priority policy. From our discussion, it should be clear that the master closer to the arbiter (in the chain) has a higher priority. In our example, master 0 has the highest priority and master 3 has the lowest. As discussed before, fixed-priority can lead to starvation problems.
- The bus arbitration time varies and is proportional to the number of masters. The reason is that the grant signal has to propagate from master to master. If each master takes Δ time units to propagate the bus grant signal from its input to output, a master that is in the ith position in the chain would experience a delay of $(i-1) \times \Delta$ time units before receiving the grant signal.
- This scheme is not fault tolerant. If a master fails, it may fail to pass the bus grant signal to the master down the chain.

The next implementation mechanism avoids these problems at the expense of increased complexity:

Independent Requests: In this implementation, the arbiter is connected to each master by separate bus request and grant lines as shown in Figure 5.10. When a master wants the bus, it sends its request through its own bus request line. Once the arbiter receives the bus requests from the masters, it uses the allocation policy to determine which master should get the bus next. Since the bus requests are received on separate lines, the arbiter can implement a variety of allocation policies: a rotating-priority policy, a fair policy, or even a hybrid policy. As we show later, PCI bus arbitration uses this implementation technique.

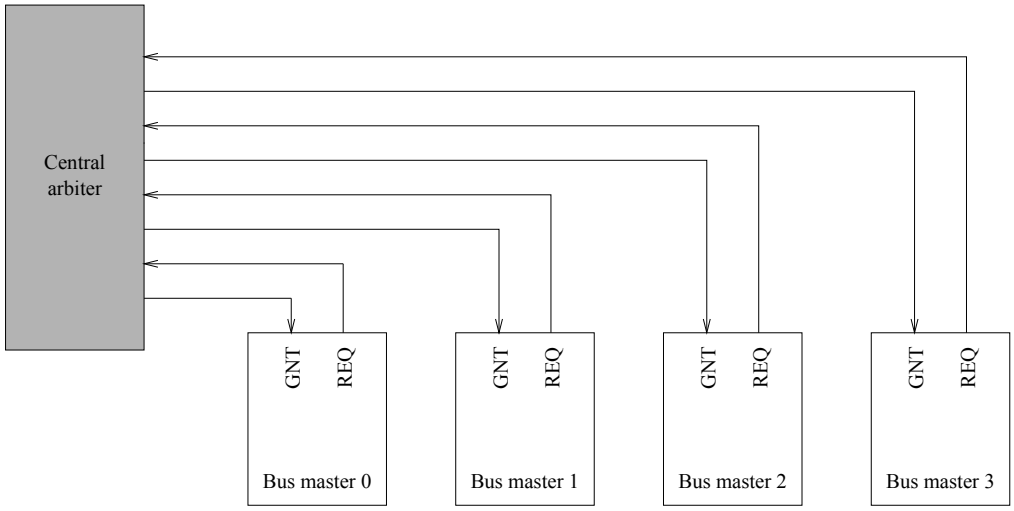


Figure 5.10 A centralized arbiter with independent bus request and grant lines.

This scheme avoids the pitfalls associated with the daisy-chain implementation. It provides short, constant arbitration times and allows flexible priority assignment so that fairness can be ensured. In addition, it provides good fault tolerance. If a master fails, the arbiter can ignore it and continue to serve the other masters. Of course, there is no free lunch. This implementation is complex. The number of control signals is proportional to the number of masters.

Hybrid Scheme: The preceding two implementation techniques represent two extremes. Daisy-chaining is cheaper but has three potential disadvantages. Although the independent request/grant lines scheme rectifies these problems, it is expensive to implement. We can implement a hybrid scheme that is a combination of these two implementation techniques.

In this scheme, bus masters are divided into N classes. Separate bus request and grant lines are used for each class as in the independent scheme. However, within each class, bus masters are connected using daisy-chaining. The VME bus uses this scheme for bus arbitration. The VME bus has four pairs of bus request/grant lines as shown in Figure 5.11.

The VME bus uses three allocation policies:

- *Fixed-Priority:* In this allocation policy, bus request 0 (BR0) is assigned the lowest priority and BR3 has the highest priority.
- *Rotating-Priority:* This is similar to the rotating priority discussed before. It uses this priority scheme to implement round-robin allocation by assigning the lowest priority to the master that just used the bus. Systems requiring fairness can use this allocation policy.
- *Daisy-Chaining:* We can also implement a pure daisy-chaining by connecting all the bus masters to grant request line BR3. In this mode, the arbiter only responds to requests on the BR3 line.

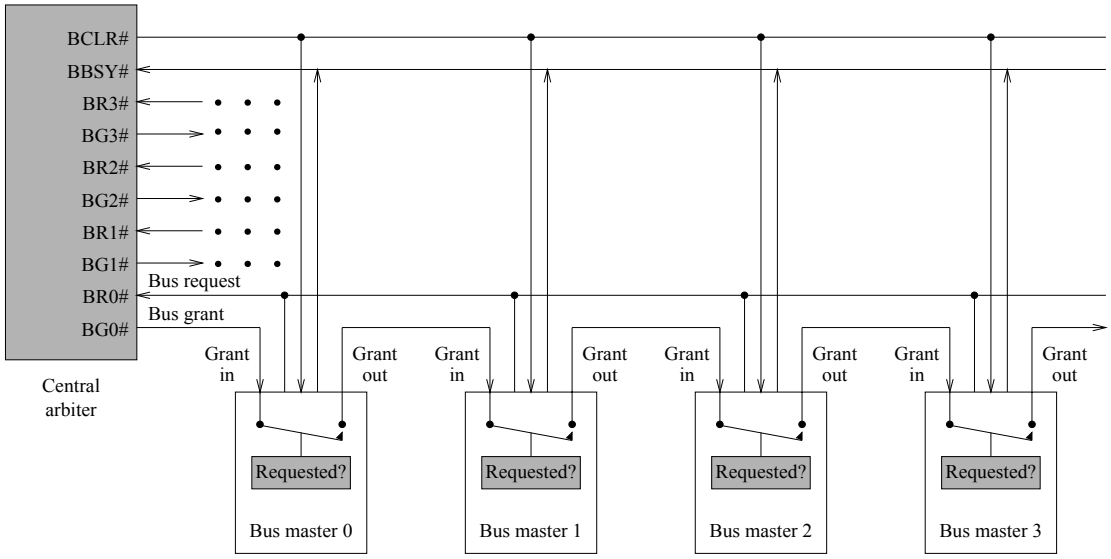


Figure 5.11 Bus arbitration scheme in the VME bus. All four classes share the bus busy (BBSY#) and bus clear (BCLR#) signals.

The default release policy is the transaction-based, nonpreemptive release policy. However, when fixed-priority allocation is used, preemptive release can be used when a higher priority bus request arrives. To effect preemption, the arbiter asserts the bus clear (BCLR) line. The current master releases the bus when the BCLR line is low.

Distributed Bus Arbitration

In distributed arbitration, bus masters themselves determine who should get the bus for the next transaction cycle. The hardware to implement the arbitration policy is distributed among the bus masters. We can have distributed versions of the daisy-chaining and independent request schemes discussed before.

A careful look at the daisy-chaining scheme reveals that the central arbiter is simply initiating the grant signal. We can do the same without using a central arbiter as shown in Figure 5.12. The bus request line is wired-ANDed so that this line goes low if one or more bus masters request the bus. The current bus master asserts the busy line. The grant line is daisy-chained through the bus masters. The leftmost grant line is grounded so that the GIN# of the leftmost master is asserted. The leftmost master that requests the bus receives GIN# as low but it will not assert its GOUT# line. Thus, the masters to the right of this node will not get the bus.

We can also use separate bus request and grant lines that are shared by all the masters. The example in Figure 5.13 shows how a system with four bus masters can implement distributed bus arbitration. As in the centralized independent scheme discussed before, each bus master

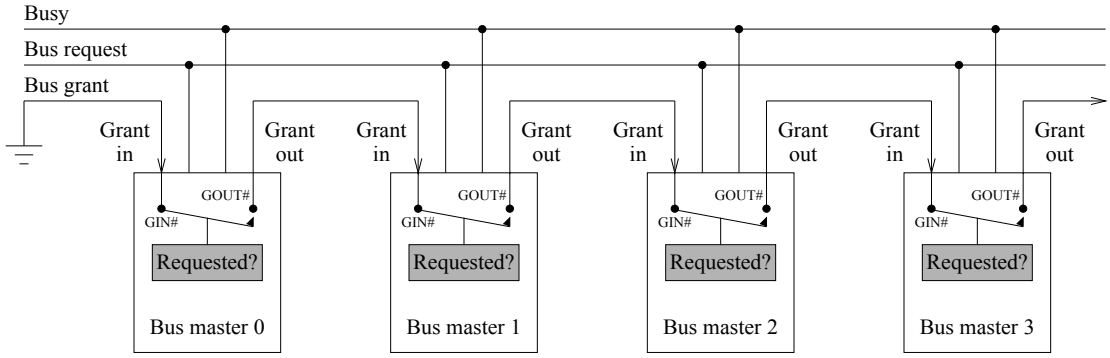


Figure 5.12 A daisy-chained distributed bus arbiter.

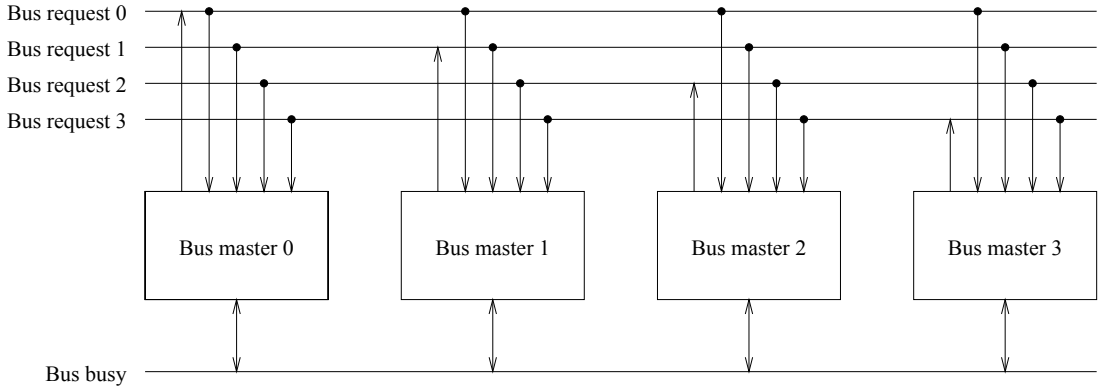


Figure 5.13 A distributed bus arbiter with separate bus request lines.

that needs the bus places a request on its private bus request line. Since all bus masters can read these bus request lines, they can determine the bus master that should get the bus (i.e., the highest priority master). This scheme is similar to the fixed-priority allocation policy. In order to avoid starvation, the highest-priority master that has just used the bus will not raise its bus request until all the other lower-priority masters have been allocated the bus. This honor system would avoid the potential starvation problem.

5.6 Example Buses

We discuss five example buses in this section and start off with the old ISA bus. This bus was proposed by Intel to match their X86 16-bit processor signal lines. It cannot support faster devices. To make the bus independent of the processor, Intel again proposed the PCI bus. This

can operate at either the 33 or 66 MHz frequency. PC systems typically use both PCI and ISA buses as shown in Figure 5.14. The PCI bus is able to support faster I/O devices, and it has been extended to provide even higher bandwidth. We discuss the PCI-X bus in Section 5.6.4.

These buses are not suitable for mobile systems (e.g., laptops). These systems have stringent space and power restrictions. The PCMCIA bus is used for these systems. We discuss this bus in Section 5.6.5. We also describe the AGP port, which provides a high-bandwidth, low-latency interface to support 3D video and full-motion video applications.

5.6.1 The ISA Bus

The Industry Standard Architecture (ISA) bus was closely associated with the system bus used by the IBM PC. The 8088-based IBM PC used an 8-bit data bus. Note that the 8088 is a cheaper version of the 8086 that uses the 8-bit data bus to save on the number of pins. Thus, the 8088 needs two cycles to transfer a 16-bit datum that the 8086 could transfer in a single cycle. Most of the signals in the ISA bus are the same signals emitted by the 8088 processor.

In current systems, the ISA plays the role of I/O bus for slower devices. However, in the IBM PC, the ISA bus was used to interface both memory and I/O devices. Since the IBM PC used the 8088 processor, the first ISA bus had an 8-bit wide data path. This bus had 62 pins, including 20 address lines, eight data lines, six interrupt signals, and one line each for memory read, memory write, I/O read, and I/O write. In addition, it also had four DMA requests and four DMA acknowledgment lines. DMA refers to *direct memory access*, which is a technique used to transfer data between memory and an I/O device without bothering the CPU with the nitty-gritty details of the transfer process. We cover DMA in detail in Chapter 19. In essence, the 8-bit ISA bus had copied the 8088 processor signals, with minor changes, onto the motherboard to interface with the two other components: memory and I/O devices. This bus, therefore, is not a processor independent *standard* bus.

When IBM introduced their PC/AT based on the 80286, the 8-bit ISA bus was not adequate for two main reasons: the 286 provided 16-bit data lines and 24 address lines. That is, compared to the 8088, the 286 provided 8 more data lines and 4 more address lines. Using the 8-bit ISA bus meant the new system could not take advantage of the features of the 286 processor. On the other hand, IBM could not devise an entirely new bus because of the backward compatibility requirement. The solution IBM came up with was to extend the connector by adding 36 signals to handle the additional data and address lines. This part of the ISA connector is referred to as the 16-bit section (see Figure 5.15). In addition to the 8 data lines and 4 address lines, the 16-bit section has five more interrupt request lines, and four more DMA request and acknowledgment lines. There are a few more control signals, including signals to indicate whether the transfer is 8-bits or 16-bits in size. The PC/AT bus was accepted as the bus standard for the ISA.

The ISA system bus operates at 8.33 MHz clock. A zero-wait-state data transfer between the processor and memory takes two bus clocks (i.e., approximately 2×125 ns = 250 ns). Since it can transfer 2 bytes in each cycle, the maximum bandwidth is about 2/250 = 8 MB/s.[1] This

[1] Throughout this chapter, we use MB to refer to 10^6 bytes rather than the usual 2^{20} bytes. Similarly, KB refers to 10^3 bytes. This is customary in the communications area.

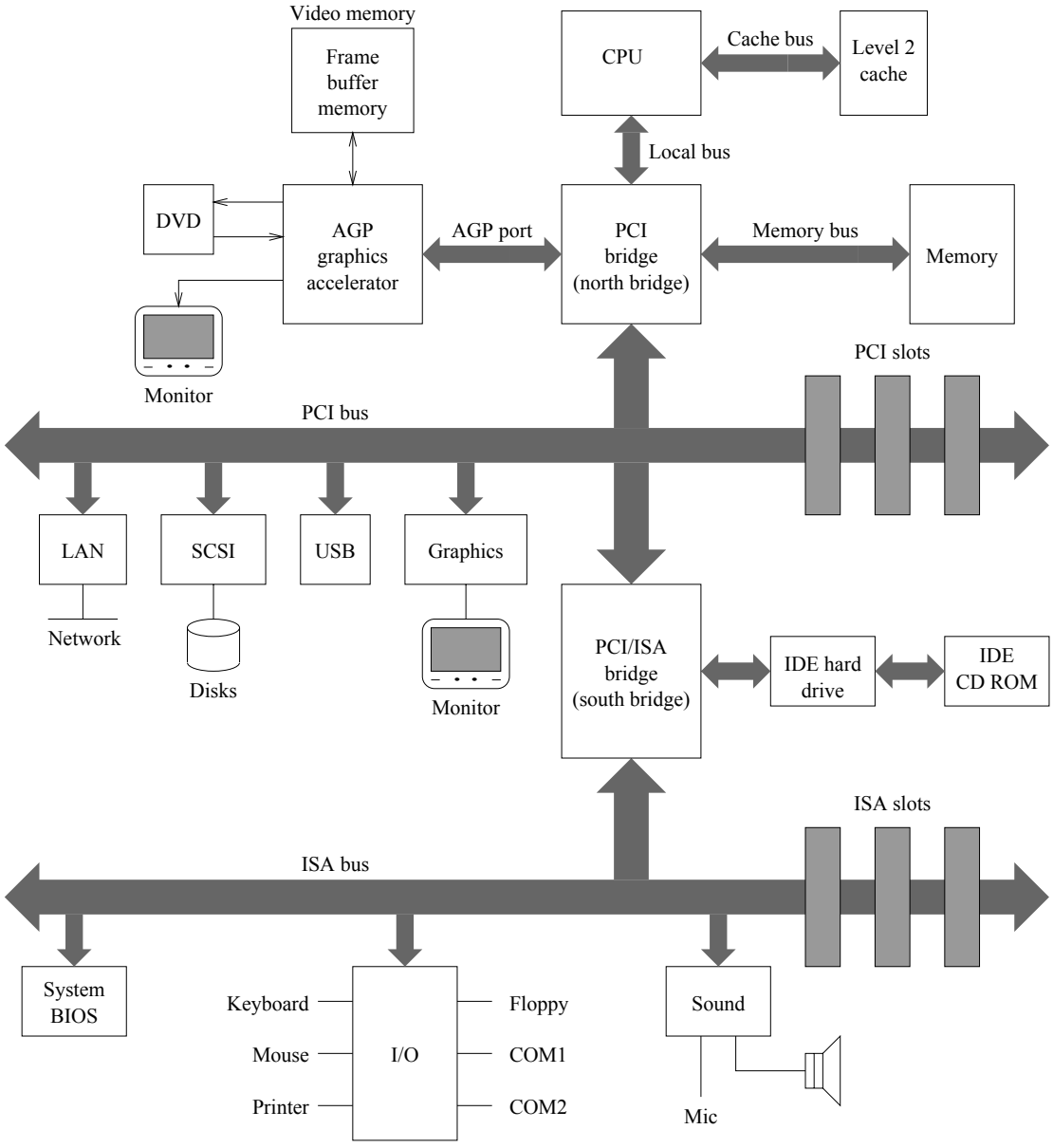


Figure 5.14 Architecture of a PC system with AGP, PCI, and ISA buses.

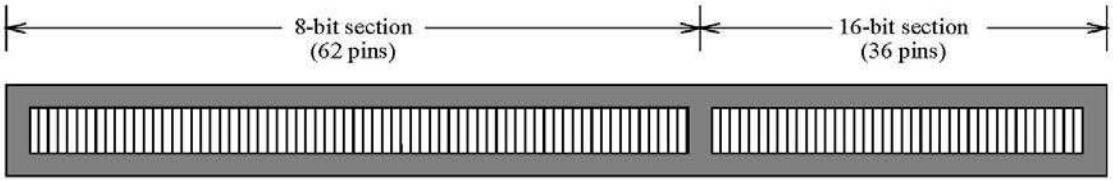


Figure 5.15 The ISA connector consists of an 8-bit section and a 16-bit section.

bandwidth was sufficient at that time as memories were slower and there were no multimedia, windows GUI, and the like to worry about.

When 32-bit processors were introduced, IBM proposed a new 32-bit bus called *Micro Channel Architecture* (MCA). The MCA bus was capable of bus mastering and automatic configuration. Bus mastering allows more than one device to access the bus. As IBM often does, it treated the MCA bus as a proprietary design. The MCA bus never really succeeded.

The ISA also extended the ISA bus to accommodate the increased data bus width. Like the MCA bus, the extended ISA (EISA) bus had a group of signals to facilitate bus mastering (similar to the bus arbitration we discussed before) to allow multiple bus masters. However, the EISA bus also never really took off mainly because of the bandwidth limitations for the new windows-based applications including color video. In fact, the demand for bandwidth for full-motion 3D video is so high we often use a special interface to serve this purpose.

We do not discuss the VESA bus proposed by the Video Electronics Standard Association (VESA) for video cards. The VESA-local bus (VL bus) is connected to the CPU internal bus to achieve the required bandwidth for faster video cards. It can transfer data at 132 MB/s. We describe another popular video interface called the AGP (Accelerated Graphics Port) later.

The ISA bus is still present in current systems for backward compatibility. It is mainly used to interface older, slower I/O devices such as the keyboard and mouse. For faster devices, current systems use the PCI bus, which we discuss next. Figure 5.14 shows how these two buses can coexist to service various I/O devices.

The ISA bus survived nearly two decades. It appears that it will not survive much longer. Efforts are underway to get rid of this bus. We can push backward compatibility only to a point. All new slower I/O devices can use new standards such as the USB, which is discussed in Chapter 19. Mainly due to this reason, we have not presented a great many details on the ISA bus. The next section discuses the PCI bus in greater detail.

5.6.2 The PCI Bus

The bandwidth provided by the ISA bus was sufficient to support mostly text-based applications on early PCs. With the introduction of Windows and full-motion video, there was need for a high-performance bus. In 1990, Intel began work on a new bus for their Pentium systems. The objective of this bus, called the Peripheral Component Interconnect (PCI) bus, was to support the higher bandwidth requirements of modern window-based systems. To encourage its use,

Intel released its patents on PCI to public domain. Furthermore, Intel formed the PCI Special Interest Group (PCISIG) to maintain and develop the standard. The original PCI specification, developed by Intel, was released as Version 1.0. Version 2.0 was introduced in 1993 and Version 2.1 in 1995. As of this writing, the recent revision to the specifications is Version 2.2. PCI 2.2 introduced power management for mobile systems.

Unlike the ISA bus, the PCI bus is processor independent. A PCI bus can be implemented as either a 32- or 64-bit bus operating at the 33 or 66 MHz clock frequency. The original 32-bit PCI bus operated at 33 MHz (it is actually 33.33 MHz, or a clock period of 30 ns) for a peak bandwidth of 133 MB/s. A 64-bit PCI operating at 66 MHz provides a peak bandwidth of 528 MB/s. Due to technical design problems in implementing the 66 MHz PCI, most PCI implementations use the 33 MHz clock for a bandwidth of about 266 MB/s.

Like the other PC buses, PCI is a synchronous bus. It uses a multiplexed address/data bus to reduce the pin count. Because of this multiplexing, it needs just 64 pins for address and data lines, even though the PCI bus supports 64-bit addresses and 64-bit data. In PCI terminology, the master and slave are called the *initiator* and *target*, respectively. All transactions on the bus are between an initiator and a target. The PCI bus uses a centralized arbiter with independent request lines.

In addition to the two choices—32-bit or 64-bit, 33 MHz or 66 MHz—PCI cards can operate at 5 V (older cards) or 3.3 V (newer ones). Figure 5.16 shows the 32- and 64-bit PCI connectors. As shown in this figure, a 64-bit connector consists of the 32-bit connector and a 64-bit extension (somewhat similar to the ISA connector). The connectors are keyed so that a 3.3 V card cannot be inserted into a 5 V connector and vice versa. The 32-bit card has 120 pins. The 64-bit version has an additional 64 pins. For more details on the connectors and the signals, see [4].

Bus Signals

The PCI bus has two types of system signals—mandatory and optional—which are briefly described below:

Mandatory Signals: These signals are divided into five groups: system signals, address/data/command signals, transaction control signals, bus arbitration signals, and error reporting signals. Note that low-active signals are indicated by #.

System Signals:

- *Clock Signal (CLK):* The clock signal provides the timing information to all bus transactions. PCI devices sample their inputs on the rising edge of the clock. The clock frequency can be between 0 and 33 MHz. PCI devices may be operated at a lower frequency, for example, to save power.
- *Reset Signal (RST#):* This signal is used for system reset. When this signal is asserted, the system is brought to an initial state, including PCI configuration registers of all PCI devices.

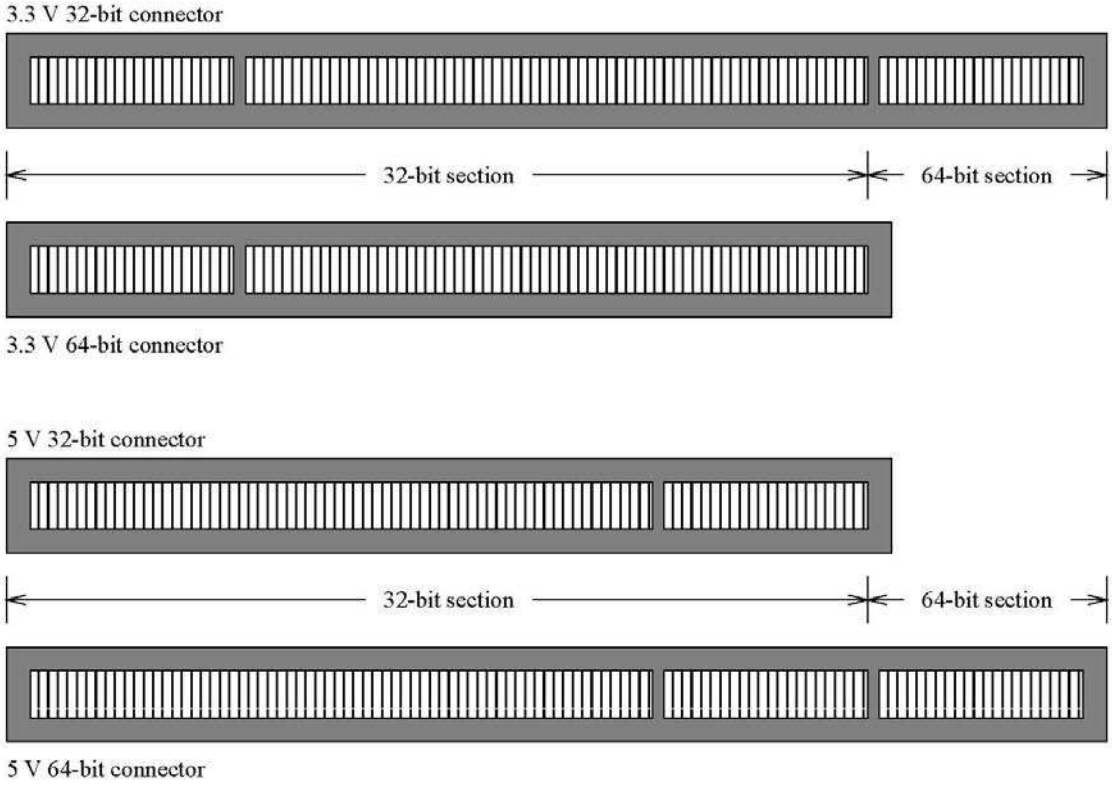


Figure 5.16 PCI connectors for 5 V and 3.3 V.

Address, Data, and Command Signals:

- *Address/Data Bus (AD[0–31]):* These 32 lines are multiplexed to serve as both address and data buses. During the address phase, the AD bus carries the address. Then the bus is used to carry the data. If it is a read operation, one clock cycle is needed to turn around the bus (i.e., to reverse the direction of data transfer).
- *Command Bus (C/BE#[0–3]):* This four-line bus is also time-multiplexed to carry commands as well as byte enable (BE) signals. Typically, during the address phase, the transaction type is identified by the command (memory read, memory write, etc.). During the data phase, these four lines identify which byte of data is to be transferred. Each BE# line identifies one byte of the 32-bit data: BE0# identifies byte 0, BE1# identifies byte 1, and so on. Thus, we can specify any combination of the bytes to be transferred. Two extreme cases are: C/BE# = 0000 indicates transfer of all four bytes, and C/BE# = 1111 indicates a null data phase (no byte transfer). In a multiple data phase bus transaction, the byte enable value can be specified for each data phase. Thus, we can transfer the bytes of

interest in each data phase. The extreme case of null data phase is useful, for example, if you want to skip one or more 32-bit values in the middle of a burst data transfer. If null data transfer is not allowed, we have to terminate the current bus transaction, request the bus again via the arbiter, and restart the transfer with a new address.

- *Parity Signal (PAR):* This line provides even parity for the AD and C/BE# lines. That is, if the number of 1s in the 36 bits (32 AD bits and 4 C/BE# bits) is odd, PAR is 1; it is zero otherwise. Parity provides a rudimentary error detection mechanism.

Transaction Control Signals:

- *Cycle Frame (FRAME#):* The current bus master drives this signal to indicate the start of a bus transaction (when the FRAME# signal goes low). This signal is also used to indicate the length of the bus transaction cycle. This signal is held low until the final data phase of the bus transaction. Our later discussion of the read and write bus transactions clarifies the function of this signal.
- *Initiator Ready (IRDY#):* The current bus master drives this signal. During the write transaction, this signal indicates that the initiator has placed the data on the AD lines. During the read transaction, asserting this signal indicates that the initiator is ready to accept data.
- *Target Ready (TRDY#):* The current target drives this signal. This is the ready signal from the target and complements the ready signal from the initiator. These two ready signals are used to perform full handshaking in transferring data. During a write transaction, the target asserts this signal to indicate that it is ready to accept data. During a read transaction, the target uses this signal to indicate that valid data are present on the AD lines.

  Both IRDY# and TRDY# signals should be asserted for successful data transmission. Otherwise, wait states are inserted until these two signals are low.
- *Stop Transaction (STOP#):* The current target drives this signal to let the initiator know that the target wants to terminate the current transaction. There are several reasons for the target to terminate a transaction. A simple example is that the target device does not support a burst mode of data transfer. If the initiator starts a burst mode, the target terminates the transfer after supplying data during the first data phase.
- *Initialization Device Select (IDSEL):* This is an input signal to a PCI device. The device uses it as a chip select for configuration read and write transactions.
- *Device Select (DEVSEL#):* The selected target device asserts this signal to indicate to the initiator that a target device is present.
- *Bus Lock (LOCK#):* The initiator uses this signal to lock the target to execute an atomic transaction such as test-and-set type of instructions to implement semaphores.

Bus Arbitration Signals:
The PCI system uses centralized bus arbitration with independent request and grant lines. Each PCI device has a request (REQ#) line and a grant (GNT#) line. These are point-to-point lines from each PCI device to the bus arbiter.

- *Bus Request (REQ#):* When a device wants to use the PCI bus, it asserts its REQ# line to the bus arbiter.
- *Bus Grant (GNT#):* This signal is asserted by the bus arbiter to indicate that the device associated with the asserted GNT# line can have the bus for the next bus transaction.

In the PCI system, bus arbitration can overlap execution of another transaction. This overlapped execution improves PCI bus performance.

Error Reporting Signals:

- *Parity Error (PERR#):* This error indicates a data parity error detected by a target during a write data phase, or by the initiator during a read data phase. All devices are generally expected to detect and report this error. There can be exceptions. For example, during a video frame buffer transmission, a data parity error check could be ignored.
- *System Error (SERR#):* Any PCI device may generate this signal to indicate address parity error and critical errors. In a PC system, SERR# is typically connected to the NMI (nonmaskable interrupt) processor input. This error is reserved for catastrophic and nonrecoverable errors.

Optional Signals: In addition, the following optional signals are defined by the PCI specification:

64-Bit Extension Signals:

- *Address/Data Lines (AD[32 to 63]):* These lines are the extension of the basic 32-bit address/data lines.
- *Command Bus (C/BE#[4 to 7]):* The command/byte enable lines are extended by four more lines. During the address phase, these lines are used to provide additional commands. During the data phase, these lines specify the bytes that the initiator intends to transfer.
- *Request 64-Bit Transfer (REQ64#):* The initiator generates this signal to indicate to the target that it would like 64-bit transfers. It has the same timing as the FRAME# signal.
- *Acknowledge 64-Bit Transfer (ACK64#):* The target, in response to the REQ64# signal, indicates that it is capable of transferring 64 bits. This signal has the same timing as the DEVSEL# signal.
- *Parity Bit for Upper Data (PAR64):* This bit provides the even parity for the upper 32 data bits (AD[32 to 63]) and four command lines (C/BE#[4 to 7]).

Interrupt Request Lines:
PCI provides four lines (INTA#, INTB#, INTC#, and INTD#) for PCI devices to generate interrupt service requests. Like the bus arbitration lines, these are not shared; rather, each device has its own interrupt lines connected to an interrupt controller.

In addition, there are also signals to support the snoopy cache protocol and IEEE 1149.1 boundary scan signals to allow incircuit testing of PCI devices.

There is also an M66EN signal to indicate the bus clock frequency. This signal should be held low for the 33 MHZ clock speed and high for the 66 MHz clock.

PCI Commands

The 32-bit PCI has four lines (C/BE#) to identify the bus transaction type. Typically, a command value is placed on these four lines during the address phase. Commands include the following:

- *I/O Operations:* PCI provides two commands to support I/O operations: *I/O Read* and *I/O Write*.
- *Memory Operations:* PCI provides two types of memory commands.
  - *Standard Memory Operations:* Two commands, *Memory Read* and *Memory Write*, allow standard read and write memory operations. All PCI devices should be able to perform these actions.
  - *Bulk Memory Operations:* Three additional memory commands to support block transfer of data are available. These three commands are optional performance enhancement commands.
    - *Memory Read Line:* Use this command if you want to transfer more than a doubleword but less than the size of the cache line. The actual data transferred are restricted to the cache line boundary (i.e., data transfer cannot cross the cache line boundary). This command allows prefetching of data.
    - *Memory Read Multiple:* This command is somewhat similar to the last one, except that it allows cache line boundaries to be crossed in prefetching the data. Thus, it is more flexible compared to the Memory Read Line command.
    - *Memory Write-and-Invalidate:* This command is similar to the standard Memory Write command, except that it guarantees the transfer of a complete cache line(s) during the current bus transaction.
- *Configuration Operations:* Every PCI device must have a 256-byte configuration space that can be accessed by other PCI devices. Configuration space of a device maintains information on the device. This information can be used for auto configuration for plug-and-play operation. Two commands to read and write the configuration space are provided: *Configuration Read* and *Configuration Write*.
- *Miscellaneous Operations:* Two more commands are provided to support 64-bit addresses on a 32-bit PCI and to convey a special message to a target group:

Special Cycle Command: Special Command is used to broadcast a message to the targets on the PCI bus. Two types of messages are conveyed by this command: *Shutdown* and *Halt*.

Dual Address Cycle Command: This command allows a 32-bit initiator to use 64-bit addresses to access memory beyond the 4 GB boundary. The 64-bit address is passed as two 32-bit values.

- During the first cycle of the bus transaction, the lower 32 bits of the 64-bit address are placed on the 32 AD lines. It uses the dual address cycle (DAC) command on C/BE# lines to indicate that the current bus transaction uses two cycles to send the 64-bit address.
- During the second clock cycle, the higher 32 bits of the address are placed on the AD lines and the normal bus command is placed on the C/BE# lines.

PCI Operations

As mentioned before, PCI transactions typically start with an address phase during which the address and the type of bus transaction are announced. This phase is followed by one or more data phases. To give you a clear idea of the type of PCI bus transactions, we discuss three operations: a simple read, a simple write, and a burst read operation.

PCI Read Operation: Figure 5.17 illustrates the timing of a single data phase read operation. The read bus transaction consists of a single-cycle address phase and a two-cycle data phase. After a bus master has acquired the bus, it asserts FRAME# to start the read cycle. The initiator places the address on the AD lines and the command (i.e., read command in this case) on the C/BE# lines. At the end of T1 (on the rising edge), a PCI device uses the address to learn that it is the target of the transaction. In order for the target to do this, the address and command signals should be stable for at least T_{setup} time units. We discuss this further later.

Shortly after this rising edge of T1, the initiator floats the AD bus in preparation for the target to place data on the AD lines. In the figure, two circular arrows show this turnaround of the bus. The initiator then asserts the IRDY# signal to indicate to the target that it is ready to accept the data. At the same time, it changes the command on the C/BE lines to Byte Enable to indicate the bytes it intends to transfer.

The initiator then deasserts the FRAME# signal to indicate that this is the last data phase. In this example, we are reading only one 32-bit datum. In general, the FRAME# signal remains asserted until the target is ready to complete the last data phase.

During the T3 cycle, the target asserts DEVSEL# to inform the initiator that a target device is responding to its request. When it is ready to transmit the requested data, it asserts the TRDY# signal.

When the initiator accepts the data, it removes the IRDY# signal. The target responds to this acknowledgment by removing data from the AD bus and deasserting the TRDY# and DEVSEL# signals. This completes the bus transaction.

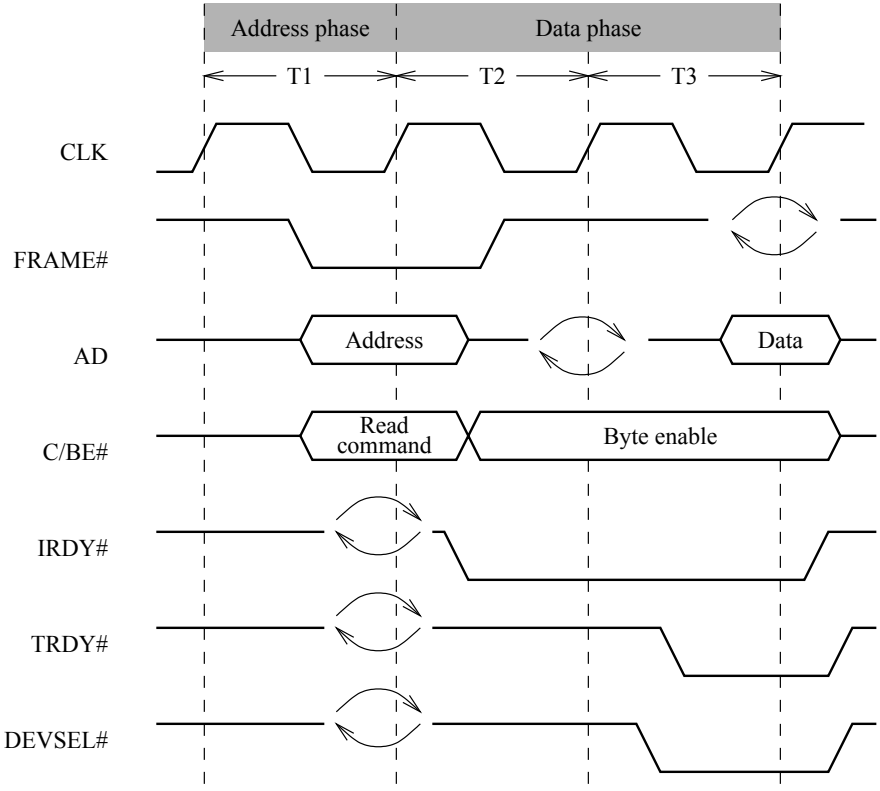


Figure 5.17 A single data phase PCI read operation. This operation can be used to read a 32-bit word.

PCI Write Operation: The single-phase PCI write operation is shown in Figure 5.18. Its timing is very similar to that of the read operation. In the write operation, the data phase takes only one clock. In contrast, the read operation's data phase needs two clocks. This is due to the fact that in a write operation the initiator provides data to the target. Thus, after the address has been removed from the AD bus, the initiator can place the data to be written. We do not need time to turn the bus around, which saves a clock cycle.

PCI Burst Read Operation: As an example of a bus transaction that transfers more than one data word, we describe a burst mode read transaction. Its timing is shown in Figure 5.19. The first three clock cycles are similar to the single-phase read operation described before. How does the target know that the initiator wants more data phases? The initiator keeps the FRAME# low to indicate that it wants another data phase. The initiator's willingness to accept a new data value is indicated by the asserted IRDY# signal.

The initiator and target use IRDY# and TRDY# signals to control the transfer speed. If the target is not ready to provide data, it deasserts the TRDY# signal to introduce wait states. In

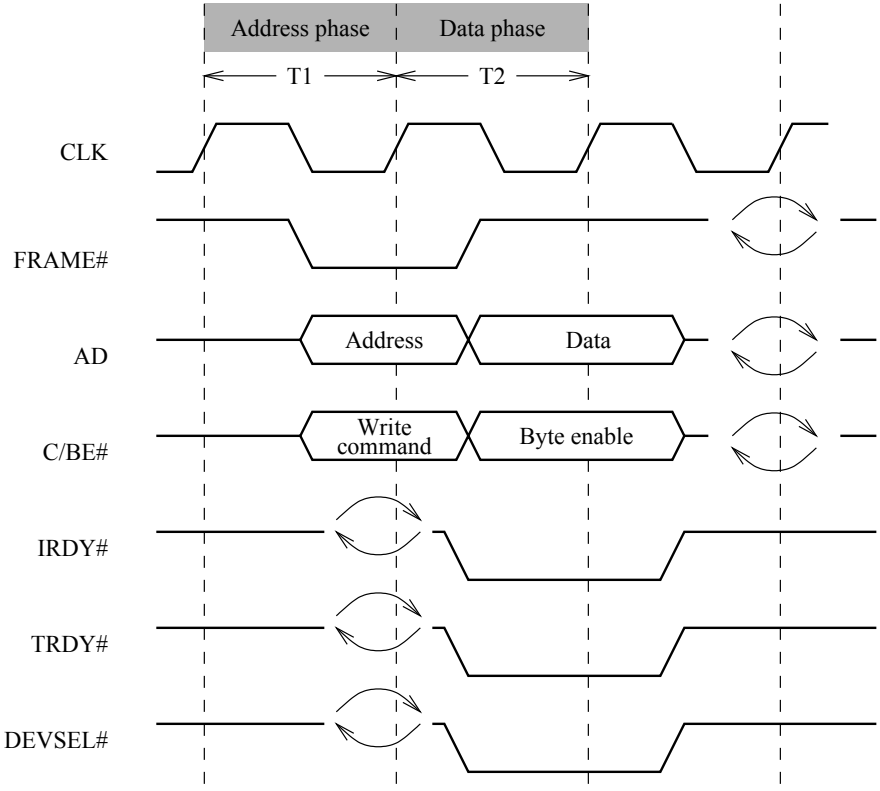


Figure 5.18 A single data phase PCI write operation.

Figure 5.19, the target deasserts TRDY# during T4 to insert a wait state (cycle T5). This wait state extends the second data phase to two clocks. A no-wait-state data phase requires only one clock, except for the first one during a read operation. As mentioned, the first data phase in a read operation needs one additional clock cycle to turn the AD bus around.

In this example, the third data phase is also two clocks long because of the wait state requested by the initiator. This wait state (T7) is inserted because the initiator deasserted the IRDY# signal during the T6 clock. Wait states can also be inserted into a data phase by both the initiator and target deasserting IRDY# and TRDY# concurrently.

PCI Bus Arbitration

PCI uses a centralized bus arbitration with independent grant and request lines (see Figure 5.10). As mentioned before, each device has separate grant (GNT#) and request (REQ#) lines connected to the central arbiter. The PCI specification does not mandate a particular arbitration policy. However, it mandates that the policy should be a fair one to avoid starvation.

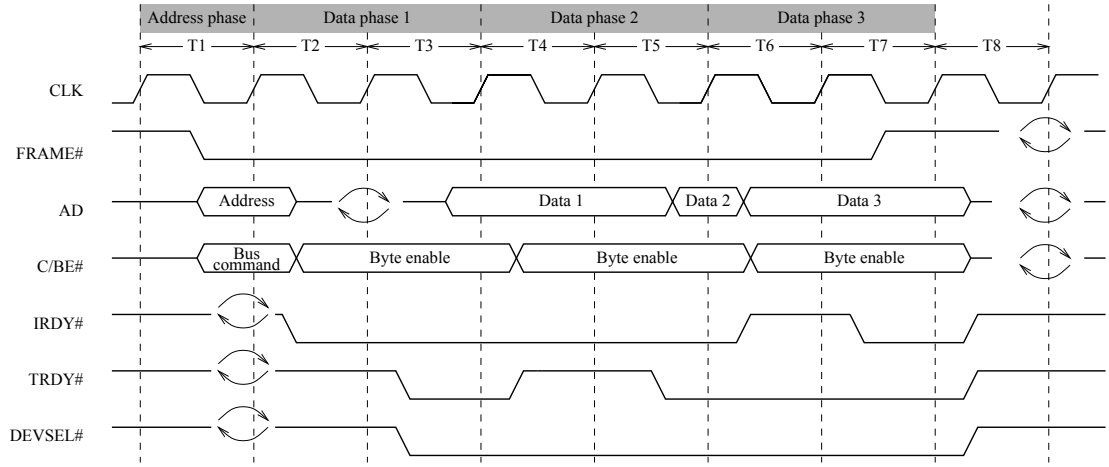


Figure 5.19 A PCI burst read operation: During the second data phase, a wait state (T5) is inserted due to the target's inability to supply data, indicated by making the TRDY# signal high during T4. The third data phase also has a wait state (T7). This time it is the initiator that is not ready to accept the data as indicated by deasserting IRDY# signal during T6.

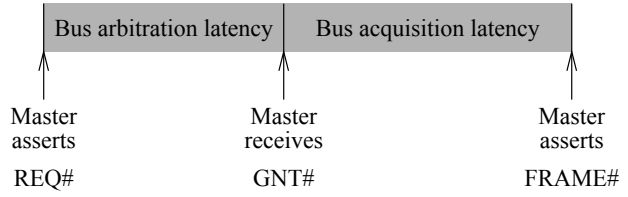


Figure 5.20 Two delay components in acquiring the PCI bus: The arbitration takes place while another master is using the bus. Thus, arbitration delay will not idle the PCI bus. The acquisition latency depends on when the current bus master releases the bus.

A device that is not the current master can request the bus by asserting the REQ# line. The arbitration takes place while the current bus master is using the bus. When the arbiter notifies a master that it can use the bus for the next transaction, the master must wait until the current bus master has released the bus (i.e., the bus is idle). The bus idle condition is indicated when both FRAME# and IRDY# are high.

A master requesting the bus experiences the two delay components (shown in Figure 5.20) from the time it intends to use the bus to the actual start of the bus transaction. However, PCI uses hidden bus arbitration in the sense that the arbiter works while another bus master is running its transaction on the PCI bus. This overlapped bus arbitration increases the PCI bus utilization by not keeping the bus idle during arbitration.

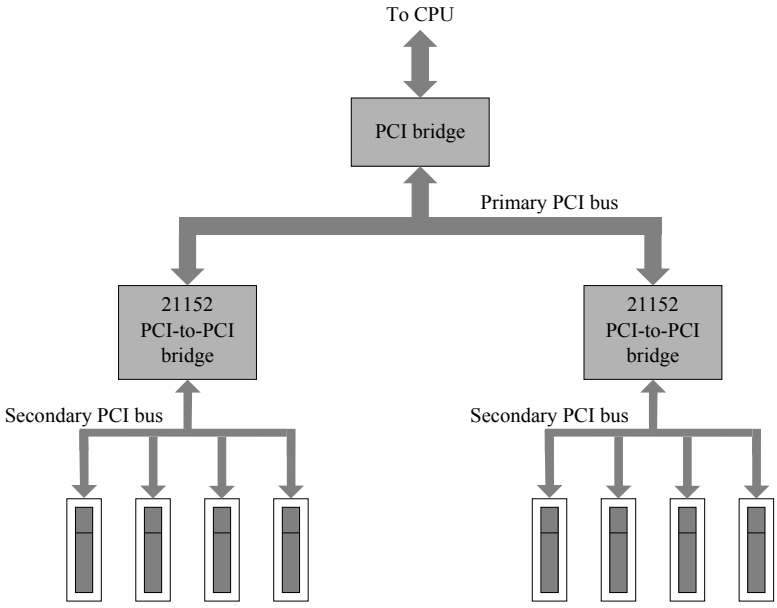


Figure 5.21 Building hierarchical PCI bus systems using the Intel 21152 PCI-to-PCI bridge chip.

PCI devices should request a bus for each transaction. However, a transaction may consist of an address phase and one or more data phases. For efficiency, data should be transferred in burst mode. Although we have not discussed it here, PCI specification has safeguards to avoid a single master from monopolizing the bus and to force a master to release the bus.

Building Hierarchical PCI Buses

PCI allows hierarchical PCI bus systems. These are typically built using PCI-to-PCI bridges. As an example of such a bridge, we present details on the Intel 21152 PCI-to-PCI bridge chip. This chip connects two independent PCI buses: a primary and a secondary. Of the two PCI buses, the one closer to the CPU is called the primary PCI bus; the other is called the secondary. Each secondary PCI bus supports up to four PCI devices, as shown in Figure 5.21.

The 21152 chip allows concurrent operation of the two PCI buses. For example, a master and target on the same PCI bus can communicate while the other PCI bus is busy. The bridge also provides traffic filtering which minimizes the traffic crossing over to the other side. Obviously, this traffic separation along with concurrent operation improves overall system performance for bandwidth-hungry applications such as multimedia.

Since the 21152 is connecting two PCI buses, we need two bus arbiters: one for the primary and the other for the secondary. On the primary side, the 21152 depends on an external arbiter available in the system. It has an internal bus arbiter for the secondary side. If the designer wishes, this internal arbiter can be disabled to allow an external arbiter.

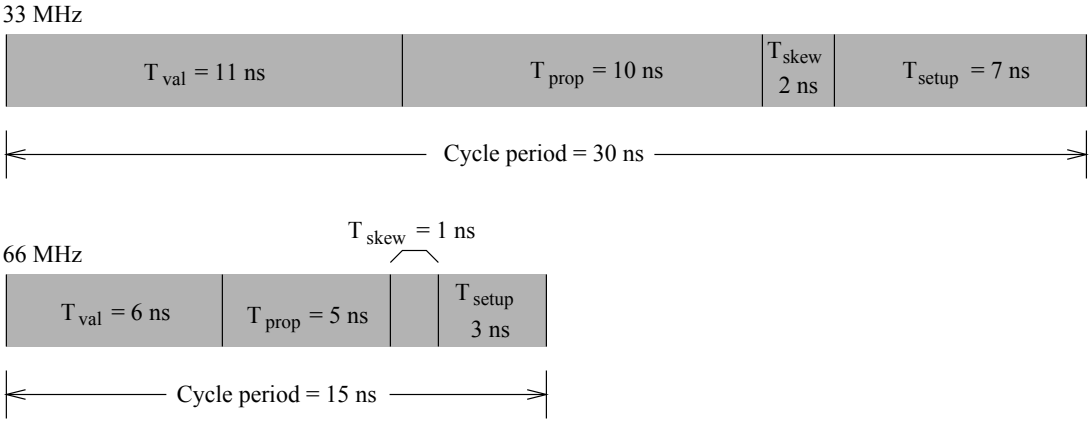


Figure 5.22 Various components of delay in a PCI cycle.

The internal arbiter implements a two-level rotating-priority arbitration policy. Bus masters are divided into high- and low-priority groups. Remember that there can be at most five bus masters—four PCI devices and the 21152—connected to the secondary bus. The entire low-priority group is represented by a single entry in the high-priority group. Within the low-priority group, priority rotates on a round-robin basis.

After initialization, the arbiter will have the four PCI devices in the low-priority group and the 21152 in the high-priority group. That is, there will be two entries in the high-priority group: one entry for the 21152 and the other for the four devices. This means that the 21152 gets the highest priority every other transaction.

Implementation Challenges for the 66 MHZ PCI Bus

In closing our discussion on the PCI bus, we should mention that most PCI buses tend to operate at 33 MHz clock speed. The main reason is that the 66 MHz implementation poses some serious design challenges. To understand this problem, look at the timing of the two buses. The 33 MHz bus cycle of 30 ns leaves about 7 ns of setup time for the target. In Figure 5.22, T_{val} represents the time allowed for the output driver to drive the signal and T_{prop} represents the propagation time. T_{skew} represents the delay from the rising edge of the CLK signal from one component to another.

When we double the clock frequency, all values are cut in half. Since T_{val} is only 6 ns, a 66 MHz bus should be less loaded than the 33 MHz bus. Similarly, propagation and skew times force shorter lengths for the 66 MHz bus. The reduction in T_{setup} is important as we have only 3 ns for the target to respond. This fact is further discussed in a later section that examines the PCI-X bus. As a result of this difficulty, most PCI buses tend to operate at 33 MHz clock speed.

5.6.3 Accelerated Graphics Port (AGP)

With the increasing demand for high-performance video due to applications such as 3D graphics and full-motion video, the PCI bus is reaching its performance limit. In response to these demands, Intel introduced the AGP to exclusively support high-performance 3D graphics and full-motion video applications. The AGP is not a bus in the sense that it does not connect multiple devices. The AGP is a port that precisely connects only two devices: the CPU and video card.

To see the bandwidth demand of a full-motion video, let us look at a 640 $\times$ 480 resolution screen. For true color, we need three bytes per pixel. Thus, each frame requires 640 * 480 * 3 = 920 KB. Full-motion video should use a frame rate of 30 frames/second. Therefore, the required bandwidth is 920 * 30/1000 = 27.6 MB/s. If we consider a higher resolution of 1024 $\times$ 768, it goes up to 70.7 MB/s. We actually need twice this bandwidth when displaying video from hard disks or DVDs. This is due to the fact that the data have to traverse the bus twice: once from the disk to the system memory and again from the memory to the graphics adaptor. The 32-bit, 33 MHz PCI with 133 MB/s bandwidth can barely support this data transfer rate. The 64-bit PCI can comfortably handle the full-motion video but the video data transfer uses half the bandwidth. Since the video unit is a specialized subsystem, there is no reason for it to be attached to a general-purpose bus like the PCI. We can solve many of the bandwidth problems by designing a special interconnection to supply the video data. By taking the video load off the PCI bus, we can also improve performance of the overall system. Intel proposed the AGP precisely for these reasons.

The AGP specification is based on the 66 MHz PCI 2.1 specification. As such, it retains many of the PCI signals. Like the PCI, it uses a 32-bit wide multiplexed address and data bus. However, instead of running at the PCI speed of 33 MHz, AGP runs at full speed of 66 MHz. Thus, the AGP provides a bandwidth of 264 MB/s. The AGP can operate in 2X mode in which it transmits data on both rising and falling edges of the clock. Thus, it can provide up to four times the 33 MHz PCI bandwidth (i.e., 528 MB). How does this compare with the system bus of a Pentium PC? If we consider a 66 MHz motherboard, the bandwidth is 66 $\times$ 8 = 528 MB as the Pentium uses a 64-bit wide bus. Current motherboards support a 100 MHz system bus, which takes the processor system/memory bus bandwidth to about 800 MB. To exploit higher memory bandwidths, the AGP can also run at 4X speed. This takes the bandwidth of the AGP to more than 1 GB/s.

The AGP provides further performance enhancements by using pipelining and sideband addressing. We first discuss the pipelining feature. Pipelining is a common technique designers use to overlap several operations. The AGP uses the pipelined mode of data transfer from memory in order to hide the memory latency. In nonpipelined data transfers (e.g., in the PCI), an operation has to be complete before we can initiate the next one. In pipelined mode, operations can overlap. Pipeline principles are discussed in detail in Chapter 8. The AGP has a special signal called PIPE# to initiate pipelined data transfer.

The AGP uses PCI bus transactions as well as AGP-specific pipelined transactions. In fact, AGP pipelined transactions can be intervened with regular PCI bus transactions. The access

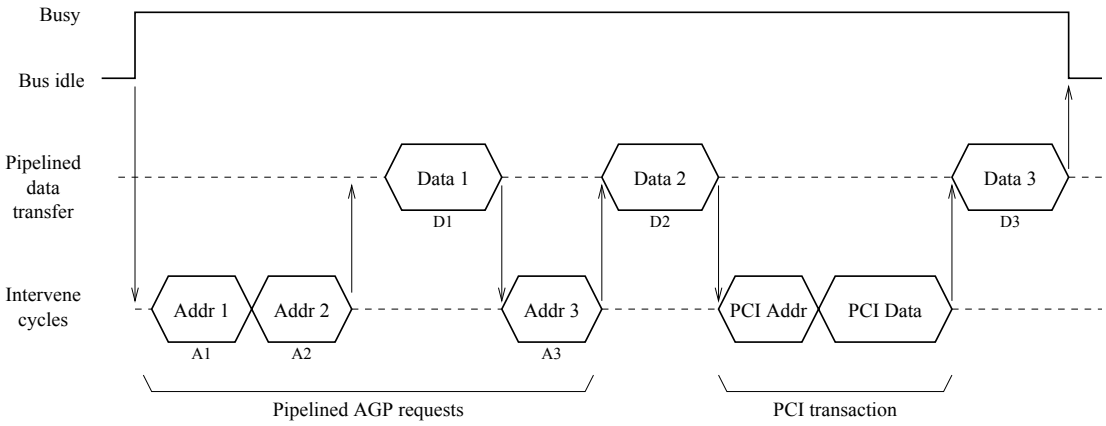


Figure 5.23 AGP pipelined transmission can be interrupted by PCI transactions.

request portion of an AGP transaction uses the AD and C/BE# lines of the bus as in a PCI transaction. However, AGP transactions use the PIPE# rather than the FRAME# signal.

AGP pipelined transmission can be interrupted by PCI transactions as shown in Figure 5.23. In this figure, when the bus is idle, AGP obtains the bus and starts a pipelined transaction. As a part of this transaction, it places the address A1, then address A2. The bus is turned around and after a delay system memory places data D1 that corresponds to address A1. Then the AGP supplies address A3 and the bus is turned around for data D2. After D2, a PCI master gets the bus and the PCI transaction is completed. Once the PCI transaction is done, the interrupted AGP pipeline transaction continues. Finally, after supplying data D3, the bus returns to the IDLE state. Note that PCI transactions should not be intervened by other transactions. Intervention of pipelined transactions is done by bus arbitration signals REQ# and GNT#.

A state diagram can express the bus operations using the four states shown in Figure 5.24. The four states are IDLE, AGP, PCI, and PDT (pipelined data transfer). When the system bus is IDLE, the standard PCI bus transaction can be initiated, as indicated by the transition between the IDLE and PCI states in Figure 5.24. If the AGP requests a pipelined transfer, it will move to the AGP state. The AGP master will output the address (or a sequence of addresses as in Figure 5.23). When the system memory is ready to transmit data, it takes control of the bus and transmits data. This causes a state change from the AGP to the PDT. While in the PDT state, both AGP and PCI requests can interrupt the pipeline transfer, as shown in Figure 5.23. Once the pipelined transfer is complete, the bus returns to the IDLE state.

From Figure 5.23 you can see the impact of the multiplexed address/data bus. For example, we use the address/data bus to send address A3 by interrupting the data transfer. The AGP uses a technique to partially demultiplex this bus. It is known as sideband addressing (SBA). Sideband addressing accelerates data transfers by allowing the graphics controller to issue new addresses and requests while the data are transferred from previous requests on the main address/data bus.

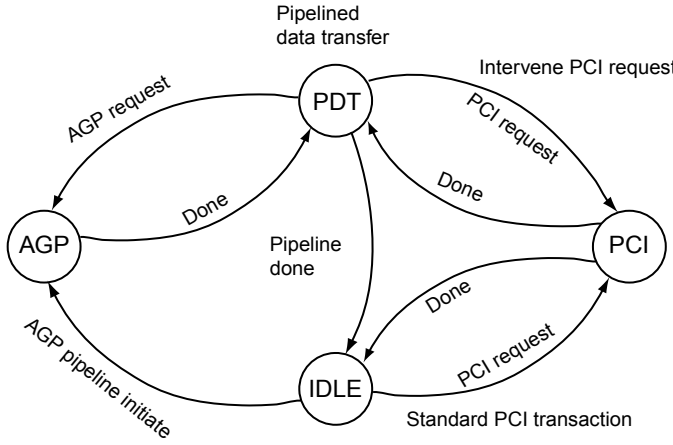


Figure 5.24 State diagram for the AGP bus operations.

In order to keep the cost down, the SBA port is only eight bits wide. More details on this are available from the AGP specification [20].

In summary, the AGP offers the following advantages:

- The AGP provides a peak bandwidth that is four times the PCI bandwidth using features such as pipelined data transfer, sideband addressing, and up to 4X data transmission.
- The AGP enables high-speed direct access to system memory by the graphics controller. This access avoids the need for loading 3D operations data into the video memory.
- Since the video traffic is taken off the PCI bus, there will be less contention of the PCI bus bandwidth by other I/O devices connected to the PCI bus.

Evolution of the AGP goes on! Intel recently announced an advanced AGP (AGP8X) that doubles the data transfer frequency to 533 MHz. Since it still uses a 32-bit wide bus, it can support data rates of about 2 GB/s.

5.6.4 The PCI-X Bus

The PCI bus is reaching its bandwidth limitation with faster I/O buses and networks. Although the PCI bus can operate at a maximum of 66 MHz, most systems use the 33 MHz bus. As mentioned, the 64-bit, 33 MHz PCI bus offers a peak bandwidth of 266 MB/s. The 66 MHz version proposed in PCI 2.2 can theoretically achieve a peak bandwidth of about 528 MB/s; there are several technical design challenges that slowed its implementation. Even this bandwidth is not adequate for future I/O needs. We give a couple of examples to illustrate the need for higher bandwidth.

- *Faster I/O Buses:* With the advent of faster I/O buses, the bandwidth provided by a 64-bit, 66 MHz PCI bus is inadequate to support more than one or two such I/O channels. For

example, the SCSI Ultra360 bus disk drive interface needs 360 MB/s bandwidth. Another example is the proposed 2 Gbit/s Fibre Channel bus.

- *Faster Networks:* On the network side, a four-port gigabit Ethernet NIC can overwhelm even the 64-bit, 66 MHz PCI bus. With the specification of the 10 Gbit/s expected around 2002, the PCI bus cannot handle such networks.

The PCI-X bus leverages existing PCI bus technology to improve the bandwidth from 133 MB/s (32-bit, 33 MHz PCI) to more than 1 GB/s. This is achieved with a 64-bit, 133 MHz PCI-X bus. As mentioned before, even though the PCI 2.2 specifies that PCI can run at the 66 MHz frequency, there are several design challenges (see page 179).

How is PCI-X solving this problem? It uses a register-to-register protocol, as opposed to the immediate protocol implemented by PCI. In the PCI-X register-to-register protocol, the signal sent by the master device is stored in a register until the next clock. In our digital logic terms, the signal is latched into a flip-flop. Thus the receiver has one full clock cycle to respond to the master's request. This makes it possible to increase the frequency to 133 MHz. At this frequency, one clock period corresponds to about 7.5 ns, about the same period allowed for the decode phase (T_{setup}) in the 33 MHz PCI implementation. We get this increase in frequency by adding one additional cycle to each bus transaction. This increased overhead is more than compensated for by the increase in the frequency. For example, on the PCI bus, a transaction with six no-wait-state data phases takes 9 clocks. On a 33 MHz PCI bus, this transaction will take 270 ns to complete. The same transaction takes 10 clocks to complete on a PCI-X bus. At 133 MHz, it takes about 75 ns—about 3.5 times faster.

The PCI-X also gives flexibility by allowing the bus to operate at three different frequencies: 66 MHz, 100 MHz, and 133 MHz. The only restriction is that the PCI-X bus must operate at frequencies higher than 50 MHz. The PCI-X bus supports up to 256 bus segments; each segment can operate at its own frequency. System designers can trade off performance for connectivity. Figure 5.25 shows three segments operating at three frequencies. As shown in this figure, a PCI-X bus may operate at 133 MHz with one slot. It can have two connection slots by operating at 100 MHz, or have four slots at 66 MHz, as shown in this figure. The reason for allowing additional slots at lower frequency is that the signal can have more propagation time to go from slot to slot. In contrast, the conventional PCI can have only two slots when operating at 66 MHz.

In addition to the register-to-register protocol that gives an extra clock cycle to decode, PCI-X incorporates several enhancements to improve bus efficiency. Below, we discuss these features briefly [10]:

- *Attribute Phase:* The PCI-X protocol has a new transaction phase called the attribute phase that uses a 36-bit attribute field to describe the bus transaction in more detail than the conventional PCI protocol. The details provided include the transaction size, relaxed transaction ordering, and the identity of the transaction initiator. The conventional PCI handles requests from multiple PCI devices in the order they were received. PCI-X allows reorder of transactions based on the resources available and the importance of a transaction. For example, audio and video transactions can have a higher priority so that timing

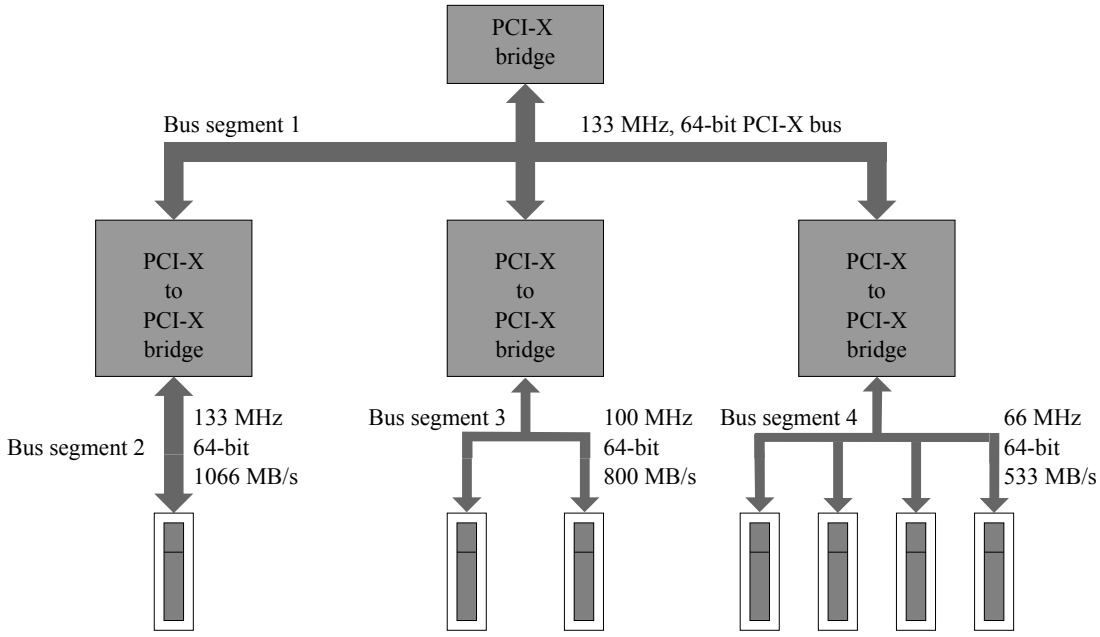


Figure 5.25 PCI-X supports up to 256 segments; each segment can operate in 66 MHz, 100 MHZ, or 133 MHz.

constraints can be satisfied. Clearly, transaction identification makes this reordering possible.

The transaction size (byte count) is explicit in PCI-X. The conventional PCI protocol does not specify this information as part of a transaction. Under conventional PCI, host-to-PCI or PCI-to-PCI bridges use a default transaction size of either one or two cache lines. Specification of transaction byte size allows better management of buffers and other resources.

- *Split Transaction Support:* The conventional PCI transaction treats a request and the corresponding reply as a transaction. It does allow delayed transactions but uses polling to see if the target is ready to supply the requested data. In contrast, PCI-X splits the request–reply into two transactions. The initiator of the transaction can send the request and, after receiving the acknowledgment from the receiver, can work on other information. The receiver, when ready to send the data, initiates a transaction to send the data. This split transaction mode improves utilization of the bus.
- *Optimized Wait States:* In a conventional PCI, if a receiver cannot supply data, wait states are introduced to extend the transaction cycle. The bus is held by the transaction during these additional wait states. In a PCI-X, the split transaction mode allows the bus to be released for other transactions. This leads to improved bus utilization.

- *Standard Block Size Movement:* PCI-X allows transactions to disconnect only on 128-bit boundaries. This allows for efficient pipelined data movement, particularly in the movement of cache line data.

A PCI-X system can maintain backward compatibility with conventional PCI cards. A PCI-X adaptor can operate in a PCI system, and vice versa. A PCI-X system can switch between conventional PCI and PCI-X modes on a segment-by-segment basis, depending on the adaptors connected to the segment. If any of the adaptors is a conventional PCI one, the entire segment operates in the PCI mode. The operating frequency is adjusted to the slowest device on the segment.

The performance of the PCI-X reported in [10] suggests that we can get up to 34% improvement in throughput when using 4 KB reads on a 64-bit, 66 MHz bus. When we read 512-byte blocks, the improvement in throughput reduces to about 14%.

5.6.5 The PCMCIA Bus

The PCMCIA bus was motivated by the desire to have a standard for memory cards. The Personal Computer Memory Card International Association (PCMCIA) released its first standard (Release 1.0) in 1990. This standard is commonly called the PC Card standard. Release 2.0 added support for I/O devices. This is particularly useful for laptops to interface expansion I/O devices such as hard drives, modems, and network cards.

PCMCIA cards have a small form factor. All cards have a length of 85.5 mm and width of 54 mm (the same as your credit card size). The thickness of the card varies depending on the card type. Three card types are supported to accommodate various types of I/O devices.

- A type I card has a thickness of 3.3 mm. These cards are normally used for memory units such as RAM, ROM, and FLASH memories.
- Type II cards are 5 mm thick. These cards are used to interface I/O devices such as modems and network interface cards.
- Type III cards are slightly more than twice as thick as type II cards (10.5 mm thick). Type III cards accommodate I/O devices such as hard drives that require more physical space.

Type I and II cards can also be extended to accommodate large external connectors. These cards maintain the same internal dimensions as the standard type I and II cards.

All cards use the same 68-pin connector/socket pair. PC cards can be connected to a variety of host buses. PC cards can use either 5 volts (standard) or 3.3 volts (low voltage). The standard also includes a future nonspecified low voltage, referred to as X.X volts. For details on this, see [2] and also Figure 5.26.

PCMCIA supports three address spaces: common, attribute, and I/O address spaces. Common address space is used for memory expansion and the attribute address space is used for automatic configuration. Each address space is 64 MB, which implies that PCMCIA uses 26 address lines. PC cards use a 16-bit data bus. The PC Card standard also defines a 32-bit standard called CardBus. Here we focus on the PC Card standard. For details on the CardBus standard, see [7].

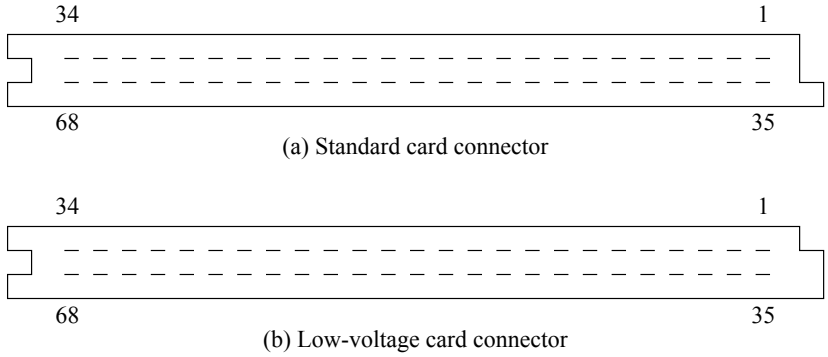


Figure 5.26 PCMCIA connectors: Sockets are keys such that a low-voltage card cannot be inserted into a 5 V standard socket.

To explain the basic operation of the PC card, we organize our discussion into two subsections. The first subsection describes the memory interface. The following subsection deals with the I/O card interface.

Memory Interface

The memory interface consists of several groups of signals. We give a brief description of these signals next.

Address signals: This group consists of address lines and two card enable signals.

- *Address Lines (A0 to A25):* These 26 address lines support a total of 64 MB of address space. For 16-bit memories, A1 to A25 lines specify the address of a 16-bit word. The lower address line A0 specifies the odd or even byte of the word.
- *Card Enable Signals (CE1#, CE2#):* These two signals are somewhat similar to the chip select signal discussed in Chapter 16. These two low-active signals control how the data are transferred. When both these signals are high (i.e., not asserted), no data transfer takes place. In this case, the data path floats (high-Z state). When CE1# is asserted, data transfer takes place on the lower data path (D0 to D7). Asserting CE2# indicates that the data transfer takes place on the upper data path (D8 to D15). When both signals are low, 16-bit data transfer takes place on the data path (D0 to D15).

  When CE1# = 1 and CE2# = 0, only the odd byte is transferred on the upper data path. This combination is valid in the 16-bit mode only. In this case, the value of the address bit A0 is ignored. When CE1# = 0, CE2# = 1, and A0 = 0, even byte transfer takes place on the lower data path (D0 to D7). This combination is valid in both 8- and 16-bit modes. However, in the 8-bit mode, A0 = 1 is also allowed. This causes the transfer of odd bytes on the lower data path (D0 to D7). PCMCIA supports only 16-bit memory cards.

Transaction Signals: As stated before, there are two memory address spaces: common and attribute memory. PC Card specifies four types of memory transactions: common memory read, common memory write, attribute memory read, and attribute memory write.

- *Data Lines (D0 to D15):* These 16 data lines are used to transfer data. As discussed, the two card enable signals determine whether odd byte, even byte, or a word is transferred on the data path.
- *Output Enable (OE#):* This is the memory read signal. This signal should be low when reading from a PC card.
- *Write Enable (WE#):* This is the memory write signal. This signal should be low when writing to a PC card. If the PC card memory requires special programming voltage (e.g., for a flash memory card), this line is used as the program command signal.
- *Wait Signal (WAIT#):* This signal serves the same purpose as the wait signal in system buses. When a normal transaction requires more time, the WAIT# signal can be used to extend the transaction cycle.
- *Register Select (REG#):* This signal selects whether the common memory or the attribute memory should be the target. To access the attribute memory, this signal should be low. A high REG# signal indicates that a read/write operation is targeted to the common memory.

PC Memory Card Status Signals: Memory cards typically have a write protect switch, just like your floppy disk. When this switch is on, PC card memory becomes read-only. If the memory is volatile, there is also space for a battery to prevent losing data when the memory card is removed from the socket. A memory is said to be volatile if it requires power to retain its contents. Flash memory is an example of a nonvolatile memory.

PCMCIA status signals are used to report the status of the PC card. This group of signals includes the following:

- *Card Detect Signals (CD1#, CD2#):* These two signals indicate the presence of a PC card in the socket. The CD1# is a low-active signal that indicates that one side of the PC card is making contact with the socket. The CD2# signal indicates that the other side is making contact. The interpretation of these two signals is shown below:

CD2#	CD1#	Interpretation
0	0	Card properly inserted
0	1	Card improperly inserted
1	0	Card improperly inserted
1	1	No card inserted

- *Ready/Busy Signal (READY):* A high on this signal indicates that the card is ready to be accessed. When the card is busy executing a command or performing initialization, the READY signal is low.

- *Write Protect (WP):* This signal gives the status of the write protect switch on the memory card.
- *Battery Voltage Detect Signals (BVD1, BVD2):* These two signals indicate the status of the PC memory card battery. If BVD1 is low, the battery cannot maintain the data integrity. When BVD1 is high, the BVD2 value indicates whether the battery is in good condition or nearing its replacement level. The interpretation of these two signals is summarized below:

BVD2	BVD1	Interpretation
0	0	Battery cannot maintain data integrity
0	1	Battery replacement warning
1	0	Battery cannot maintain data integrity
1	1	Battery is in good condition

As an example of a transaction cycle, we show the common memory read cycle in Figure 5.27. Each transaction cycle consists of three phases: a setup phase to allow time for the input signals to settle; a command execution phase to perform the action requested (e.g., memory read); and a hold phase for the target to read the data. Since we are reading a 16-bit word, CE1#, CE2#, and A0 signals are all low. And, as we are reading from the common memory, the REG# signal should be high. The output enable signal OE# is asserted after the setup time to indicate that this is a read operation. For the same reason, the write enable signal WE# is high. This transaction introduces no wait states as the WAIT# signal is high.

I/O Interface

I/O devices use either a type II or III card. To interface I/O cards, some memory interface signals are changed. In particular, some reserved pins in the memory interface are used for I/O purposes. Furthermore, four memory interface signals (READY, WP, BVD1, and BVD2) are assigned different meanings. Here is a summary of the new signals for the I/O interface:

- *I/O Read and Write (IORD#, IOWR#):* These signals use pins 44 and 45, which are reserved in the memory interface. They serve to define the transaction type as I/O read or write. Thus, there can be six types of transactions as shown below:

Transaction type	REG#	WE#	OE#	IORD#	IOWR#
Common memory read	1	1	0	1	1
Common memory write	1	0	1	1	1
Attribute memory read	0	1	0	1	1
Attribute memory write	0	0	1	1	1
I/O read	0	1	1	0	1
I/O write	0	1	1	1	0

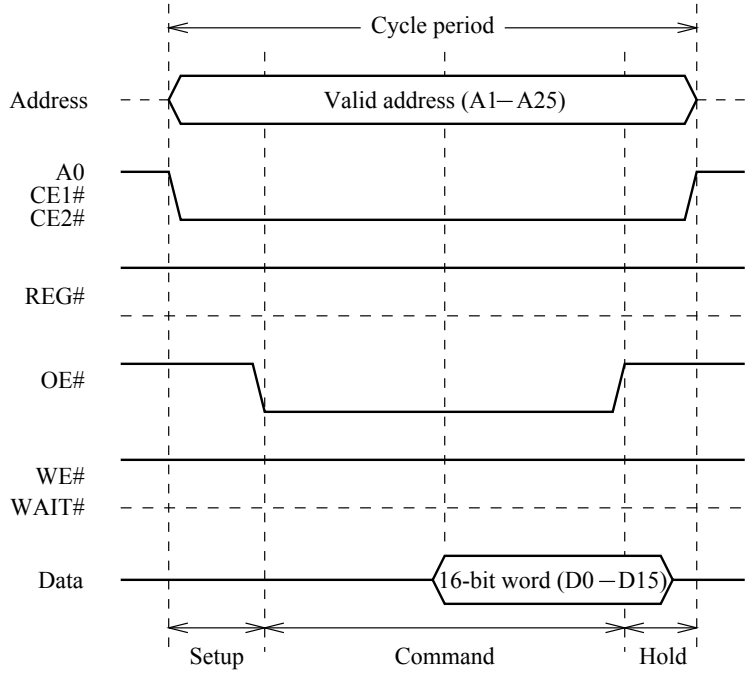


Figure 5.27 A no-wait state 16-bit word common memory read cycle.

- *Interrupt Request (IREQ#):* This signal replaces the READY signal from the memory interface. The PC card asserts this signal to indicate that it requires interrupt service. We discuss interrupts in Chapter 20.
- *I/O Size Is 16 Bits (IOIS16#):* This replaces the write protect (WP) memory interface signal. When asserted, this signal indicates that the I/O is a 16-bit device. A high on this signal indicates an 8-bit I/O device.
- *System Speaker Signal (SPKR#):* This line sends the audio signal to the system speaker. It replaces the BVD2 signal in the memory interface.
- *I/O Status Change (STSCHG#):* This signal replaces the BVD1 memory interface signal. In a pure I/O PC card, we do not normally require this signal. However, in multifunction PC cards containing memory and I/O functions, this signal is needed to report the status-signals removed from the memory interface (READY, WP, BVD1, and BVD2). A configuration register (called the pin replacement register) in the attribute memory maintains the status of the signals removed from the memory interface. For example, since BVD signals are removed, this register keeps the BVD information to report the status of the battery. When a status change occurs, this signal is asserted. The host can read the pin replacement register to get the status.

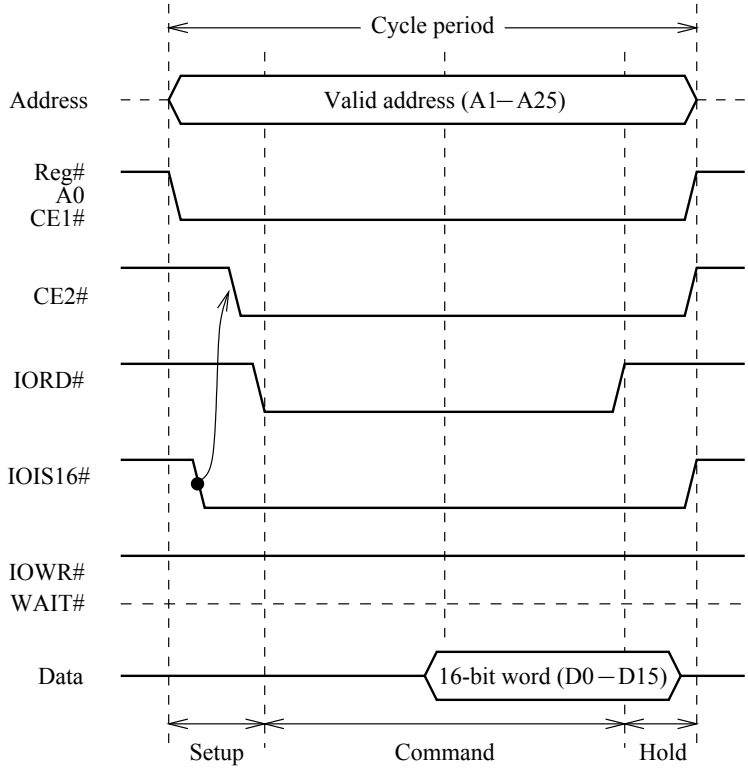


Figure 5.28 A no-wait state 16-bit word I/O read cycle: To read only a byte of data, we have to make the IOIS16# high and A0 identifies the byte (odd or even) to read.

An example of an I/O read transaction is shown in Figure 5.28. It is similar to the memory read transaction. Typically, at the start of an I/O transaction, the addressing mode defaults to the 8-bit mode. The three signals—REG#, A0, and CE1#—are asserted. When the IOIS16# signal is asserted to indicate that this is a 16-bit I/O operation, the CE2# signal is asserted. The remaining signals are straightforward to understand. If it is an 8-bit I/O read operation, the IOIS16# and CE2# signals remain high. Then, as discussed earlier, the A0 address bit determines whether the odd (A0 = 1) or even (A0 = 0) byte is transferred.

5.7 Summary

A bus is the link that connects various system components. In a typical computer system, we find a hierarchy of buses: a memory bus, a local bus, a cache bus, and one or more expansion buses. Each type of bus has several restrictions depending on the purpose of the bus. Bus designers have to decide on several parameters such as bus width, bus type, bus clocking, type of bus operations supported, and bus arbitrating mechanism.

Bus width mainly refers to data bus and address bus widths. Higher data bus width increases the data transfer rate. For example, the Pentium uses a 64-bit data bus, whereas the Itanium uses a 128-bit data bus. Therefore, if all other parameters are the same, we can double the bandwidth in the Itanium relative to the Pentium processor.

The address bus width determines the memory addressing capacity of the system. Typically, 32-bit processors such as the Pentium use 32-bit addresses, and 64-bit processors use 64-bit addresses. Although the 4 GB address space provided by 32-bit processors seems large for a PC, servers find this address space rather restrictive.

Bus type refers to dedicated or multiplexed buses. These offer a tradeoff between cost and bandwidth. Dedicated buses are expensive but provide higher bandwidth compared to multiplexed buses. Bus clocking refers to timing the bus activities. We have discussed two basic types: synchronous and asynchronous. In the synchronous bus, all activities are synchronized to a common bus clock. Asynchronous buses, on the other hand, use full handshaking by means of special synchronization signals. Asynchronous buses are more flexible and can handle heterogeneous devices (i.e., devices that operate at different speeds). Most buses, however, are synchronous as it is easier to build a synchronous bus system.

All buses support basic bus operations such as the memory read, write, and I/O read and write. Most buses also support additional operations such as the atomic read-modify-write and block transfers.

Bus systems typically have more than one bus master. Thus, we need a bus arbitration mechanism for bus allocation. We have discussed several dynamic bus arbitration schemes. Bus arbiters can be centralized or distributed. Each type of bus arbiter can be built using daisy-chaining or independent request lines. We can also use a hybrid scheme as in the VME bus.

In the last section, we have presented details on several example buses, the ISA, PCI, PCI-X, and PCMCIA. We have also discussed the graphics port AGP that provides higher bandwidth dedicated to 3D graphics applications.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Accelerated graphics port (AGP)
- Asynchronous bus
- Block transfer
- Bus arbitration
- Bus operations
- Bus types
- Bus width
- External buses
- ISA bus
- PC card bus
- PCI bus
- PCI-X bus
- PCMCIA bus
- Synchronous bus
- System bus design issues
- Wait states

5.8 Web Resources

Details on PCI and PCI-X buses are available at `www.pcisig.com`. Compaq also maintains information on the PCI-X bus at `http://www5.compaq.com/products/servers/technology/pci-x-enablement.html`. You can also search `www.compaq.com` for details.

Details on the AGP are available at `www.agpforum.org`. Intel also maintains information on the AGP at `http://developer.intel.com/technology/agp`.

Details on the PC Card (PCMCIA) bus are available at `www.pc-card.com`.

5.9 Exercises

5–1 What is a bus transaction?

5–2 What is a bus operation? How does this relate to the bus transaction?

5–3 What is the significance of address bus width?

5–4 What is the significance of data bus width?

5–5 We have discussed two bus types: dedicated and multiplexed. Discuss the advantages and disadvantages of these two bus types.

5–6 Discuss the advantages and disadvantages of synchronous and asynchronous buses. Then explain why most system buses are synchronous.

5–7 We have discussed synchronous and asynchronous buses. Why is it critical to select the right clock frequency only in synchronous buses?

5–8 What is the purpose of the READY signal in a synchronous bus? Do we need this signal in an asynchronous bus?

5–9 What is the main advantage of block transfer transactions?

5–10 Discuss the advantages of dynamic bus arbitration over a static arbitration scheme.

5–11 In the dynamic bus arbitration section, we discussed fair allocation policies. What happens if an allocation policy is not fair?

5–12 What are the advantages of preemptive bus release policies over their nonpreemptive cousins?

5–13 What is the main advantage of demand-based bus release policy over transaction-based release policy?

5–14 Compare the centralized and distributed implementations of dynamic bus arbitration policies.

5–15 What are some of the problems with the daisy-chaining scheme?

5–16 What are advantages of the hybrid bus arbitration scheme used in the VME bus?

5–17 We have stated that the ISA bus gets stepmotherly treatment in the bus hierarchy. Specifically, discuss the reasons for using the ISA bus for slower I/O devices. Is there an alternative bus that could replace it?

5–18 Explain why the PCI bus uses the multiplexed address/data bus.

5–19 What is the purpose of Byte Enable lines in the PCI bus?

5–20 The PCI bus uses IRDY# and TRDY# signals to control data flow. Then why does it require the FRAME# signal?

5–21 Describe the PCI bus arbitration mechanism.

5–22 We have stated that using a PCI-to-PCI bridge improves system performance. We have used the Intel 21152 chip to illustrate the specifics of such a bridge. Discuss why using the 21152 bridge improves performance.

5–23 What are some of the difficulties in implementing the 66 MHz PCI bus? How is the PCI-X bus overcoming these difficulties?

5–24 Explain the rationale for proposing the accelerated graphics port.

5–25 What are the reasons for allowing PCI requests to interrupt AGP pipelined requests?

5–26 What is the need for the STSCHG# signal in the PC Card bus?

Chapter 6

Processor Organization and Performance

Objectives

- To introduce processor design issues;
- To describe flow control mechanisms used in RISC and CISC processors;
- To present details on microprogrammed control;
- To discuss performance issues.

This chapter looks at processor design and performance issues. We start our discussion with the number of addresses used in processor instructions. This is an important design characteristic that influences the instruction set design. This section also describes the load/store architecture used by RISC and vector processors.

Flow control deals with branching, procedure calls, and interrupts. It is an important aspect that affects the performance of the overall system. In Section 6.3, we discuss the general principles used to efficiently implement branching and procedure invocation mechanisms. We discuss the interrupt mechanism in Chapter 20. Instruction set design issues are discussed in Section 6.4.

The next section focuses on how the instructions are executed in hardware. A datapath is used to execute instructions. We have already presented an example datapath in Chapter 1. To execute instructions, we need to supply appropriate control signals for the datapath. We can generate these control signals in one of two basic ways: we can design our hardware to directly generate the control signals, or use what is known as the microprogram to issue necessary control signals to the underlying hardware. Typically, RISC processors use the direct hardware execution method, as their instructions are simple. The CISC processors, on the other hand,

depend on microprogrammed control in order simplify the hardware. These details are covered in Section 6.5.

Section 6.6 discusses how the performance of the CPU can be quantified and measured. This section also describes some sample performance benchmarks from the SPEC consortium. The chapter concludes with a summary.

6.1 Introduction

Processor designs can be broadly divided into three types: RISC, CISC, and vector processors. We briefly mentioned CISC and RISC processors in Chapter 1. We give a detailed discussion of the RISC and CISC processors in Chapter 14. Vector processors exploit pipelining to the fullest extent possible. We present details on vector processors in Chapter 8.

This chapter deals with three processor-related topics: instruction set design issues, microprogrammed control, and performance issues. In Chapter 1, we introduced the two main components of the processor: the datapath and control. Although the processors designed in the 1970s consisted of these two components, current processors have many more onchip entities such as caches and pipelined execution units. These features are discussed in other chapters of this book. In this chapter, we mainly focus on the datapath and control. As well, we look at the instruction set architecture and performance issues.

One of the characteristics that influences the ISA is the number of addresses used in the instructions. Since typical operations require two operands, we need three addresses: two source addresses to specify the two input operands and a destination address to indicate where the result should be stored. Most processors specify three addresses. We can reduce the number of addresses to two by using one address to specify a source address as well as the destination address. The Pentium uses two-address format instructions. It is also possible to have instructions that use one or even zero address. The one-address machines are called accumulator machines and the zero-address machines are called stack machines. The relative advantages and drawbacks of these schemes are discussed in Section 6.2.

RISC processors tend to use a special architecture known as the load/store architecture. In this architecture, special load and store instructions are used to move data between the processor's internal registers and memory. All other instructions require the necessary operands to be present in the registers. Vector processors originally used the load/store architecture. We discuss vector processors in Chapter 8. The load/store architecture is described in Section 6.2.6.

Instruction set design involves several other issues. The addressing mode is another important aspect that specifies where the operands are located. CISC processors typically allow a variety of addressing modes, whereas RISC processors support only a couple of addressing modes. The addressing modes and number of addresses directly influence the instruction format. CISC processors use variable-length instructions whereas the RISC processors use fixed-length instructions. The difference is mainly due to the fact that CISC processors use from simple to complex addressing modes. These and other issues such as the instruction and operand types are discussed in Section 6.4.

As mentioned in Section 1.4, the datapath provides the basic hardware to execute instructions. We have given an example datapath on page 16. This datapath uses three internal buses. The performance of the processor depends on the number of internal buses used. To execute instructions on the datapath, we have to provide appropriate control signals. These control signals can be generated by implementing a finite state machine in hardware. Hardware implementation is used for simple, well-structured instructions. RISC processors take this approach. CISC processors use a software approach that uses a microprogram for this purpose. Processors like the Pentium use this approach. We discuss microprogrammed control in detail in Section 6.5.

Performance quantification is very important for both designers and users. Designers need a way to compare performance of various designs in order to select an appropriate design. Users need to know the performance in order to buy the best system that meets their needs. The processor is a critical component that influences overall system performance. We discuss processor performance metrics and standards in Section 6.6.

6.2 Number of Addresses

One of the characteristics of the ISA that shapes the architecture is the number of addresses used in an instruction. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Most recent processors use three addresses. However, it is possible to design systems with two, one, or even zero addresses. In the rest of this section, we give a brief description of these four types of machines. In Section 6.2.5, we discuss their advantages and disadvantages.

6.2.1 Three-Address Machines

In three-address machines, instructions carry all three addresses explicitly. The RISC processors we discuss in Chapters 14 and 15 use three addresses. Table 6.1 gives some sample instructions of a three-address machine.

In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
mult  T,C,D    ; T = C*D
add   T,T,B    ; T = B + C*D
sub   T,T,E    ; T = B + C*D - E
add   T,T,F    ; T = B + C*D - E + F
add   A,T,A    ; A = B + C*D - E + F + A
```

Table 6.1 Sample three-address machine instructions

Instruction	Semantics
`add dest,src1,src2`	Adds the two values at `src1` and `src2` and stores the result in `dest` M(dest) = [src1] + [src2]
`sub dest,src1,src2`	Subtracts the second source operand at `src2` from the first at `src1` and stores the result in `dest` M(dest) = [src1] − [src2]
`mult dest,src1,src2`	Multiplies the two values at `src1` and `src2` and stores the result in `dest` M(dest) = [src1] * [src2]

We use the notation that each variable represents a memory address that stores the value associated with that variable. This translation from symbol name to the memory address is done by using a symbol table. We discuss the function of the symbol table in Section 9.3.3 (see page 330).

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary `T` and in the last one, it is `A`. This is the motivation for using two addresses, as we show next.

6.2.2 Two-Address Machines

In two-address machines, one address doubles as a source and destination. Usually, we use `dest` to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. We discuss Pentium processor details in the next chapter. Table 6.2 gives some sample instructions of a two-address machine.

On these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
load  T,C    ; T = C
mult  T,D    ; T = C*D
add   T,B    ; T = B + C*D
sub   T,E    ; T = B + C*D - E
add   T,F    ; T = B + C*D - E + F
add   A,T    ; A = B + C*D - E + F + A
```

Table 6.2 Sample two-address machine instructions

Instruction	Semantics
`load dest,src`	Copies the value at `src` to `dest` M(dest) = [src]
`add dest,src`	Adds the two values at `src` and `dest` and stores the result in `dest` M(dest) = [dest] + [src]
`sub dest,src`	Subtracts the second source operand at `src` from the first at `dest` and stores the result in `dest` M(dest) = [dest] − [src]
`mult dest,src`	Multiplies the two values at `src` and `dest` and stores the result in `dest` M(dest) = [dest] * [src]

Since we use only two addresses, we use a load instruction to first copy the `C` value into a temporary represented by `T`. If you look at these six instructions, you will notice that the operand `T` is common. If we make this our default, then we don't need even two addresses: we can get away with just one address.

6.2.3 One-Address Machines

In the early machines, when memory was expensive and slow, a special set of registers was used to provide an input operand as well as to receive the result from the ALU. Because of this, these registers are called the *accumulators*. In most machines, there is just a single accumulator register. This kind of design, called *accumulator machines*, makes sense if memory is expensive.

In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to store the result in memory: this reduces the need for larger memory as well as speeds up the computation by reducing the number of memory accesses. A few sample accumulator machine instructions are shown in Table 6.3.

In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

Table 6.3 Sample accumulator machine instructions

Instruction	Semantics
`load addr`	Copies the value at address `addr` into the accumulator accumulator = [`addr`]
`store addr`	Stores the value in the accumulator at the memory address `addr` M(`addr`) = accumulator
`add addr`	Adds the contents of the accumulator and value at address `addr` accumulator = accumulator + [`addr`]
`sub addr`	Subtracts the value at memory address `addr` from the contents of the accumulator accumulator = accumulator − [`addr`]
`mult addr`	Multiplies the contents of the accumulator and value at address `addr` accumulator = accumulator * [`addr`]

```
load    C    ; load C into the accumulator
mult    D    ; accumulator = C*D
add     B    ; accumulator = C*D+B
sub     E    ; accumulator = C*D+B-E
add     F    ; accumulator = C*D+B-E+F
add     A    ; accumulator = C*D+B-E+F+A
store   A    ; store the accumulator contents in A
```

6.2.4 Zero-Address Machines

In zero-address machines, locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in-first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. We discuss the stack later in this book (see Section 10.1 on page 388).

All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack. Table 6.4 gives some sample instructions for the stack machines.

Notice that the first two instructions are not zero-address instructions. These two are special instructions that use a single address and are used to move data between memory and stack.

Table 6.4 Sample stack machine instructions

Instruction	Semantics
`push addr`	Places the value at address `addr` on top of the stack push([addr])
`pop addr`	Stores the top value on the stack at memory address `addr` M(`addr`) = pop
`add`	Adds the top two values on the stack and pushes the result onto the stack push(pop + pop)
`sub`	Subtracts the second top value from the top value of the stack and pushes the result onto the stack push(pop − pop)
`mult`	Multiplies the top two values in the stack and pushes the result onto the stack push(pop * pop)

All other instructions use the zero-address format. Let's see how the stack machine translates the arithmetic expression we have seen in the previous subsections. In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
push  E     ; <E>
push  C     ; <C, E>
push  D     ; <D, C, E>
mult        ; <C*D, E>
push  B     ; <B, C*D, E>
add         ; <B+C*D, E>
sub         ; <B+C*D-E>
push  F     ; <F, B+D*C-E>
add         ; <F+B+D*C-E>
push  A     ; <A, F+B+D*C-E>
add         ; <A+F+B+D*C-E>
pop   A     ; < >
```

On the right, we show the state of the stack after executing each instruction. The top element of the stack is shown on the left. Notice that we pushed `E` early because we need to subtract it from (`B+C*D`).

Stack machines are implemented by making the top portion of the stack internal to the processor. This is referred to as the *stack depth*. The rest of the stack is placed in memory. Thus, to access the top values that are within the stack depth, we do not have to access the memory. Obviously, we get better performance by increasing the stack depth. Examples of stack-oriented machines include the earlier Burroughs B5500 system and the HP3000 from Hewlett–Packard. Most scientific calculators also use stack-based operands. For more details on the HP3000 architecture, see [16].

6.2.5 A Comparison

Each of the four address schemes discussed in the previous subsections has certain advantages. If you count the number of instructions needed to execute our example C statement, you notice that this count increases as we reduce the number of addresses. Let us assume that the number of memory accesses represents our performance metric: the lower the number of memory accesses, the better.

In the three-address machine, each instruction takes four memory accesses: one access to read the instruction itself, two for getting the two input operands, and a final one to write the result back in memory. Since there are five instructions, this machine generates a total of 20 memory accesses.

In the two-address machine, each arithmetic instruction still takes four accesses as in the three-address machine. Remember that we are using one address to double as a source and destination address. Thus, the five arithmetic instructions require 20 memory accesses. In addition, we have the load instruction that requires three accesses. Thus, it takes a total of 23 memory accesses.

The count for the accumulator machine is better as the accumulator is a register and reading or writing to it, therefore, does not require a memory access. In this machine, each instruction requires just two accesses. Since there are seven instructions, this machine generates 14 memory accesses.

Finally, if we assume that the stack depth is sufficiently large so that all our push and pop operations do not exceed this value, the stack machine takes 19 accesses. This count is obtained by noting that each `push` or `pop` instruction takes two memory accesses, whereas the five arithmetic instructions take one memory access each.

This comparison leads us to believe that the accumulator machine is the fastest. The comparison between the accumulator and stack machines is fair because both machines assume the presence of registers. However, we cannot say the same for the other two machines. In particular, in our calculation, we assumed that there are no registers on the three- and two-address machines. If we assume that these two machines have a single register to hold the temporary `T`, the count for the three-address machine comes down to 12 memory accesses. The corresponding number for the two-address machine is 13 memory accesses. As you can see from this simple example, we tend to increase the number of memory accesses as we reduce the number of addresses.

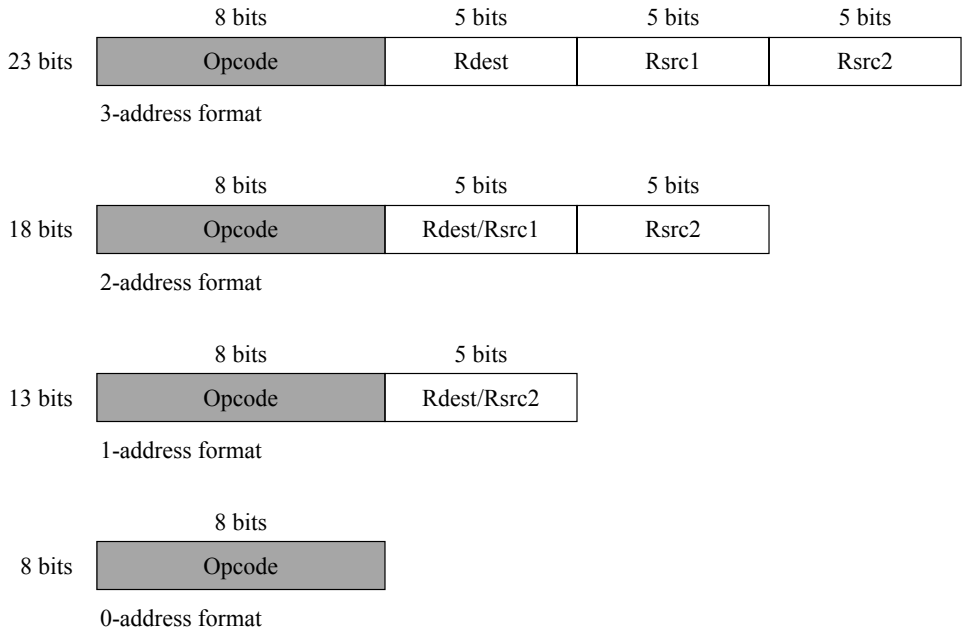


Figure 6.1 Instruction sizes for the four formats: This format assumes that the operands are located in registers.

There are still problems with this comparison. The reason is that we have not taken the size of the instructions into account. Since the stack machine instructions do not need to specify the operand addresses, each instruction takes fewer bits to encode than an instruction in the three-address machine. Of course, the difference between the two depends on several factors including how the addresses are specified and whether we allow registers to hold the operands. We discuss these issues shortly.

Figure 6.1 shows the size of the instructions when the operands are available in the registers. This example assumes that the processor has 32 registers like the MIPS processor and the opcode takes 8 bits. The instruction size varies from 23 bits to 8 bits.

In practice, most systems use a combination of these address schemes. This is obvious from our stack machine. Even though the stack machine is a zero-address machine, it uses load and store instructions that specify an address. Some processors impose restrictions on where the operands can be located. For example, the Pentium allows only one of the two operands to be located in memory. Part V provides details on the Pentium instruction set.

RISC processors take the Pentium's restriction further by allowing most operations to work on the operands located in the processor registers. These processors provide special instructions to move data between the registers and memory. This architecture is called the load/store architecture, which is discussed next.

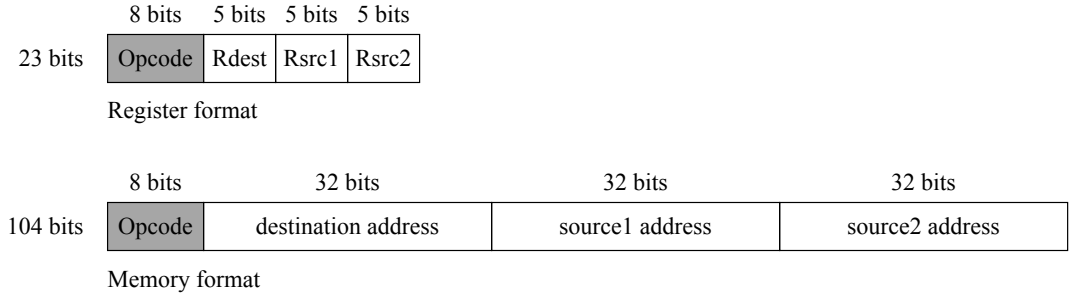


Figure 6.2 A comparison of the instruction size when the operands are in registers versus memory.

6.2.6 The Load/Store Architecture

In the load/store architecture, instructions operate on values stored in internal processor registers. Only load and store instructions move data between the registers and memory. RISC machines as well as vector processors use this architecture, which reduces the size of the instruction substantially. If we assume that addresses are 32 bits long, an instruction with all three operands in memory requires 104 bits whereas the register-based operands require instructions to be 23 bits, as shown in Figure 6.2.

We discuss RISC processors in more detail in Part VI. We look at the vector processors in Chapter 8. Table 6.5 gives some sample instructions for the load/store machines.

In these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
load   R1,B      ; load B
load   R2,C      ; load C
load   R3,D      ; load D
load   R4,E      ; load E
load   R5,F      ; load F
load   R6,A      ; load A
mult   R2,R2,R3  ; R2 = C*D
add    R2,R2,R1  ; R2 = B + C*D
sub    R2,R2,R4  ; R2 = B + C*D - E
add    R2,R2,R5  ; R2 = B + C*D - E + F
add    R2,R2,R6  ; R2 = B + C*D - E + F + A
store  A,R2      ; store the result in A
```

Each load and store instruction takes two memory accesses: one to fetch the instruction and the other to access the data value. The arithmetic instructions need just one memory access to fetch the instruction, as the operands are in registers. Thus, this code takes 19 memory accesses.

Table 6.5 Sample load/store machine instructions

Instruction	Semantics
`load Rd,addr`	Loads the `Rd` register with the value at address `addr` Rd = [addr]
`store addr,Rs`	Stores the value in `Rs` register at address `addr` (addr) = Rs
`add Rd,Rs1,Rs2`	Adds the two values in `Rs1` and `Rs2` registers and places the result in `Rd` register Rd = Rs1 + Rs2
`sub Rd,Rs1,Rs2`	Subtracts the value in `Rs2` from that in `Rs1` and places the result in `Rd` register Rd = Rs1 − Rs2
`mult Rd,Rs1,Rs2`	Multiplies the two values in `Rs1` and `Rs2` and places the result in `Rd` register Rd = Rs1 * Rs2

Note that the elapsed execution time is not directly proportional to the number of memory accesses. Overlapped execution reduces the execution time for some processors. In particular, RISC processors facilitate this overlapped execution because of their load/store architecture. We give more details in Chapter 8.

In the RISC code, we assumed that we have six registers to load the values. However, you don't need this many registers. For example, once the value in R3 is used, we can reuse this register. Typically, RISC machines tend to have many more registers than CISC machines. For example, the MIPS processor has 32 registers and the Intel Itanium processor has 128 registers. Both are RISC processors and are covered in Part VI.

6.2.7 Processor Registers

Processors have a number of registers to hold data, instructions, and state information. We can classify the processors based on the structure of these registers and how the processor uses them. Typically, we can divide the registers into general-purpose or special-purpose registers. Special-purpose registers can be further divided into those that are accessible to the user programs and those reserved for the system use. The available technology largely determines the structure and function of the register set.

The number of addresses used in instructions partly influences the number of data registers and their use. For example, stack machines do not require any data registers. However, as noted, part of the stack is kept internal to the processor. This part of the stack serves the same purpose

that registers do. In three- and two-address machines, there is no need for the internal data registers. However, as we have demonstrated before, having some internal registers improves performance by cutting down the number of memory accesses. The RISC machines typically have a large number of registers.

Some processors maintain a few special-purpose registers. For example, the Pentium uses a couple of registers to implement the processor stack. Processors also have several registers reserved for the instruction execution unit. Typically, there is an instruction register that holds the current instruction and a program counter that points to the next instruction to be executed.

Throughout the book we present details on several processors. In total we describe five processors—the Pentium, MIPS, PowerPC, Itanium, and SPARC. Of these five processors, only the Pentium belongs to the CISC category. The rest are RISC processors. Our selection of processors reflects the dominance of the RISC designs in newer processors and the market domination of the Pentium. Pentium processor details are given in the next chapter. Part VI and Appendix H give details on the RISC processors.

6.3 Flow of Control

Program execution, by default, proceeds sequentially. This default behavior is due to the semantics associated with the execution cycle described in Section 1.5. The program counter (PC) register plays an important role in managing the control flow. At a simple level, the PC can be thought of as pointing to the next instruction. The processor fetches the instruction at the address pointed to by the PC (see Figure 1.9 on page 17). When an instruction is fetched, the PC is incremented to point to the next instruction. If we assume that each instruction takes exactly four bytes as in MIPS and SPARC processors, the PC is automatically incremented by four after each instruction fetch. This leads to the default sequential execution pattern. However, sometimes we want to alter this default execution flow. In high-level languages, we use control structures such as `if-then-else` and `while` statements to alter the execution behavior based on some run-time conditions. Similarly, the procedure call is another way we alter the sequential execution. In this section, we describe how processors support flow control. We look at both branch and procedure calls next. Interrupt is another mechanism to alter flow control, which is discussed in Chapter 20.

6.3.1 Branching

Branching is implemented by means of a branch instruction. This instruction carries the address of the target instruction explicitly. Branch instruction in processors such as the Pentium is also called the jump instruction. We consider two types of branches: unconditional and conditional. In both cases, the transfer control mechanism remains the same as that shown in Figure 6.3a.

Unconditional Branch

The simplest of the branch instructions is the *unconditional branch*, which transfers control to the specified target. Here is an example branch instruction:

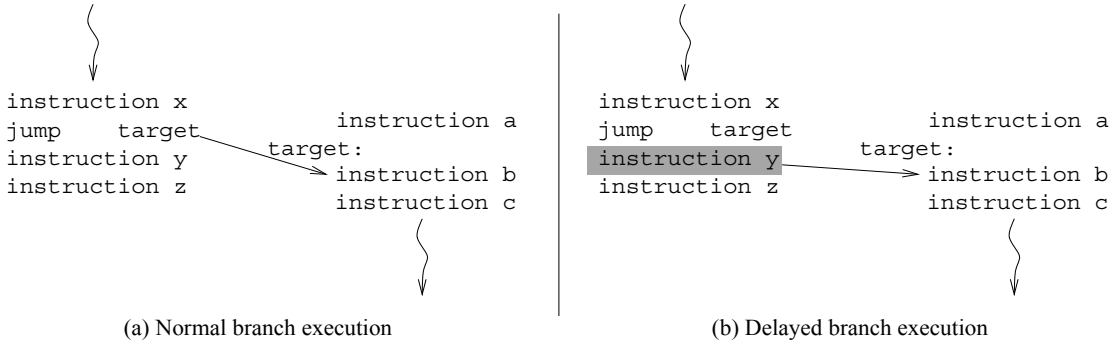


Figure 6.3 Control flow in branching.

```
branch    target
```

Specification of the target address can be done in one of two ways: absolute address or PC-relative address. In the former, the actual address of the target instruction is given. In the PC-relative method, the target address is specified relative to the PC contents. Most processors support absolute address for unconditional branches. Others support both formats. For example, MIPS processors support absolute address-based branch by

```
j    target
```

and PC-relative unconditional branch by

```
b    target
```

In fact, the last instruction is an assembly language instruction, although the processor only supports the `j` instruction.

The PowerPC allows each branch instruction to use either an absolute or a PC-relative address. The instruction encoding has a bit—called the absolute address (AA) bit—to indicate the type of address. As shown on page 589, if AA = 1, the absolute address is assumed; otherwise, the PC-relative address is used.

If the absolute address is used, the processor transfers control by simply loading the specified target address into the PC register. If PC-relative addressing is used, the specified target address is added to the PC contents, and the result is placed in the PC. In either case, since the PC indicates the next instruction address, the processor will fetch the instruction at the intended target address.

The main advantage of using the PC-relative address is that we can move the code from one block of memory to another without changing the target addresses. This type of code is called *relocatable code*. Relocatable code is not possible with absolute addresses.

Conditional Branch

In conditional branches, the jump is taken only if a specified condition is satisfied. For example, we may want to take a branch if the two values are equal. Such conditional branches are handled in one of two basic ways:

- *Set-Then-Jump:* In this design, testing for the condition and branching are separated. To achieve communication between these two instructions, a condition code register is used. The Pentium follows this design, which uses a flags register to record the result of the test condition. It uses a compare (`cmp`) instruction to test the condition. This instruction sets the various flag bits to indicate the relationship between the two compared values. For our example, we are interested in the zero bit. This bit is set if the two values are the same. Then we can use the conditional jump instruction that jumps to the target location if the zero bit is set. The following code fragment, which compares the values in registers AX and BX, should clarify this sequence:

```
        cmp  AX,BX    ;compare the two values in AX and BX
        je   target   ;if equal, transfer control to target
        sub  AX,BX    ;if not, this instruction is executed
          . . .
target:
        add  AX,BX    ;control is transferred here if AX = BX
          . . .
```

  The `je` (jump if equal) instruction transfers control to `target` only if the two values in registers AX and BX are equal. More details on the Pentium jump instructions are presented in Part V.

- *Test-and-Jump:* Most processors combine the testing and branching into a single instruction. We use the MIPS processor to illustrate the principle involved in this strategy. The MIPS provides several branch instructions that test and branch (for a quick peek, see Table 15.9 on page 633). The one that we are interested in here is the branch on equal instruction shown below:

```
beq    Rsrc1,Rsrc2,target
```

  This conditional branch instruction tests the contents of the two registers `Rsrc1` and `Rsrc2` for equality and transfers control to `target` if equal. If we assume that the numbers to be compared are in register `t0` and `t1`, we can write the branch instruction as

```
beq    $t1,$t0,target
```

  This single instruction replaces the two-instruction `cmp/je` sequence used by the Pentium.

Some processors maintain registers to record the condition of the arithmetic and logical operations. These are called *condition code registers.* These registers keep a record of the

status of the last arithmetic/logical operation. For example, when we add two 32-bit integers, it is possible that the sum might require more than 32 bits. This is the overflow condition that the system should record. Normally, a bit in the condition code register is set to indicate this overflow condition. The MIPS, for example, does not use condition registers. Instead, it uses exceptions to flag the overflow condition. On the other hand, the Pentium, PowerPC, and SPARC processors use condition registers. In the Pentium, the flags register records this information. In the PowerPC, this information is maintained by the XER register. SPARC processors use a condition code register.

Some instruction sets provide branches based on comparisons to zero. Some example processors that provide this type of branch instructions include the MIPS and SPARC processors.

Highly pipelined RISC processors support what is known as delayed branch execution. To see the difference between the delayed and normal branch execution, let us look at the normal branch execution shown in Figure 6.3*a*. When the branch instruction is executed, control is transferred to the target immediately. The Pentium, for example, uses this type of branching.

In delayed branch execution, control is transferred to the target after executing the instruction that follows the branch instruction. For example, in Figure 6.3*b*, before the control is transferred, the instruction `instruction y` (shown shaded) is executed. This instruction slot is called the *delay slot*. For example, the SPARC uses delayed branch execution. In fact, it also uses delayed execution for procedure calls. Why does this help? The reason is that by the time the processor decodes the branch instruction, the next instruction is already fetched. Thus, instead of throwing it away, we improve efficiency by executing it. This strategy requires reordering of some instructions. In Appendix H, which gives the SPARC processor details, we give examples of how it affects the programs.

6.3.2 Procedure Calls

The use of procedures facilitates modular programming. Procedure calls are slightly different from the branches. Branches are one-way jumps: once the control has been transferred to the target location, computation proceeds from that location, as shown in Figure 6.3. In procedure calls, we have to return control to the calling program after executing the procedure. Control is returned to the instruction following the call instruction as shown in Figure 6.4.

From Figures 6.3 and 6.4, you will notice that the branches and procedure calls are similar in their initial control transfer. For procedure calls, we need to return to the instruction following the procedure call. This return requires two pieces of information:

- *End of Procedure:* We have to indicate the end of the procedure so that the control can be returned. This is normally done by a special return instruction. For example, the Pentium uses `ret` and the MIPS uses the `jr` instruction to return from a procedure. We do the same in high-level languages as well. For example, in C, we use the `return` statement to indicate an end of procedure execution. High-level languages allow a default fall-through mechanism. That is, if we don't explicitly specify the end of a procedure, control is returned at the end of the block.
- *Return Address:* How does the processor know where to return after completing a proce-

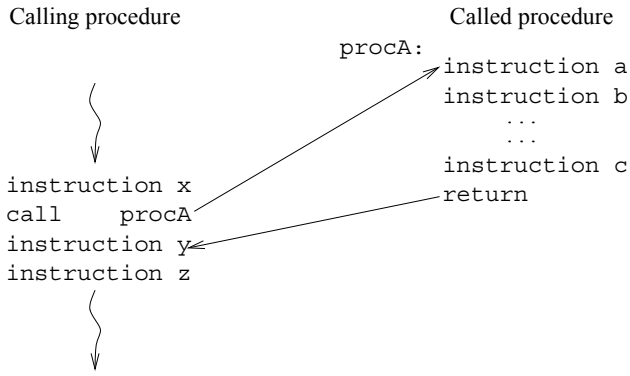


Figure 6.4 Control flow in procedure calls.

dure? This piece of information is normally stored when the procedure is called. Thus, when a procedure is called, it not only modifies the PC as in a branch instruction, but also stores the return address. Where does it store the return address? Two main places are used: a special register or the stack. In processors that use a register to store the return address, some use a special dedicated register, whereas others allow any register to be used for this purpose. The actual return address stored depends on the processor. Some processors such as the SPARC store the address of the `call` instruction itself. Others such as the MIPS and the Pentium store the address of the instruction *following* the `call` instruction.

The Pentium uses the stack to store the return address. Thus, each procedure call involves pushing the return address onto the stack before control is transferred to the procedure code. The return instruction retrieves this value from the stack to send the control back to the instruction following the procedure call. A more detailed description of the procedure call mechanism is found in Chapter 10.

MIPS processors allow any general-purpose register to store the return address. The return statement can specify this register. The format of the return statement is

```
jr    $ra
```

where `ra` is the register that contains the return address. The PowerPC, on the other hand, has a dedicated register, called the link register (LR), to store the return address. Both the MIPS and the PowerPC use a modified branch to implement a procedure call. The advantage of these processors is that simple procedure calls do not have to access memory. In Appendix H, we describe the procedure call mechanism used by the SPARC processor.

Most RISC processors that support delayed branching also support delayed procedure calls. As in the branch instructions, control is transferred to the target after executing the instruction that follows the call (see Figure 6.5). Thus, after the procedure is done, control should be

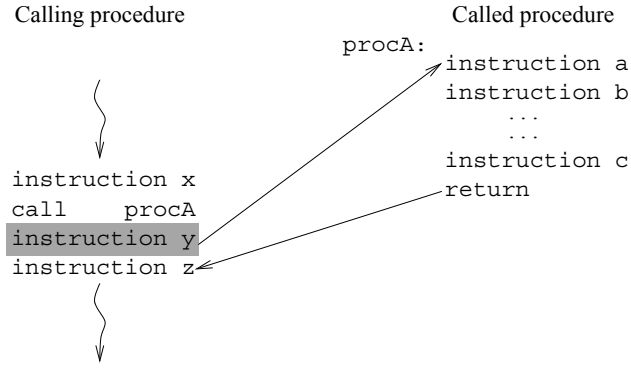


Figure 6.5 Control flow in delayed procedure calls.

returned to the instruction after the delay slot (to `instruction z` in the figure). We show some SPARC examples of this in Appendix H.

Parameter Passing

The general architecture dictates how parameters are passed on to the procedures. There are two basic techniques: register-based or stack-based. In the first method, parameters are placed in processor registers and the called procedure will read the parameter values from these registers. In the stack-based method, parameters are pushed onto the stack and the called procedure would have to pop them off the stack.

The advantage of the register method is that it is faster than the stack method. However, because of the limited number of registers, it imposes a limit on the number of parameters. Furthermore, recursive procedures cannot use the register-based mechanism. Because RISC processors tend to have more registers, register-based parameter passing is used in PowerPC and MIPS processors. The Pentium, due to the small number of registers, tends to use the stack for parameter passing. We describe these two parameter passing mechanisms in detail in Chapter 10.

Recent processors use a register window mechanism that allows a more flexible parameter passing. The SPARC and Intel Itanium processors use this parameter passing mechanism. We describe this method in detail in Chapter 14 and Appendix H.

6.4 Instruction Set Design Issues

There are several design issues that influence the instruction set of a processor. We have already discussed one issue, the number of addresses used in an instruction. Recent processors, except for the Pentium, use three-address instructions. The Pentium, as mentioned, uses the two-address format. In this section, we discuss some other design issues.

6.4.1 Operand Types

Processor instructions typically support only the basic data types. These include characters, integers, and floating-point numbers. Since most memories are byte addressable, representing characters does not require special treatment. Recall that in a byte-addressable memory, the smallest memory unit we can address, and therefore access, is one byte. We can, however, use multiple bytes to represent larger operands. Processors provide instructions to load various operand sizes. Often, the same instruction is used to load operands of different sizes. For example, the Pentium instruction

```
mov     AL,address      /* Loads an 8-bit value */
```

loads the AL register with an 8-bit value from memory at `address`. The same instruction can also be used to load 16- and 32-bit values as shown in the following two Pentium instructions.

```
mov     AX,address      /* Loads a 16-bit value */
mov     EAX,address     /* Loads a 32-bit value */
```

In these instructions, the size of the operand is indirectly given by the size of the register used. The AL, AX, and EAX are 8-, 16-, and 32-bit registers, respectively. In those instructions that do not use a register, we can use size specifiers. We show examples of this in Section 9.5.1 on page 339. This type of specification is typical for the CISC processors.

RISC processors specify the operand size in their load and store operations. Note that only the load and store instructions move data between memory and registers. All other instructions operate on register-wide data. Below we give some sample MIPS load instructions:

```
lb      Rdest,address       /* Loads a byte */
lh      Rdest,address       /* Loads a halfword (16 bits) */
lw      Rdest,address       /* Loads a word (32 bits) */
ld      Rdest,address       /* Loads a doubleword (64 bits) */
```

The last instruction is available only on 64-bit processors. In general, when the size of the data moved is smaller than the destination register, it is sign-extended to the size of `Rdest`. There are separate instructions to handle unsigned values. For unsigned numbers, we use `lbu` and `lhu` instead of `lb` and `lh`, respectively.

Similar instructions are available for store operations. In store operations, the size is reduced to fit the target memory size. For example, storing a byte from a 32-bit register causes only the lower byte to be stored at the specified address. SPARC processors also use a similar set of instructions.

So far we have seen operations on operands located either in registers or in memory. In most instructions, we can also use constants. These constants are called immediate values because these values are available immediately as they are encoded as part of the instruction. In RISC processors, instructions excluding the load and store use registers only; any nonregister value is treated as a constant. In most assembly languages, a special notation is used to indicate registers. For example, in MIPS assembly language, the instruction

```
add    $t0,$t0,-32    /* t0 = t0 - 32 */
```

subtracts 32 from the `t0` register and places the result back in the `t0` register. Notice the special notation to represent registers. But there is no special notation for constants. Pentium assemblers also use a similar strategy. Some assemblers, however, use the "#" sign to indicate a constant.

6.4.2 Addressing Modes

Addressing mode refers to how the operands are specified. As we have seen in the last section, operands can be in one of three places: in a register, in memory, or part of the instruction as a constant. Specifying a constant as an operand is called the *immediate addressing mode*. Similarly, specifying an operand that is in a register is called the *register addressing mode*. All processors support these two addressing modes.

The difference between the RISC and CISC processors is in how they specify the operands in memory. RISC processors follow the load/store architecture. Instructions other than load and store expect their operands in registers or specified as constants. Thus, these instructions use register and immediate addressing modes. Memory-based operands are used only in the load and store instructions. In contrast, CISC processors allow memory-based operands for all instructions. In general, CISC processors support a large variety of addressing modes. RISC processors, on the other hand, support only a few, often just two, addressing modes in their load/store instructions. Most RISC processors support the following two addressing modes to specify the memory-based operands:

- The address of the memory operand is computed by adding the contents of a register and a constant. If this constant is zero, the contents of the register are treated as the operand address. In this mode, the memory address is computed as

  Address = Register + constant.

- The address of the memory operand is computed by adding the contents of two registers. If one of the register contents is zero, this addressing mode becomes the same as the one above with zero constant. In this mode, the memory address is computed as

  Address = Register + Register.

Among the RISC processors we discuss, the Itanium provides slightly different addressing modes. It uses the computed address to update the contents of the register. For example, in the first addressing mode, the register contents are replaced by the value obtained by adding the constant to the contents of the register.

The Pentium provides a variety of addressing modes. The main motivation for this is the desire to support high-level language data structures. For example, one of the Pentium's addressing modes can be used to access elements of a two-dimensional array. We discuss the addressing modes of the Pentium in Chapter 11.

6.4.3 Instruction Types

Instruction sets provide different types of instructions. We describe some of these instruction types here.

Data Movement Instructions: All instruction sets support data movement instructions. The type of instructions supported depends on the architecture. We can divide these instructions into two groups: instructions that facilitate movement of data between memory and registers and between registers. Some instruction sets have special data movement instructions. For example, the Pentium has special instructions such as `push` and `pop` to move data to and from the stack.

In RISC processors, data movement between memory and registers is restricted to load and store instructions. Some RISC processors do not provide any explicit instructions to move data between registers. This data transfer is accomplished indirectly. For example, we can use the `add` instruction

```
add    Rdest,Rsrc,0 /* Rdest= Rsrc + 0 */
```

to copy contents of `Rsrc` to `Rdest`. The Pentium provides an explicit `mov` instruction to copy data. The instruction

```
mov    dest,src
```

copies the contents of `src` to `dest`. The `src` and `dest` can be either registers or memory. In addition, `src` can be a constant. The only restriction is that both `src` and `dest` cannot be located in memory. Thus, we can use the `mov` instruction to transfer data between registers as well as between memory and registers.

Arithmetic and Logical Instructions: Arithmetic instructions support floating-point as well as integer operations. Most processors provide instructions to perform the four basic arithmetic operations: addition, subtraction, multiplication, and division. Since the 2s complement number system is used, addition and subtraction operations do not need separate instructions for unsigned and signed integers. However, the other two arithmetic operations need separate instructions for signed and unsigned numbers.

Some processors do not provide division instructions, whereas others support only partially. What do we mean by partially? Remember that the division operation produces two outputs: a quotient and a remainder. We say that the division operation is fully supported if the division instruction produces both results. For example, the Pentium and MIPS provide full division support. On the other hand, the SPARC and PowerPC only provide the quotient, and the Itanium does not support the division instruction at all.

Logical instructions provide the basic bit-wise logical operations. Processors typically provide logical `and` and `or` operations. Other logical operations including the `not` and `xor` operations are supported by most processors.

Most of these instructions set the condition code bits, either by default or when explicitly instructed. The common condition code bits, which record the status of the most recent operation, are

S — Sign bit (0 = positive, 1 = negative);
Z — Zero bit (0 = nonzero value, 1 = zero value);
O — Overflow bit (0 = no overflow, 1 = overflow);
C — Carry bit (0 = no carry, 1 = carry).

The sign bit is updated to indicate whether the result is positive or negative. Since the most significant bit indicates the sign, the S bit is a copy of the sign bit of the result of the last operation. The zero bit indicates whether the last operation produced a zero or nonzero result. This bit is useful in comparing two values. For example, the Pentium instructions

```
cmp     count,25      /* compare count to 25 */
je      target        /* if equal, jump to target*/
```

compare the value of `count` to 25 and set the condition code bits. The jump instruction checks the zero bit and jumps to `target` if the zero bit is set (i.e., Z = 1). Note that the `cmp` instruction actually subtracts 25 from `count` and sets the Z bit if the result is zero.

The overflow bit records the overflow condition when the operands are signed numbers. The carry bit is set if there is a carry out of the most significant bit. The carry bit indicates an overflow when the operands are unsigned numbers.

In the Pentium, the condition code bits are set by default. In other processors, two versions of arithmetic and logical instructions are provided. For example, in the SPARC processor, `ADD` does not update the condition codes, whereas the `ADDcc` instruction updates the condition codes.

Flow Control and I/O Instructions: The flow control instructions include the branch and procedure calls discussed before. Since we have already discussed these instructions, we do not describe them. Interrupt is another flow control mechanism that is discussed in Chapter 20.

The type of input/output instructions provided by processors varies widely from processor to processor. The main characteristic that influences the I/O instructions is whether the processor supports isolated or memory-mapped I/O. Recall that isolated I/O requires special I/O instructions whereas memory-mapped I/O can use the data movement instructions to move data to or from the I/O devices (see Section 1.7 on page 27).

Most processors support memory-mapped I/O. The Pentium is an example of a processor that supports isolated I/O. Thus, it provides separate instructions to perform input and output. The `in` instruction can be used to read a value from an I/O port into a register. For example, the instruction

```
in    AX,io_port
```

reads a 16-bit value from the specified I/O port. Similarly, the `out` instruction

```
out    io_port,AX
```

writes the 16-bit value in the AX register to the specified I/O port. More details on the Pentium I/O instructions are given in Chapter 19.

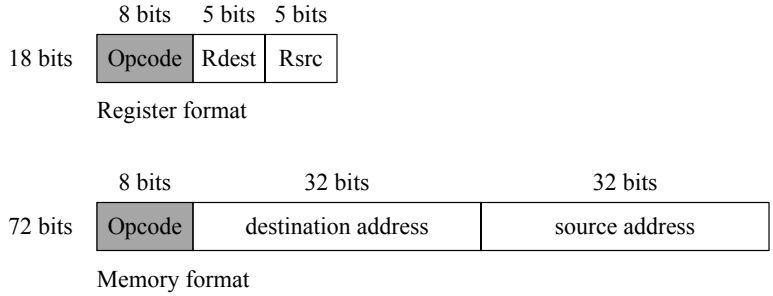


Figure 6.6 Instruction size depends on whether the operands are in registers or memory.

6.4.4 Instruction Formats

Processors use two types of basic instruction format: fixed-length or variable-length instructions. In the fixed-length encoding, all (or most) instructions use the same size instructions. In the latter encoding, the length of the instructions varies quite a bit. Typically, RISC processors use fixed-length instructions, and the CISC designs use variable-length instructions.

All 32-bit RISC processors discussed in this book use instructions that are 32-bits wide. Some examples are the SPARC, MIPS, and PowerPC processors. The Intel Itanium, which is a 64-bit processor, uses fixed-length, 41-bit wide instructions. We discuss the instruction encoding schemes of all these processors throughout the book.

The size of the instruction depends on the number of addresses and whether these addresses identify registers or memory locations. Figure 6.1 shows how the size of the instruction varies with the number of addresses when all operands are located in registers. This format assumes that eight bits are reserved for the operation code (opcode). Thus we can have 256 different instructions. Each operand address is five bits long, which means we can have 32 registers. This is the case in processors like the MIPS. The Itanium, for example, uses seven bits as it has 128 registers.

As you can see from this figure, using fewer addresses reduces the length of the instruction. The size of the instruction also depends on whether the operands are in memory or in registers. As mentioned before, RISC processors keep their operands in registers. In CISC processors like the Pentium, operands can be in memory. If we use 32-bit memory addresses for each of the two addresses, we would need 72 bits for each instruction (see Figure 6.6) whereas the register-based instruction requires only 18 bits. For this and other efficiency reasons, the Pentium does not permit both addresses to be memory addresses. It allows at most one address to be a memory address.

The Pentium, which is a CISC processor, encodes instructions that vary from one byte to several bytes. Part of the reason for using variable length instructions is that CISC processors tend to provide complex addressing modes. For example, in the Pentium, if we use register-based operands, we need just 3 bits to identify a register. On the other hand, if we use a memory-based operand, we need up to 32 bits. In addition, if we use an immediate operand,

we need a further 32 bits to encode this value into the instruction. Thus, an instruction that uses a memory address and an immediate operand needs 8 bytes just for these two components. You can realize from this description that providing flexibility in specifying an operand leads to dramatic variations in instruction sizes.

The opcode is typically partitioned into two fields: one identifies the major operation type, and the other defines the exact operation within that group. For example, the major operation could be a branch operation, and the exact operation could be "branch on equal." These points become clearer as we describe the instruction formats of various processors in later chapters.

6.5 Microprogrammed Control

In the last section, we discussed several issues in designing a processor's instruction set. Let us now focus on how these instructions are executed in the hardware. The basic hardware is the datapath discussed in Chapter 1 (e.g., see page 16). Before proceeding further, you need to understand the digital logic material presented in Chapters 2 and 3.

We start this section with an overview of how the hardware executes the processor's instructions. To facilitate our description, let's look at the simple datapath shown in Figure 6.7. This datapath uses a single bus to interconnect the various components. For the sake of concreteness, let us assume the following:

- The *A bus*, all registers, and the system data and address buses are all 32 bits wide,
- There are 32 general-purpose registers `G0` to `G31`,
- The ALU can operate on 32-bit operands.

Since we are using only a single bus, we need two temporary holding registers: registers A and C. Register A holds the A operand required by the ALU. The output of the ALU is stored in register C. If you have read the material presented on digital logic design in Part II, you will see that the implementation of these registers is straightforward. A sample design is shown in Figure 6.8. A set of 32 D flip-flops is used to latch the A operand for the ALU. As shown in this figure, we use the control input `Ain` to clock in the data. The output of the A register is always available to the A input of the ALU. A similar implementation for the C register uses `Cin` as the clock input signal to store the ALU output. The output of this register is fed to the A bus only if the control signal `Cout` is activated. Later on we show how these control signals are used to execute processor instructions.

The memory interface uses the four shaded registers shown in Figure 6.7. These registers interface to the data and address buses on the one side and to the A bus on the other. Figure 6.9 shows how these registers are interfaced to these buses. Details about the use of these registers and the required control signals are discussed next.

- *PC Register:* This is the program counter register we have discussed before. It contains the address of the next instruction to be executed. In our datapath, we assume that we can place the PC contents on the system address bus. This register can also place its contents

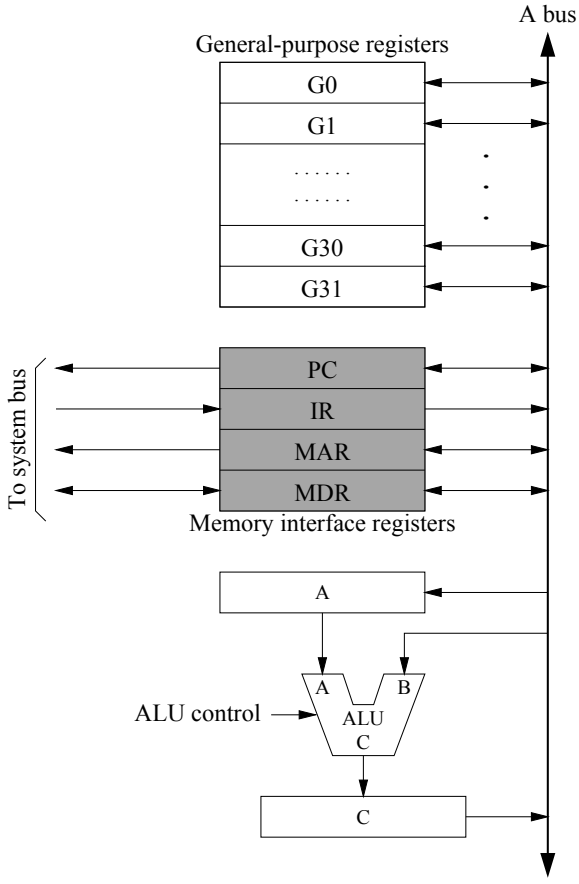


Figure 6.7 An example 1-bus datapath.

on the A bus; in addition, we can write into this register from the A bus. We use `PCin` to load the contents of the A bus into the PC register. The contents of the PC register can be placed on the system address bus and A bus, simultaneously if required. The two control signals, `PCbout` and `PCout`, independently control this operation.

- *IR Register:* The instruction register holds the instruction to be executed. The IR register receives the instruction from the system data bus. Because of its simple interface, we just need `IRbin` and `IRout` control signals.
- *MAR Register:* The memory address register is used to hold the address of an operand stored in memory. This is used in addressing modes that allow an operand to be located in memory. This register interface is similar to that of the PC register. It uses three control signals as does the PC register: `MARbout`, `MARin`, and `MARout`.

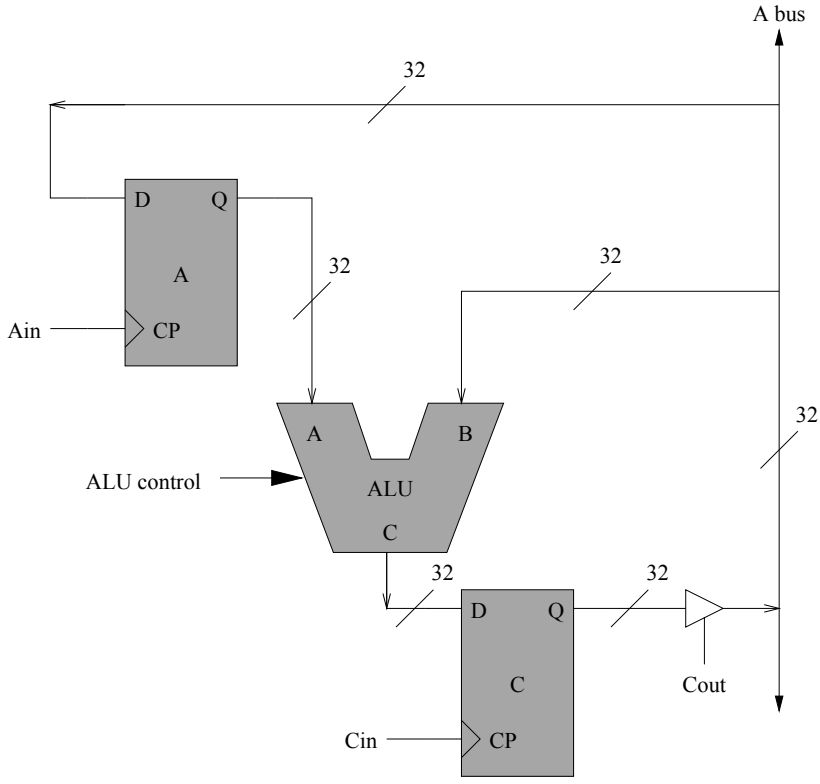


Figure 6.8 ALU circuit details: All datapaths are assumed to be 32 bits wide.

- *MDR Register:* The memory data register is used to hold the operand read from memory. The address of the operand is in MAR. This register provides a bidirectional interface to both the system data bus and the A bus. Thus, we need the four control signals shown in Figure 6.9.

The general-purpose registers interface to the A bus only. Each of the 32 registers has two control signals, `Gxin` and `Gxout`, where `GX` is in the range `G0` to `G31`.

So how do we use this hardware to execute processor instructions? Let us consider the `add` instruction

```
add     Rd,Rs1,Rs2
```

to illustrate the control signals needed to execute instructions. This instruction adds the contents of general-purpose registers `Rs1` and `Rs2` and stores the result in `Rd`. Suppose we want to add the contents of registers 5 and 7 and place the result in register 9. That is, we want to execute the following instruction:

```
add     %G9,%G5,%G7
```

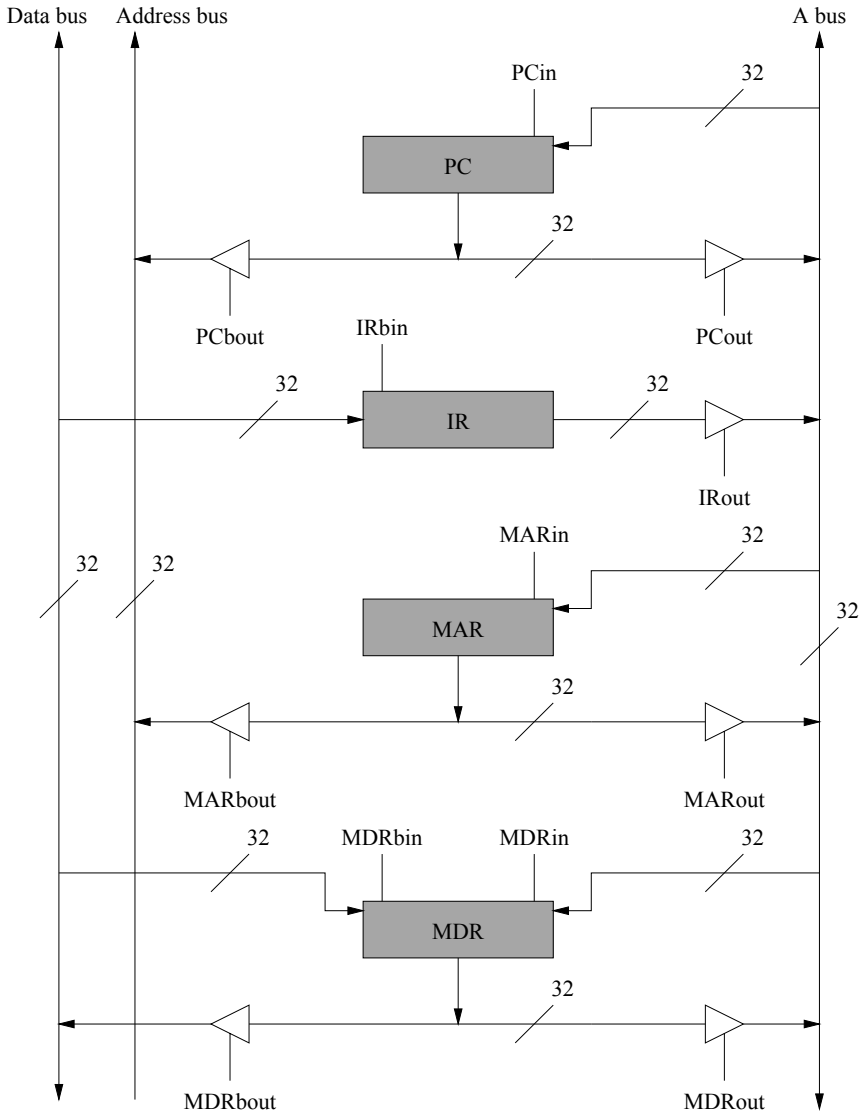


Figure 6.9 Implementation details of the memory interface.

Since we have only a single bus, we have to latch the contents of one of the two source registers in our temporary holding register A. Then we can move the contents of the other register on to the A bus to add the two values. We outline the sequence of steps involved in performing this addition:

1. We assert the `G5out` signal to place the contents of the general-purpose register G5 on the A bus. Simultaneously, we assert the `Ain` signal to latch the A bus contents. Asserting these two control signals simultaneously transfers contents of the G5 register to the A register via the A bus.
2. We now have to place the contents of G7 on the A bus. We do this by asserting the `G7out` control signal. Since the output of the A register is always available to the A input of the ALU, we can now instruct the ALU to perform the addition by specifying appropriate function control. The ALU output is latched into the C register by asserting the `Cin` signal.
3. The final step is to write the value in the C register to G9. This transfer is achieved by asserting `Cout` and `G9in` simultaneously. This step completes the addition operation.

Ideally, the time required to perform each of these steps should be the same. This defines our cycle time. Although the actions taken in Steps 1 and 3 are similar (assume that each step takes one cycle), the second step might require more time. It depends on the time needed by the ALU to perform the addition operation. If this time is more than that required for the other two steps, we can add more cycles to the second step. This is similar to the "wait" cycles inserted into a memory read or write operation (see Section 5.3.2 on page 154).

In our description, we conveniently skipped one important question: How does the processor know that it has to perform the addition operation? This information is obtained from the opcode field of the instruction.

Now we show how instructions are fetched from memory. Instruction fetch involves placing the PC contents on the system address bus and, after waiting for the memory to place the data on the system data bus, reading the data into the IR register. We have to also update the PC to point to the next instruction. We assume that each instruction is 32 bits wide. Updating the PC means adding 4 to the PC contents. As in the `add` instruction execution, we detail the sequence of steps involved below:

1. Assert `PCbout` to place the PC contents on the system address bus. Since we have to update the PC contents to point to the next instruction, we use the services of the ALU to do this. Therefore, we simultaneously pass the PC contents to the ALU via the A bus by asserting the `PCout` signal. The ALU is asked to perform the `add4` operation on its B input. The `add4` is a unary operator that adds 4 to the input. As in the `add` instruction execution, the ALU output is latched into the C register by asserting the `Cin` signal.
2. We wait one clock cycle to give time for the memory to retrieve the instruction. We read this instruction during the next clock cycle. During this cycle, we also load the updated PC value by copying it from the C register. This transfer requires the `Cout` and `PCin` signals.
3. Let us assume that the memory is able to place the data on the system data bus by this clock cycle. All we have to do now is to copy the data into the IR register. We can easily accomplish this by asserting the `IRbin` signal. This completes the instruction fetch operation.

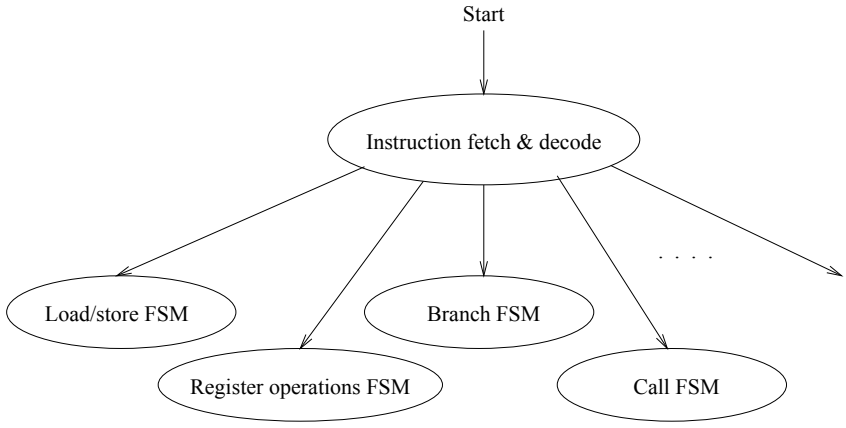


Figure 6.10 A high-level FSM for instruction execution.

The instruction in the IR register is decoded to find the operation to be performed (e.g., `add`). The opcode field specifies the type of operation to be done. If, for example, the operation is addition, we have to identify the source and destination registers and generate control signals to perform the operation as explained before. The behavior of the fetch-decode-execute cycle can be expressed by using a finite state machine. Recall that we have used finite state machines in Chapter 4 to design digital logic circuits.

Figure 6.10 shows a high-level FSM for implementing the instruction execution cycle. The first step is common to all instructions: the instruction must be fetched and decoded. After decoding, the opcode identifies the group to which the instruction belongs. In this figure, we have used typical instruction groups found on RISC machines. We have shown four example instruction groups:

1. *Load/Store Instructions:* These instructions move data between registers and memory. All other instructions operate on the data located in the registers. The FSM associated with this group of instructions will further distinguish the various types of load and store instructions.
2. *Register Instructions:* Instructions in this group include the arithmetic and logical instructions. All required operands are assumed to be in the processor's internal registers. The FSM for this group will generate the necessary control signals depending on the actual instruction (such as `add`, which we have seen before).
3. *Branch Instructions:* These instructions alter the flow control of a program. The target of the branch can be specified directly as a constant in the instruction or indirectly through a register (see our discussion in Section 6.3.1). The branch FSM distinguishes among the different types of branches and generates appropriate control signals.
4. *Call Instructions:* The last group we have shown is the procedure call instructions. As mentioned in our discussion in Section 6.3.2, call instructions are related to the branch

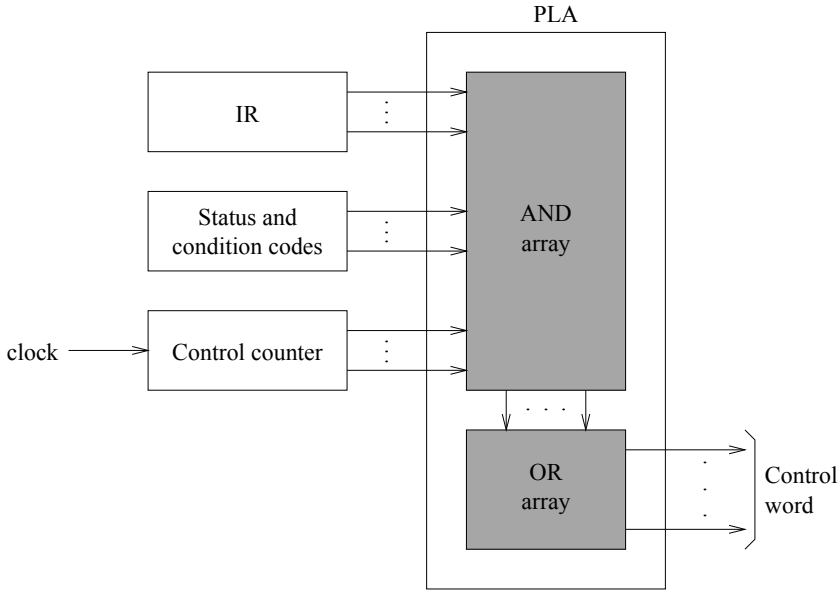


Figure 6.11 Hardware implementation of the controller.

group but are more complicated as they have to return control after completing the procedure.

Depending on the processor instruction set, more instruction groups can be added. Since we have covered FSMs in detail in Chapter 4, we do not discuss this topic in any more detail. Instead, we look at hardware- and software-based implementations of such an FSM.

6.5.1 Hardware Implementation

From our discussion, it is clear that we can implement instructions by generating appropriate control signals. The required control signals are described in the corresponding FSM. As described in Chapter 4, we can implement the FSM in hardware. Figure 6.11 shows an example implementation using a PLA.

The input to the PLA consists of three groups of signals. We need to feed the opcode so that the circuit generates appropriate control signals for that instruction. This input comes from the opcode field of the IR register.

The next group is the status and condition codes. This input is required for instructions such as conditional branches. For example, branch on equal `beq` requires the zero flag input to decide whether to take the branch.

The third input is driven by the clock input. The control counter keeps track of the steps involved in executing an instruction. For example, in executing the `add` instruction, we identified three steps. We can use this counter to specify the control signals that should be generated during each step.

If the instruction set is simple, hardware implementation is preferred. This is typically the case for RISC processors. However, for complex instruction sets, this implementation is not preferred. Instead, a program is used to generate the control signals. This is the approach taken by CISC processors. We describe this approach next.

6.5.2 Software Implementation

The hardware approach is complex and expensive to implement for CISC machines. This was particularly true in the 1960s and 1970s. Furthermore, hardware implementation is very rigid. To avoid these problems, Wilkes and Stinger [40] proposed a software approach. If we look closely, the FSM specifies the control signals that should be generated during each step. To see what we mean, let's rewrite the instruction fetch and add instruction control sequences.

Instruction	Step	Control signals
Instruction fetch	S1	`PCbout: read: PCout: ALU=add4: Cin;`
	S2	`read: Cout: PCin;`
	S3	`read: IRbin;`
	S4	Decodes the instruction and jumps to the appropriate execution routine
`add %G9,%G5,%G7`	S1	`G5out: Ain;`
	S2	`G7out: ALU=add: Cin;`
	S3	`Cout: G9in: end;`

If we assume that each step can be executed in one cycle, we need three cycles to fetch the instruction and at least one cycle to decode the instruction. Another three cycles are needed to execute the `add` instruction. All signals in a single step can be asserted simultaneously. We separate the signals by a colon (:) and use a semicolon (;) to indicate the end of a step. Most of the signals are from the datapath shown in Figure 6.7, but there are some new signals that need explanation. The `read` signal is used to generate the system control bus read signal. As we have seen in Chapter 5, this signal initiates a memory read cycle.

In instruction fetch, we use the ALU function `add4` to update the PC contents. This ALU function adds 4 to the B input. In the `add` instruction, we use `GXout` and `GXin` to control output and input to the general-purpose register `GX`. The `end` signal indicates that the instruction execution has been completed and we should initiate another instruction fetch cycle.

To illustrate the use of the MAR and MDR registers, let us see how the instruction

```
add     %G9,[%G5],%G7
```

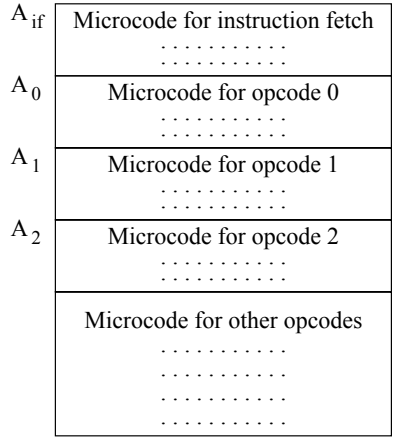


Figure 6.12 A simple microcode organization.

is implemented. This instruction uses register indirect addressing to specify one of the operands, and is very similar to the previous `add` instruction except that one of the operands is in memory. The general-purpose register G5 gives the operand address. To execute this instruction, we need to get this operand from memory. To do this, we place the contents of G5 in MAR and initiate a memory read cycle by placing the address in MAR on the system address bus. After a cycle, the operand from the memory is placed in MDR. From then on, we go through the same sequence of steps as in the previous `add` instruction, as shown below:

Instruction	Step	Control signals
`add %G9,[%G5],%G7`	S1	`G5out: MARin: MARbout: read;`
	S2	`read;`
	S3	`read: MDRbin: MDRout: Ain;`
	S4	`G7out: ALU=add: Cin;`
	S5	`Cout: G9in: end;`

These examples suggest an alternative way of generating the control signals. Suppose that we encode the signals for each step as a codeword. Then we can store these codewords as a program just as with machine language instructions. Each such codeword is referred to as a *microinstruction* and the sequence of codewords for an instruction constitutes a *microroutine*. We can write a *microprogram* that implements the FSM we talked about before.

A straightforward way of structuring the microprogram is shown in Figure 6.12, which shows a linear organization. The instruction fetch microroutine is shown first in this micropro-

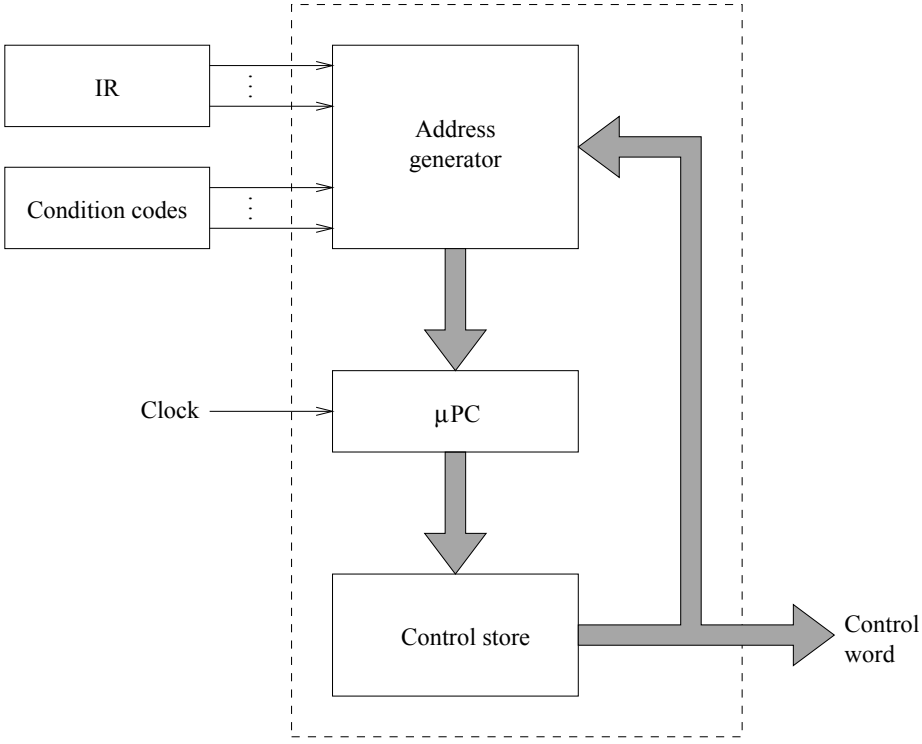


Figure 6.13 Microprogramming uses a control store to control the actions.

gram. After the instruction has been decoded, it jumps to the appropriate microroutine based on the opcode. The execution of the microcode is sequential. When the `end` signal is encountered, the instruction fetch routine is executed.

A microcontroller that executes this microprogram is shown in Figure 6.13. The microprogram is stored in the control store. The microprogram counter (μPC) is similar to the program counter we have for machine language programs. Like the PC, μPC specifies the codeword that should be executed next. The address generation circuit is used to initiate the starting address (i.e., the address of the instruction fetch microroutine) and to implement microprogram jumps. For example, at the end of executing an instruction, the `end` signal causes initiation of the instruction fetch. If we assume that the instruction fetch microroutine is at address 0, the `end` signal can be used to clear the address register to initiate an instruction fetch cycle.

The address generation circuit is also useful to generate the appropriate address depending on the opcode from the IR register and conditional branch type of instructions by taking the condition code inputs. The clock input steps the μPC through the microprogram.

The microprogram organization shown in Figure 6.12 makes the microprogram unnecessarily long as common parts of the code are replicated due to its linear organization. An efficient

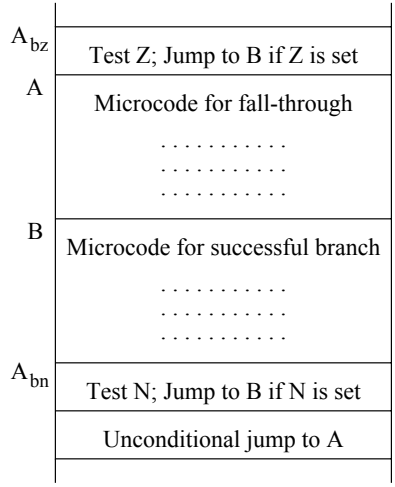


Figure 6.14 Microcode organization to allow conditional branches in the microprogram.

way of organizing the microprogram is shown in Figure 6.14. In this organization, as in the programs we write, we can keep only one copy of the common microcode. To use this organization, however, we have to augment each microinstruction with the address of the next microinstruction. Thus, our control word gets longer than in the other organization. Since we do not replicate the common code, we end up saving space in the control store.

Microinstruction Format

Each microinstruction consists of the control signals needed to execute that instruction on the datapath. Let us consider the single-bus datapath shown in Figure 6.7. The microinstruction format for this datapath is shown in Figure 6.15.

The first group of 12 signals comes from the control signals shown in Figure 6.9. These signals control the memory interface. The next three bits control the A and C latches (see Figure 6.8). The general-purpose registers are controlled by the 64 signals: two for each register. We are assuming that the ALU can perform eight functions: `add`, `add4`, `sub`, `BtoC`, `and`, `or`, `shl`, and `shr`. These functions are self-explanatory except for the following:

- The `add4` function is used to update the PC contents. We have seen an example usage of this function in the instruction fetch microroutine.
- The `BtoC` function copies the B input to the C output. This function, for example, is useful for moving data from one register to another. However, in our single-bus datapath, we can do this transfer without involving the ALU. For example, to copy contents of G5 to G6, we use the following microinstruction:

```
G5out: G6in;
```

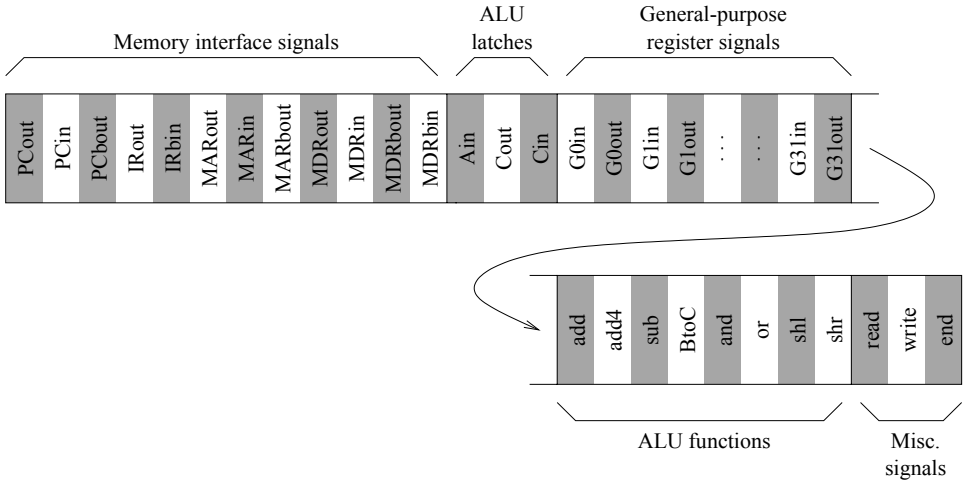


Figure 6.15 A simple microinstruction format for the datapath shown in Figure 6.7. In this organization, there is one bit for each signal. This microcode organization is called the horizontal organization.

As we show next, depending on the encoding scheme used for the microinstructions, we may not be able to specify both `G5out` and `G6in` in the same codeword. Furthermore, as we show later, in 2- and 3-bus systems, such a transfer will have to go through the ALU. In that case, we need a function to pass one of the ALU inputs to the output.

- The `shl` and `shr` functions shift left and right by one bit position, respectively.

These are some of the typical instructions provided by processors. Deriving microinstructions in the format shown in Figure 6.15 is straightforward. For example, the codeword for

```
G5out: G6in;
```

consists of `G5out` = 1, `G6in` = 1, and all the other bits are zero.

The advantage of this microinstruction format is that it allows specification of many actions in a single instruction. For example, we can copy the contents of the G0 register to registers G2 through G5 in one step, as shown below:

```
G0out: G2in: G3in: G4in: G5in: end;
```

The main problem with this format is the size of the microinstruction. In our example, we need 90 bits for each codeword. This encoding scheme follows the *horizontal organization*. Clearly, horizontal organization does not require any decoding of the information contained in the microinstruction. These bits can be used directly to generate control signals.

We can reduce the codeword size by encoding the information in the microinstruction. For example, instead of using 64 bits to control the 32 general-purpose registers, we could use a 5-bit register number and a single bit to indicate in or out control. This type of encoding

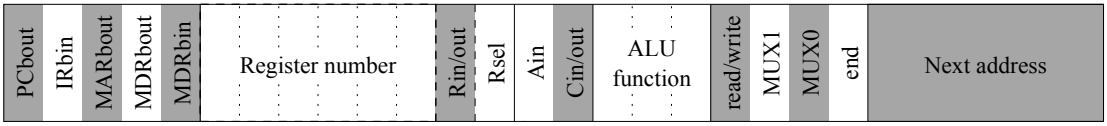


Figure 6.16 A vertical microcode organization.

scheme is called the *vertical organization*. Vertical organization can specify only a few functions compared to horizontal organization. The microinstruction format, shown in Figure 6.16, is organized along these lines. As you can see from this figure, the size of the codeword is substantially smaller: we need only 20 bits (excluding the *Next address* portion and the two MUX signals). Of course, there is no free lunch. This format needs additional decoders and execution takes more cycles. Since we can specify only one of the 32 general-purpose registers, even a simple register-to-register copy takes two steps. For example, to copy the contents of G0 to G2, we use the following microinstruction sequence:

```
G0out: ALU=BtoC: Cin;
Cout: G2in: end;
```

To further demonstrate the disadvantages of the vertical microcode organization, let's look at the copying example discussed before. To copy G0 to G2 through G5, we need several cycles:

```
G0out: ALU=BtoC: Cin;
Cout: G2in;
Cout: G3in;
Cout: G4in;
Cout: G5in: end;
```

This code also shows the need for the `BtoC` ALU function.

The semantics of the microinstruction format shown in Figure 6.16 requires some explanation. The encoding scheme uses six bits for the register number. The reason is that we have 32 general-purpose register and four memory interface registers. When the most significant bit of the "Register number" field is 0, the remaining five bits are taken as the register number of a general-purpose register. When this bit is 1, each of the remaining four bits is used to indicate one of the four memory interface registers—PC, IR, MAR, and MDR, as shown below:

Register number field	Register specified
`0xxxxx`	General-purpose register `Gxxxxx`
`100001`	PC register
`100010`	IR register
`100100`	MAR register
`101000`	MDR register

The first five bits are used to generate memory control signals for the four memory interface registers. The Rin/out signal specifies the in (1) or out (0) direction of data movement on the A bus. These actions take place only when the register selection signal Rsel is asserted. When this bit is 0, no register is selected. For example, we set Rsel to 0 if we want to feed the C register back to the A register. The read and write actions are combined into a single control bit. Like the Rin/out bit, the Cin/out bit is used to control data movement to the C register. For the A register, we just have a single Ain control bit.

The ALU functions are encoded using three bits as follows:

ALU function field	Function specified
000	add
001	add4
010	sub
011	BtoC
100	and
101	or
110	shl
111	shr

Deriving microinstructions in the vertical format is simple. For example, the microinstruction

```
G0out: ALU=BtoC: Cin;
```

can be encoded as

Register number = 000000
Rin/out = 0
ALU function = 011
Cin/out = 1

All other fields in the instruction are zero.

The register number of a general-purpose register comes from two main sources:

- The machine instruction in IR specifies the registers that should be used in the current instruction. In our case, we need three register specifications: two source registers Rs1 and Rs2 and a destination register Rd.
- The microinstruction can also specify one of these registers using the register number field.

To select a particular register, we can use a multiplexer/decoder circuit such as the one shown in Figure 6.17. The microinstruction controls the multiplexer function using the MUX1 and MUX0

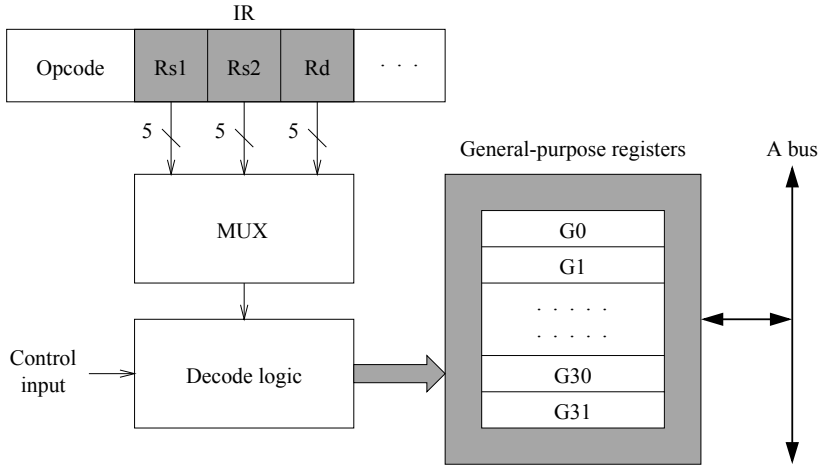


Figure 6.17 An example general register control circuit.

control bits. The decoder circuit takes the 5-bit multiplexer output and `Rin/out` and `Rsel` control inputs from the microinstruction to enable and select a register. If the `Rsel` signal is inactive, no G register is selected.

We mentioned before that, to provide complete support for microinstruction branching, we need to add the next microinstruction address. The format shown in Figure 6.16 includes this field. The microcontroller shown in Figure 6.18 shows how the vertically organized microcode is executed.

The microinstruction register (μIR) holds the microinstruction. Since we are using the vertical organization, we need to decode the microinstruction to generate the control signals. The μPC provides the address of the microinstruction to the control store. The μPC can be loaded from either the `Next address` field or from the start address generator. The start address generator outputs the appropriate microroutine address depending on the opcode and control codes.

The designer needs to weigh the pros and cons of the horizontal and vertical microprogram organizations. For example, to provide improved performance, the horizontal scheme is preferred, as it does not impose any restrictions on the concurrent use of the recourses in the datapath. But the microprogram tends to be large, and the hardware cost increases. If the designer wants a cheaper version, vertical organization may be used. The downside is the reduced performance due to the restrictions imposed by the vertical scheme.

Datapaths with More Buses

In our discussions so far, we have used a single-bus datapath. The use of single bus forces us to multiplex the bus to transfer operands, which takes more time to execute instructions. Figure 6.19 shows a 2-bus datapath with one input bus (A bus) and one output bus (C bus).

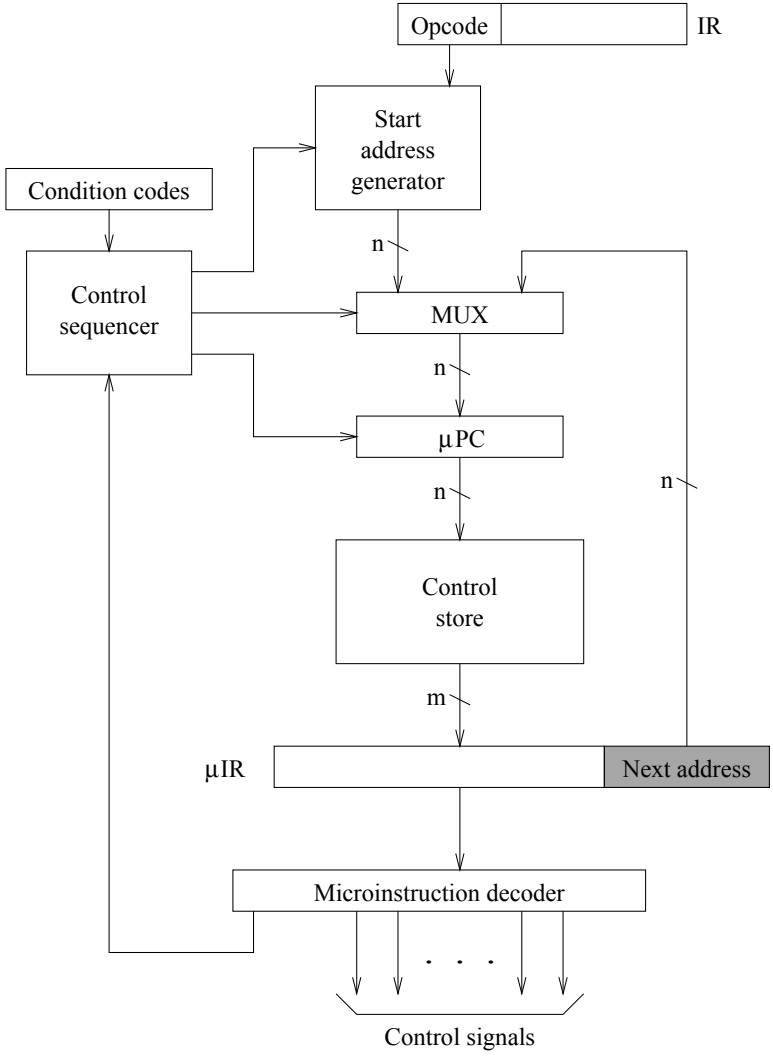


Figure 6.18 Microprogramming uses a control store to control the actions.

Since we have a separate output bus, we do not need the C register to capture the ALU output.

The availability of two buses reduces the time needed to execute instructions. To see the impact of two buses, let us implement the `add` instruction:

```
add     %G9,%G5,%G7
```

The microroutine for this instruction is shown below:

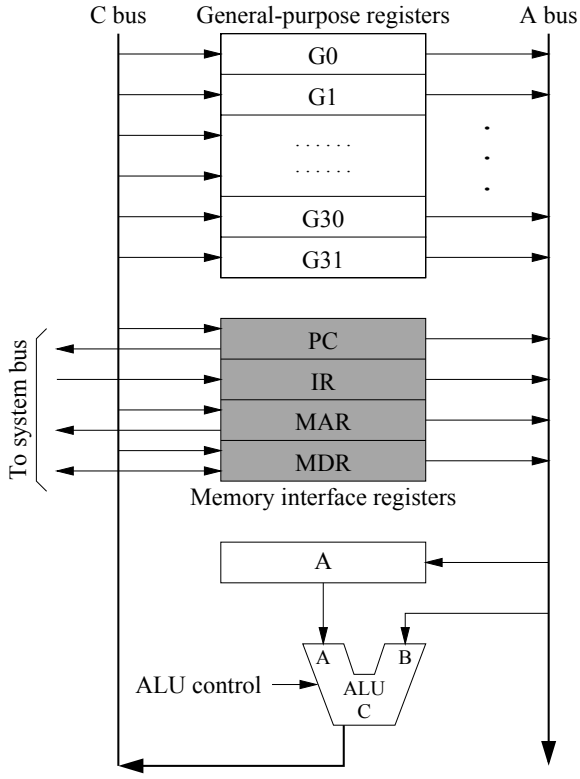


Figure 6.19 An example 2-bus datapath.

Instruction	Step	Control signals
`add %G9,%G5,%G7`	S1	`G5out: Ain;`
	S2	`G7out: ALU=add: G9in;`

Compared to the single-bus datapath, we reduce the number of steps by one. How do we execute this instruction on the 3-bus datapath shown on page 16? We just need one step to execute this instruction:

Instruction	Step	Control signals
`add %G9,%G5,%G7`	S1	`G5outA: G7outB: ALU=add: G9in;`

Notice that the 3-bus data path does not require the A and C registers. However, since the registers are connected to both A and B buses, we have to specify which bus should receive the register output. We use the notations `G?outA` and `G?outB` to indicate that register `G?` output should be placed on the A bus and B bus, respectively.

6.6 Performance

Measuring performance of a computer system is a complex task. It is easy to define a set of very complex performance metrics, but such metrics are not useful in practice. In addition, complex metrics make performance measurement difficult. What we need is a simple yet representative metric that captures the capabilities of the computer system for our usage. The keyword is *our usage*, which means we want a metric that takes into account the kind of applications we run on our system. For example, if we plan to run scientific applications involving lots of floating-point calculations, there is no point in knowing how many integer calculations a given machine can perform. Similarly, if our application almost always uses character manipulation, we don't find much use in knowing how many integer and floating-point calculations per second the system can do. Thus, it is important to take the expected mix of applications, also called the *workload*, into account and derive metrics that make sense for the target user group.

A workload typically consists of a suite of representative programs, which can be executed to measure the time. If this suite of applications represents the target user application mix reasonably, then we can compare the performance of different systems by comparing the execution times for this particular workload. Obviously, if machine X executes the workload in 300 seconds and machine Y takes 330 seconds, we say that machine X is better *for this workload*. You should note that, if we change the workload, it is quite possible that machine Y performs better than machine X for the new workload. The point to take away from this discussion is that the workload is important in comparing the performance of different machines.

If your company is evaluating the performance of two computer systems from competing manufacturers, you can run some typical programs from your application mix and compare the execution times. However, not every company is going spend time and effort in evaluating the machine they intend to buy. Furthermore, such a strategy does not always work. For example, if you are a designer working on a new processor, it is not feasible to run programs during the initial stages of the design process. Furthermore, there may not be an optimizing compiler available for you to run the programs. Thus, we need some general-purpose metrics that give us an idea of the relative performance of various systems.

We already mentioned that the workload is important. This implies that we should not define a single metric and use it for all purposes. Standard bodies define a set of *benchmark* programs that approximate the intended real-world applications. Benchmarks can be *real* programs taken from sample applications or *synthetic*. In synthetic benchmarks, artificial programs are created to exercise the system in a specific way. For example, the Whetstones and Dhrystones benchmarks, described later, are examples of synthetic benchmarks. In Section 6.6.4, we describe some real benchmarks from SPEC.

6.6.1 Performance Metrics

Computer system performance can be measured by several performance metrics. The metrics we use depend on the purpose as well as the component of the system in which we are interested. For example, if you are interested in the network component, we can use network bandwidth, which tells us the number of bits it can transmit per second. Two common metrics are used for almost all components: *response time* and *throughput*. Response time expresses the time needed to execute a task. For example, on a network, we may be interested in message delivery time. In this context, message delivery time represents the response time.

Throughput refers to the rate of flow. Looking at the network example again, throughput of the network represents the number of messages delivered per second. In this section, in order to limit the scope of our discussion, we focus on the processor.

For processors, response time represents the time it takes for a job to complete its execution. Response time includes the time to preprocess the job, any waiting time if the processor is busy with other jobs, and the actual execution time.

As you can see from this discussion, the response time metric is something in which a user is interested. When we say time, we usually mean the wall clock time, the amount of time the user had to wait to finish the job. This time consists of the actual CPU time spent on the job and waiting time that includes the time to access the disk and execute other jobs.

Throughput expresses the system capacity. For example, we say, "The system can execute 100 transactions per second," to express its capacity. Throughput is a system metric, whereas the response time is a single job metric. As a result of this orientation, users are interested in minimizing response times for their jobs, whereas a system administrator strives to get higher throughput from the system. Often, these two goals are conflicting. To minimize response time to your job, you don't want the system to run any other job, which wastes system resources. To increase throughput, you want to run a certain number of jobs concurrently so that the system resources are well utilized. A compromise is usually struck between these conflicting objectives. In Section 6.6.4, we show that both response time and throughput are used to characterize a system.

MIPS and MFLOPS are sometimes used as performance metrics. MIPS stands for millions of instructions per second. Although it is a simple metric, it is practically useless to express the performance of a system. Since instructions vary widely among the processors, a simple instruction execution rate will not tell us anything about the system. For example, complex instructions take more clocks than simple instructions. Thus, a complex instruction rate will be lower than that for simple instructions. The MIPS metric does not capture the actual work done by these instructions. MIPS is perhaps useful in comparing various versions of processors derived from the same instruction set.

MFLOPS is another popular metric often used in the scientific computing area. MFLOPS stands for millions of floating-point operations per second. This is a far better metric than MIPS as it captures the number of operations in which the user is interested. This measure also takes various system overheads to read operands, store results, and loop testing. We later look at more useful metrics.

6.6.2 Execution Time Calculation

The time required to execute a program represents an intrinsic measure of a processor's capability. Execution time depends on the following three factors:

- *Instruction Count (IC):* We need to know the number of instructions required to execute the program. Obviously, the more instructions a program takes, the more time it needs to execute the program. If all instructions of a processor take more or less the same amount of time to execute, a simple instruction count is sufficient. If, on the other hand, instruction execution times vary widely as in CISC processors, we need to get an effective instruction count.
- *Clocks per Instruction (CPI):* This represents time in terms of CPU clocks required for an average instruction. In RISC processors most instructions take the same number of clocks. In CISC processors, however, the clock count depends on the instruction type. In such cases, we can take the average value. We describe later a number of ways one can compute averages.
- *Clock Period (T):* Clock period is defined as the time taken by a single clock cycle.

Given these three factors, we can estimate the execution time of a program as

$$\text{Execution time} = IC \times CPI \times T\,. \tag{6.1}$$

We can then define performance of a system as

$$\text{Performance} = \frac{1}{\text{Execution time}}\,. \tag{6.2}$$

These three factors provide us with an understanding of the impact of various improvements on the performance. For example, we double the performance by increasing the clock rate from 500 MHz to 1 GHz, which reduces the clock period from 2 to 1 ns. In reality, application performance is dependent on many other factors including the number of other programs running on the system, the performance of the cache subsystem, and I/O subsystem latencies.

6.6.3 Means of Performance

We often want a single summarizing metric to get an idea of performance, even though we may conduct several experiments. Once the appropriate workload has been identified and the performance metric has been selected, we need to find a method to get a value for the performance metric. There are several ways of obtaining such a metric. We start with the simplest of all, the *arithmetic mean*. Suppose you run two programs to evaluate a system. If the individual execution times are 100 seconds (for Program 1) and 80 seconds (for Program 2), we compute the arithmetic mean as

$$\text{Mean execution time} = \frac{100 + 80}{2} = 90 \text{ seconds}.$$

In general, the arithmetic mean of n numbers $a_1, a_2, \ldots, a_n$ is computed as

$$\text{Arithmetic mean} = \frac{1}{n}\sum_{i=1}^{n} a_i\,,$$

where $\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n$.

There is one implicit assumption in our arithmetic mean calculation of the two programs: We assume that both programs appear equally likely in the target workload. Now suppose we know that Program 2 appears three times more often than Program 1. What would be the summary execution time that reflects this reality? Of course, we want to give three times more weight to the execution time of Program 2. That is, the mean is computed as

$$\text{Mean execution time} = \frac{1 \times 100 + 3 \times 80}{4} = 85 \text{ seconds.}$$

This is called the *weighted arithmetic mean*. This computation assigns a weight for each value. This fact becomes clear if we rewrite the previous expression as

$$\text{Mean execution time} = \frac{1}{4} \times 100 + \frac{3}{4} \times 80 = 85 \text{ seconds.}$$

This expression clearly shows that Program 1 execution time is given a weight of 1/4 and the other program 3/4. The general formula is

$$\text{Weighted mean execution time} = \sum_{i=1}^{n} w_i \times a_i\,,$$

where w_i is the weight expressed as a fraction. In our example, Program 1 has a weight of 25% and Program 2 has 75%. We express these weights as $w_1 = 0.25$ and $w_2 = 0.75$. Note that all weights should add up to 1. That is, $\sum_{i=1}^{n} w_i = 1$. The normal arithmetic mean is a special case of the weighted arithmetic mean with equal weights.

The weighted arithmetic mean is fine for metrics such as the response time to look at the performance of a single system. When comparing relative performance of two systems, it does cause problems. Let's assume that the response time of each machine is expressed relative to a reference machine. For example, most performance metrics from SPEC are expressed as a ratio relative to a reference machine. Table 6.6 shows an example to demonstrate the problems in using the arithmetic means. It lists the execution times of two programs on two machines (A and B) and a reference machine (REF). The first two columns under "Normalized values" give response time values normalized to the reference machine. That is, these values are obtained by dividing the response time of machines A and B by the corresponding response times for the reference machine.

When we use the arithmetic mean, we get 30.25 and 36 for machines A and B, respectively. The next column, labeled "Ratio," gives the ratio of B over A (i.e., 36/30.25) as 1.19. When we compute the corresponding ratio using the normalized values, we get 1.16. Clearly, there is a mismatch between the two values.

Table 6.6 Arithmetic versus geometric mean

	Response time on machine				Normalized values		
	REF	A	B	Ratio	A	B	Ratio
Program 1	10	11	12		1.1	1.2	
Program 2	40	49.5	60		1.24	1.5	
Arithmetic mean		30.25	36	1.19	1.17	1.35	1.16
Geometric mean		23.33	26.83	1.15	1.167	1.342	1.15

This is where the geometric mean is useful. The geometric mean of n numbers $a_1, a_2, \ldots, a_n$ is defined as

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} a_i} \qquad \text{or} \qquad \left(\prod_{i=1}^{n} a_i\right)^{1/n},$$

where $\prod_{i=1}^{n} a_i = a_1 \times a_2 \times \cdots \times a_n$. When we use the geometric mean, we get a matching value of 1.15 for the two ratios computed from the normalized and the original values. This is because the geometric mean has the property

$$\frac{\text{Geometric mean}(a_i)}{\text{Geometric mean}(b_i)} = \text{Geometric mean}\left(\frac{a_i}{b_i}\right).$$

Analogous to the weighted arithmetic mean, we can also define the weighed geometric mean as

$$\text{Weighted geometric mean} = \prod_{i=1}^{n} {a_i}^{w_i},$$

where w_i is the weight as defined in the weighted arithmetic mean. The geometric mean can be used to maintain consistency in summarizing normalized results. Unfortunately, geometric means do not predict execution times. To see this, consider the execution times of two machines, A and B, shown in Table 6.7. The arithmetic mean says that Machine A is about three times faster than Machine B. On the other hand, the geometric mean suggests that both machines perform the same. Why? The geometric mean keeps track of the performance ratio. Since Program 1 runs 10 times faster on Machine A and Program B runs 10 times faster on Machine B, by using the geometric mean we erroneously conclude that the average performance of the two programs is the same.

The geometric mean, however, is useful when our metric is a ratio, like the throughput. For example, the SPECviewperf benchmark from SPEC, which measures the 3D rendering performance of systems running under OpenGL, uses the weighted arithmetic mean [34]. This benchmark uses a throughputlike measure (frames/second) as the unit.

Table 6.7 An example to demonstrate the drawback of the arithmetic mean

	Response time on machine	
	A	B
Program 1	20	200
Program 2	50	5
Arithmetic mean	35	102.5
Geometric mean	31.62	31.62

6.6.4 The SPEC Benchmarks

We mentioned two types of benchmarks: synthetic and real. Synthetic benchmarks are programs specifically written for performance testing. Whetstone and Dhrystone benchmark programs are two example synthetic benchmarks. The *Whetstones* benchmark, named after the Whetstone Algol compiler, was developed in the mid-1970s to measure floating-point performance. The performance is expressed in MWIPS, millions of Whetstone instructions per second. The *Dhrystone* benchmark was developed in 1984 to measure integer performance. Both these benchmarks are small programs. A drawback with these benchmarks is that they encouraged excessive optimization by compilers to distort the performance results.

As computer systems become more complex, we need to measure performance of various components for different types of applications. The Standard Performance Evaluation Corporation (SPEC) was formed as a nonprofit consortium consisting of computer vendors, system integrators, universities, research organizations, publishers, and consultants. The objective is to provide benchmarks to measure performance of components as well as the system as a whole for multiple operating systems and environments. These benchmarks would be based on real-world applications. To give you an idea of the types of benchmarks provided, we describe some sample benchmarks next.

SPEC CPU2000

This benchmark is used for measuring the processor performance, memory, and compiler. The previous version, CPU95, was retired at the end of June, 2000. For this benchmark, applications are classified as "integer" if they spend less than 1% of their time performing floating-point calculations. This definition covers most nonscientific applications such as compilers, utilities, and simulators [34, 18]. SPEC CPU2000 consists of 26 applications that span four languages: C, C++, FORTRAN 77, and FORTRAN 90. SPEC CPU2000 consists of integer and floating-point components.

Table 6.8 SPEC CINT2000 integer benchmarks

Benchmark	Language	Description
164.gzip	C	Compression (A GNU data compression program)
175.vpr	C	Integrated circuit computer-aided design program (It performs field-programmable gate arrays (FPGA) circuit placement and routing.)
176.gcc	C	Compiler (GNU C compiler)
181.mcf	C	Combinatorial optimization program (It performs single-depot vehicle scheduling in public mass transportation.)
186.crafty	C	Game-playing program (chess)
197.parser	C	Word-processing program (a syntactic parser of English, which has a dictionary of about 60,000 word forms)
252.eon	C++	Computer visualization program (a probabilistic raytracer)
253.perlbmk	C	PERL programming language (The reference workload consists of four scripts.)
254.gap	C	Group theory (an interpreter used to implement a language and library designed mostly for computing in groups)
255.vortex	C	Object-oriented database (a single-user object-oriented database transaction benchmark)
256.bzip2	C	Compression (This is based on Julian Seward's bzip2 version 0.1.)
300.twolf	C	Place and route simulator (a placement and global routing package used for creating the microchip lithography artwork)

CINT2000: This is an integer benchmark to measure the performance for integer operations. This benchmark consists of the 12 applications shown in Table 6.8.

CFP2000: This is a floating-point benchmark that measures the performance for floating-point operations. It consists of the 14 applications shown in Table 6.9. As you can see from this list, these applications are all derived mainly from a scientific computation workload.

Performance is expressed relative to a reference machine, which is a 300 MHz Sun Ultra 5. This machine gets a score of 100. Integer and floating-point performance of various Pentium III and 4 processors is shown in Figure 6.20.

Table 6.9 SPEC CFP2000 floating-point benchmarks

Benchmark	Language	Description
168.wupwise	FORTRAN 77	Physics/quantum chromodynamics
171.swim	FORTRAN 77	Shallow water modeling
172.mgrid	FORTRAN 77	Multigrid solver (3D potential field)
173.applu	FORTRAN 77	Parabolic/elliptic partial differential equations
177.mesa	C	3D graphics library
178.galgel	FORTRAN 90	Computational fluid dynamics
179.art	C	Image recognition/neural networks
183.equake	C	Seismic wave propagation simulation
187.facerec	FORTRAN 90	Image processing (face recognition)
188.ammp	C	Computational chemistry
189.lucas	FORTRAN 90	Number theory/primality testing
191.fma3d	FORTRAN 90	Finite-element crash simulation
200.sixtrack	FORTRAN 77	High energy nuclear physics accelerator design
301.apsi	FORTRAN 77	Meteorology (pollutant distribution)

SPECmail2001

This is a standardized mail server benchmark designed to measure a system's ability to service email requests. It was developed by mail server vendors and research organizations to enable performance evaluation of systems supporting the Post Office Protocol (POP3) and Simple Mail Transfer Protocol (SMTP). This benchmark uses both throughput and response time to characterize a mail server system with realistic network connections, disk storage, and client workloads. The benchmark focuses on the ISPs with 10,000 to 1,000,000 users. It can also be used by vendors to test and finetune products under development.

Results from SPECmail2001 are based on a messages-per-minute rating that indicates the load the mail server can sustain with a reasonable quality of service. For example, Mirapoint MSR 2.8 has a SPECMail2001 rating of 2000 messages/minute. It uses a single 400 MHz Pentium II processor with 32 KB of primary cache (16 KB of instruction cache and 16 KB data cache) and 512 KB of secondary cache.

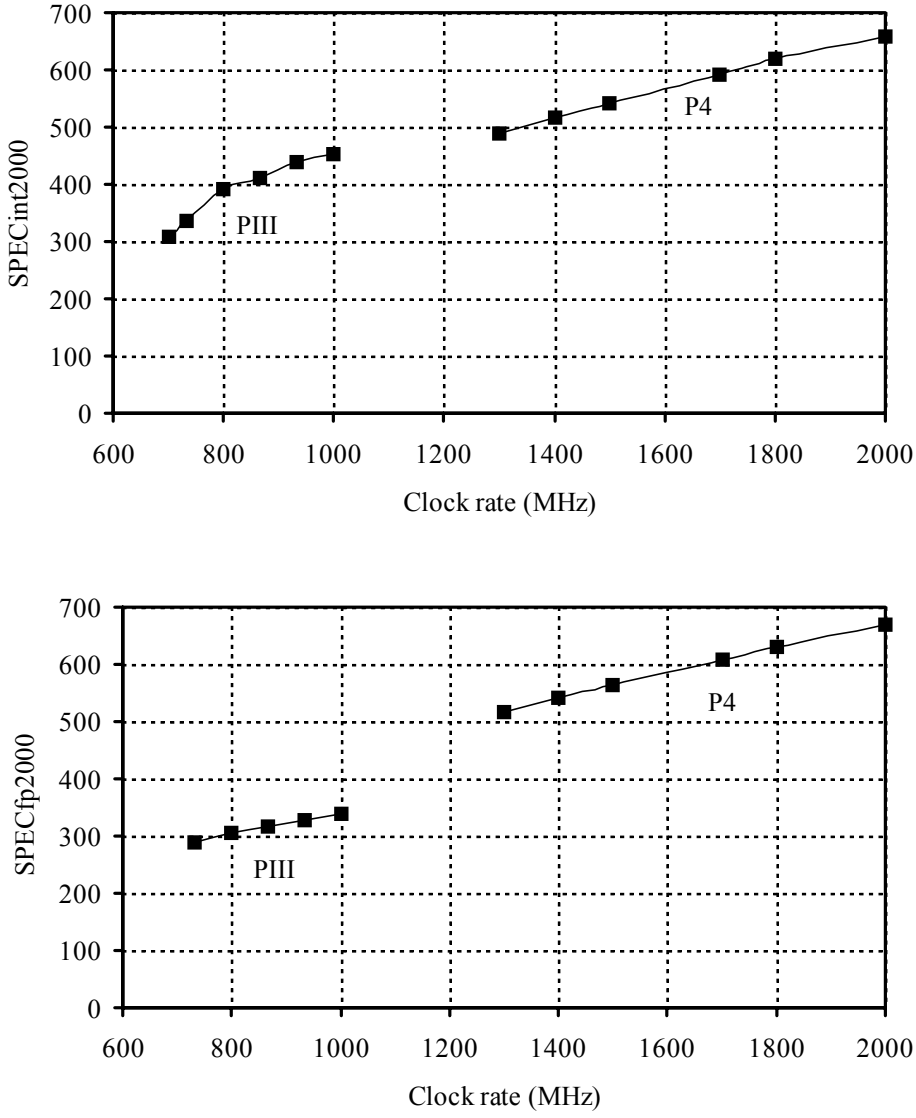


Figure 6.20 SPEC CPU2000 scores for Pentium III and 4 processors.

SPECMail2001 also specifies a response time limit for various actions such as SMTP connect and POP delete, as shown in Table 6.10. The percentage compliance rating of MSR 2.8 is also shown in this table.

Table 6.10 SPECMail2000 results for Mirapoint MSR 2.8 mail server system

Function	Response time limit (seconds)	Required percentage compliance (%)	Percentage Compliance (%) 80%	100%	120%
SMTP Connect	5	> 95	100.00	100.00	98.02
SMTP Data	5	> 95	100.00	100.00	100.00
POP Connect	5	> 95	100.00	100.00	100.00
POP Status	5	> 95	100.00	100.00	100.00
POP Retrieve	5	> 95	100.00	100.00	100.00
POP Delete	5	> 95	100.00	100.00	100.00
Delivery Time	60	> 95	99.14	99.01	95.60
Error Rate	N/A	< 1	0.15	0.14	1.13

SPECweb99

SPECweb99 is a benchmark used to measure the performance of HTTP servers. It measures a server's ability to handle HTTP GET requests from a number of external client drivers. The metric used is the number of simultaneous connections that conform to the specified bit rate limits. Each test is repeated three times for a reported result. The SPECweb99 metric is the median result for the three iterations.

As an example of the SPECweb99 benchmark, we present the values for the Sun Fire 4810 that runs the iPlanet Web Server 6.0. This server, which uses 12 750 MHz UltraSPARC III with 96 KB of primary cache (32 KB instruction cache and 64 KB data cache) and 8 MB of secondary cache, has a SPECweb99 rating of 8739 simultaneous connections. It uses 12 gigabit Ethernets to support the Web activities. Table 6.11 gives the three iteration results for the throughput and response time.

SPECjvm98

SPECjvm98 is the Java Virtual Machine benchmark suite that allows users to evaluate performance of the JVM client platform. This benchmark evaluates performance of both hardware and software components. It measures the efficiency of software components such as the JVM and the just-in-time (JIT) compiler. It also takes into account hardware components including the performance of the CPU for integer and floating-point operations, cache, and memory.

The SPECjvm98 benchmark suite consists of eight different applications. Five of these are either real applications or derived from real applications. Each test measures the time it takes to

Table 6.11 Sun Fire SPECWeb99 results

Iteration	Conforming connections	Conformance (%)	Throughput (operations/sec)	Response time (msec)
1	8749	100.0	24414.4	358.2
2	8739	99.9	24188.4	361.5
3	8739	99.9	24184.7	361.6

load the program, verify the class files, compile on the fly if a JIT compiler is used, and execute the test. Each test is run several times and a geometric mean is used to compute a composite score for all tests. Test scores are normalized against a reference machine: a midrange IBM PowerPC 604 with a 133 MHz processor.

6.7 Summary

When designing a processor, several design choices will have to be made. These choices are dictated by the available technology as well as the requirements of the target user group. Processor designers will have to make compromises in order to come up with the best design. This chapter looked at some of the important design issues involved in such an endeavor. Other design issues are covered in the rest of the book.

Here we looked at how the processor design at the ISA level gets affected by various design choices. We stated that the number of addresses in an instruction is one of the choices that can have an impact on the instruction set design. It is possible to have zero-, one-, two-, or three-address instruction sets; however, most recent processors use the three-address format. The Pentium, on the other hand, uses the two-address format.

The addressing mode is another characteristic that affects the instruction set. RISC processors tend to use the load/store architecture and use simple addressing modes. Often, these processors support just two addressing modes. CISC processors such as the Pentium provide a wide variety of addressing modes.

Both of these choices—number of addresses and the complexity of addressing modes—affect the instruction format. RISC processors use fixed-length instructions because they use the load/store architecture and support simple addressing modes. CISC processors use variable-length instructions to accommodate various complex addressing modes.

We also looked at how the instructions are executed in the underlying hardware. The hardware consists of a datapath with one, two, or three internal buses. We have seen the tradeoffs associated with the three types of datapaths. For simple instruction sets, typically used by RISC processors, necessary control signals for the datapath can be generated by the hardware. For complex instruction sets used by CISC processors, a software-based approach called microprogram control is used. We have discussed in detail how the microprogrammed control works.

In the last section, we covered processor performance. We introduced the concept of clocks per instruction and how it can be used to estimate the execution time of a program. We provided information on quantifying the performance of processors. Synthetic benchmarks tend to be exploited to produce skewed performance results. The recent trend is to use real application-based benchmarks to evaluate performance. Furthermore, benchmarks are specialized to the target application. For example, there is a benchmark for mail servers, another for web servers, and so on. To give you a concrete idea, we have presented several example benchmarks proposed by the SPEC consortium.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- 0-address machines
- 1-address machines
- 2-address machines
- 3-address machines
- Absolute address
- Accumulator machines
- Addressing modes
- Arithmetic instructions
- Conditional branch
- Data movement instructions
- Delayed procedure call
- End of procedure
- Flow control instructions
- Immediate addressing mode
- Input/output instructions
- Instruction format
- Instruction set design issues
- Instruction types
- Isolated I/O
- Load/store architecture
- Load/store instructions
- Logical instructions
- Memory-mapped I/O
- Microcontroller
- Microprogrammed control
- Number of addresses
- Opcode
- Operand types
- Parameter passing
- PC-relative
- Procedure call
- Processor registers
- Register addressing mode
- Return address
- Stack depth
- Stack machines
- Unconditional branch
- Wait cycles

6.8 Exercises

6–1 We have discussed instructions with zero to three addresses. Discuss why modern RISC processors use the three-address format.

6–2 The Pentium processor uses the two-address format. Do you support the decision made by the Pentium designers in this regard? Justify your answer.

6–3 Discuss the advantages of the load/store architecture. Focus your discussion on why current RISC processors use this architecture.

6–4 In Section 6.2.5, we have stated that 19 memory accesses are required to execute the example expression under the assumption that the push and pop operations do not require any memory accesses. Calculate the number of memory references required if the stack depth is zero (i.e., all push/pop operations require memory access).

6–5 RISC processors tend to have a large number of registers compared to CISC processors. Explain why.

6–6 What is the difference between normal and delayed branch execution? Why do some processors use delayed branch execution?

6–7 Conditional branching can be done in one of two basic ways: *set-then-jump* or *test-and-jump*. Discuss the advantages and disadvantages of these two methods.

6–8 During a procedure invocation, the return address must be saved in order to send the control back to the calling program. Most RISC processors store the return address in a register, whereas the Pentium uses the stack to store the return address. Discuss the pros and cons of these two schemes.

6–9 Explain why RISC processors tend to use fixed-length instructions whereas the CISC processors such as the Pentium do not.

6–10 We stated that the Pentium does not allow both operands to be located in memory. Explain the rationale for this.

6–11 In the single-bus datapath shown in Figure 6.7 (page 220), both the PC and IR registers are connected to the system bus. This allows the PC register to place the address on the system bus and the IR register to receive the instruction from the system bus. Suppose that these two registers are connected only to the A bus as are the general-purpose registers. Describe the steps involved in placing the PC address on the system bus. Also explain how the IR register will receive the instruction. What impact would this modification have on the processor performance?

6–12 Suppose we want to implement the instruction

```
shl4    %G7,%G5
```

on the single-bus datapath shown in Figure 6.7. This instruction shifts the contents of `G5` by four bit positions and stores the result in `G7`. Show how this instruction is implemented using the table format we used for the `add` instruction.

6–13 Our example ALU does not have a multiply by 10 function. Show how we can implement the following instruction:

```
mul10   %G7,%G5
```

This instruction multiplies the contents of `G5` by 10 and places the result in `G7`. Assume that there will be no overflow. How many cycles do you need to implement this instruction?

6–14 On the 2-bus datapath shown in Figure 6.19 (page 235) implement the data movement instruction

```
mov     %G7,%G5
```

to copy contents of `G5` to `G7`.

6–15 What is wrong with performance metrics like MIPS? What are the circumstances in which they are useful to compare the performance of processors?

6–16 What are real and synthetic benchmarks? Why is there a preference for the use of real benchmarks in current standards?

6–17 What are the major problems with synthetic benchmarks such as Whetstones?

6–18 What is the need for having so many specialized benchmarks such as SPECmail2001 and SPECweb99?

Chapter 7

The Pentium Processor

Objectives

- To describe the basic organization of the Intel Pentium processor;
- To introduce the Pentium real mode memory organization;
- To discuss the protected mode memory architecture.

We discussed processor design space in the last chapter. Now we look at Pentium processor details. We present details of its registers and memory architecture. Other Pentium details are discussed in later chapters.

We start our discussion with a brief history of the Intel architecture. This architecture encompasses the X86 family of processors. All these processors, including the Pentium, belong to the CISC category. In the following section, we describe the Pentium processor signals. Section 7.3 presents the internal register details of the Pentium. Even though the Pentium is a 32-bit processor, it maintains backward compatibility to the earlier 16-bit processors. The next two sections describe the real and protected mode memory architectures. The real mode is provided to mimic the 16-bit 8086 memory architecture. Protected mode architecture is the native mode for the Pentium. In both modes, the Pentium supports segmented memory architecture. In the protected mode, it also supports paging to facilitate implementation of virtual memory. It is important for an assembly language programmer to understand the segmented memory organization supported by the Pentium. We conclude the chapter with a summary.

7.1 The Pentium Processor Family

Intel introduced microprocessors way back in 1969. Their first 4-bit microprocessor was the 4004. This was followed by the 8080 and 8085. The work on these early microprocessors led to the development of the Intel architecture (IA). The first processor in the IA family was the 8086 processor, introduced in 1979. It has a 20-bit address bus and a 16-bit data bus.

The 8088 is a less expensive version of the 8086 processor. The cost reduction is obtained by using an 8-bit data bus. Except for this difference, the 8088 is identical to the 8086 processor. Intel introduced segmentation with these processors. These processors can address up to four segments of 64 KB each. This IA segmentation is referred to as the real mode segmentation and is discussed in detail later in this chapter.

The 80186 is a faster version of the 8086. It also has a 20-bit address bus and 16-bit data bus, but has an improved instruction set. The 80186 was never widely used in computer systems. The real successor to the 8086 is the 80286, which was introduced in 1982. It has a 24-bit address bus, which implies 16 MB of memory address space. The data bus is still 16 bits wide, but the 80286 has some memory protection capabilities. It introduced the protection mode into the IA architecture. Segmentation in this new mode is different from the real mode segmentation. We present details on this new segmentation later. It is backwards compatible in that it can run the 8086-based software.

Intel introduced its first 32-bit CPU—the 80386—in 1985. It has a 32-bit data bus and 32-bit address bus. The memory address space has grown substantially (from 16 MB address space to 4 GB). This processor introduced paging into the IA architecture. It also allowed definition segments as large as 4 GB. This effectively allowed for a "flat" model (i.e., effectively turning off segmentation). Later sections present details on this. Like the 80286, it can run all the programs written for 8086 and 8088 processors.

The Intel 80486 was introduced in 1989. This is an improved version of the 80386. While maintaining the same address and data buses, it combined the coprocessor functions for performing floating-point arithmetic. The 80486 processor has added more parallel execution capability to instruction decode and execution units to achieve a scalar execution rate of one instruction per clock. It has an 8 KB onchip L1 cache. Furthermore, support for the L2 cache and multiprocessing has been added. Later versions of the 80486 processors incorporated features such as energy savings for notebooks.

The latest in the family is the Pentium series. It is not named 80586 because Intel found belatedly that numbers couldn't be trademarked! The first Pentium was introduced in 1993. The Pentium is similar to the 80486 but uses a 64-bit wide data bus. Internally, it has 128- and 256-bit wide datapaths to speed internal data transfers. However, the Pentium instruction set supports 32-bit operands like the 80486. The Pentium has added a second execution pipeline to achieve superscalar performance by having the capability to execute two instructions per clock. It has also doubled the onchip L1 cache, with 8 KB for data and another 8 KB for the instructions. Branch prediction (discussed in the next chapter) has also been added.

The Pentium Pro processor has a three-way superscalar architecture. That is, it can execute three instructions per CPU clock. The address bus has been expanded to 36 bits, which gives it an address space of 64 GB. It also provides dynamic execution including out-of-order and speculative execution. These features are discussed in Chapter 14. In addition to the L1 caches provided by the Pentium, the Pentium Pro has a 256 KB L2 cache in the same package as the CPU.

The Pentium II processor has added multimedia (MMX) instructions to the Pentium Pro architecture. It has expanded the L1 data and instruction caches to 16 KB each. It has also

Table 7.1 Key characteristics of the IA family of processors ("Year" refers to the year of introduction; "Frequency" refers to the frequency at introduction)

Processor	Year	Frequency (MHz)	Transistor count	Register width	Data bus width	Maximum address space
8086	1979	8	29 K	16	16	1 MB
80286	1982	12.5	134 K	16	16	16 MB
80386	1985	20	275 K	32	32	4 GB
80486	1989	25	1.2 M	32	32	4 GB
Pentium	1993	60	3.1 M	32	64	4 GB
Pentium Pro	1995	200	5.5 M	32	64	64 GB
Pentium II	1997	266	7 M	32	64	64 GB
Pentium III	1999	500	8.2 M	32	64	64 GB

added more comprehensive power management features including Sleep and Deep Sleep modes to conserve power during idle times. Table 7.1 summarizes the key characteristics of the IA family of processors.

Intel's 64-bit Itanium processor is targeted for server applications. For these applications, the Pentium's memory address space is not adequate. The Itanium uses a 64-bit address bus to provide substantially large address space. Its data bus is 128 bits wide. In a major departure, Intel has moved from the CISC designs of Pentium processors to RISC orientation for their Itanium processors. The Itanium also incorporates several advanced architectural features to provide improved performance for the high-end server market. We discuss Itanium details in Chapter 14.

In the rest of the chapter, we look at the basic architectural details of the Pentium processor. Our focus is on the internal registers and memory architecture. Other Pentium details are covered in later chapters. For example, Chapter 20 discusses its interrupt processing details, and Chapter 17 gives details on its cache organization.

7.2 The Pentium Processor

A block diagram of the Pentium showing the major signal groups is given in Figure 7.1. In the following, we describe these signals. As mentioned in Chapter 2, the pound sign (#) is used to indicate a low-active signal:

Data Bus (D0 to D63): This is the 64-bit data bus. The least significant byte is on data lines D0 to D7, and the most significant byte is on lines D56 to D63.

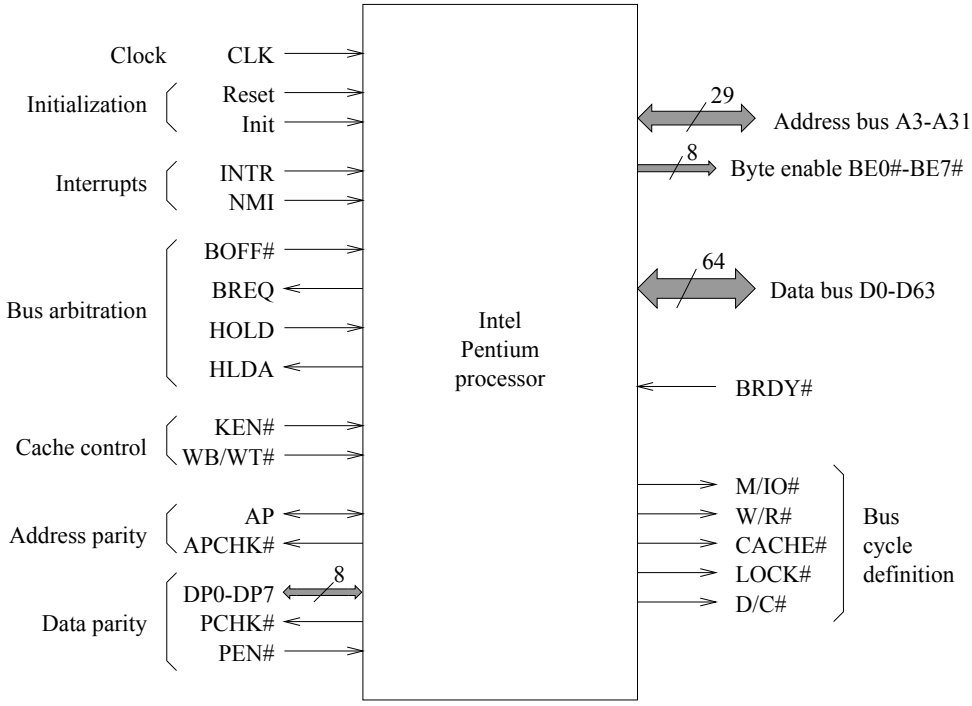


Figure 7.1 Selected signals of the Pentium processor.

Address Bus (A3 to A31): These 29 lines represent the address bus. Since the data bus is eight bytes wide, the least significant three address lines are not present. However, to precisely identify a set of bytes to read or write, the byte enable signals (described next) are used. These address lines are normally output signals. But external devices can drive these signals to perform inquire cycles. These cycles use only address lines A5 to A31.

Byte Enable (BE0# to BE7#): These low-active signals identify the set of bytes to read or write. Each byte enable signal identifies a byte (BE0 applies to D0 to D7, BE1 to D8 to D15, . . ., BE7 to D56 to D63). Since each byte is individually identified, any combination of the bytes can be specified.

Data Parity (DP0 to DP7): These eight lines are used to encode even parity for the eight data bytes. There is one bit for each data byte on the data bus: DP0 applies to D0 to D7, DP1 to D8 to D15, and so on.

Parity Check (PCHK#): This signal indicates the result of a parity check on the data read by the processor. Parity is checked only for the valid bytes (indicated by the byte enable signals).

Parity Enable (PEN#): This signal determines whether the parity check should be used. If enabled, an exception is taken on a parity error in case of a data read.

Address Parity (AP): This signal represents the even parity for the address lines A5 to A31. Address lines A3 and A4 are not included in the parity determination.

Address Parity Check (APCHK#): This signal indicates a bad address parity during inquire cycles.

Memory/IO (M/IO#): This control signal defines the bus cycle as either memory (M/IO# = 1) or I/O (M/IO# = 0) cycle.

Write/Read (W/R#): This control signal distinguishes between a read (W/R# = 0) and write (W/R# = 1) cycle.

Data/Code (D/C#): This control signal distinguishes a data (D/C# = 1) access from a code (D/C# = 0) access.

Cacheability (CACHE#): This output signal indicates internal cacheability of the current cycle if it is a read cycle; it indicates burst write-back in the case of a write cycle.

Bus Lock (LOCK#): This output signal indicates the processor's current sequence of bus cycles should not be interrupted. It also indicates that the processor is running a read-modify-write cycle (e.g., when executing a test-and-set type of instruction) where the external bus should not be relinquished between the read and write cycles. LOCK# is typically used to implement memory-based semaphores.

Interrupt (INTR): This input pin receives the external interrupt signal. As we show later in this chapter, the processor will process the interrupt if the interrupt enable flag (IF) is set in the EFLAGS register. For this reason, it is called the *maskable* interrupt.

Nonmaskable Interrupt (NMI): The Pentium receives external nonmaskable interrupt on this pin. Interrupts are discussed in Chapter 20.

Clock (CLK): This pin receives the system clock that provides the fundamental timing information to the processor. Other signals are sampled with reference to the clock signal.

Burst Ready (BRDY#): This input signal is used by external devices to extend the bus cycle (i.e., to introduce wait states, as discussed in Chapter 5).

Bus Request (BREQ): The Pentium asserts this signal whenever a bus cycle is pending internally. This signal is used by external logic for bus arbitration (see Chapter 5).

Backoff (BOFF#): This input signal causes the Pentium to abort all pending bus cycles and float the processor bus in the next clock. The processor remains in this state until BOFF# is removed. The processor then restarts the aborted bus cycles. This signal can be used to resolve deadlock between two bus masters.

Bus Hold (HOLD): This input signal will cause the Pentium to complete any outstanding bus cycles and float most of the output and input/output pins of the processor bus and assert HLDA (discussed next). This signal allows another bus master complete control of the processor bus.

Bus Hold Acknowledge (HLDA): This signal becomes active in response to the HOLD input. HLDA indicates that the Pentium has given the bus another local bus master. The Pentium continues with its execution from internal caches during the hold period.

Cache Enable (KEN#): This input signal is used to determine whether the system can support a cache line fill during the current cycle. If this signal and CACHE# are asserted, the current cycle is transformed into a cache line fill. These signals do not make any sense now as we have not covered caches yet. We discuss cache memories in Chapter 17.

Write-Back/Write-Through (WB/WT#): This input signal indicates that the cache line should use the write-back or write-through policy, on a line by line basis. We discuss cache write policies in Chapter 17.

Reset (RESET): This signal forces the Pentium to begin execution in a known state. After RESET, the Pentium starts execution at FFFFFFF0H. It invalidates all internal caches.

Initialization (INIT): Like the RESET signal, INIT also forces the Pentium to begin execution in a known state. However, internal caches and floating-point registers are not flushed. RESET, not INIT, should be used after powerup.

7.3 The Pentium Registers

The Pentium has ten 32-bit and six 16-bit registers. These registers are grouped into general, control, and segment registers. The general registers are further grouped into data, pointer, and index registers.

7.3.1 Data Registers

There are four 32-bit data registers that can be used for arithmetic, logical, and other operations (see Figure 7.2). These four registers are unique in that they can be used as follows:

- Four 32-bit registers (EAX, EBX, ECX, EDX); or
- Four 16-bit registers (AX, BX, CX, DX); or
- Eight 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL).

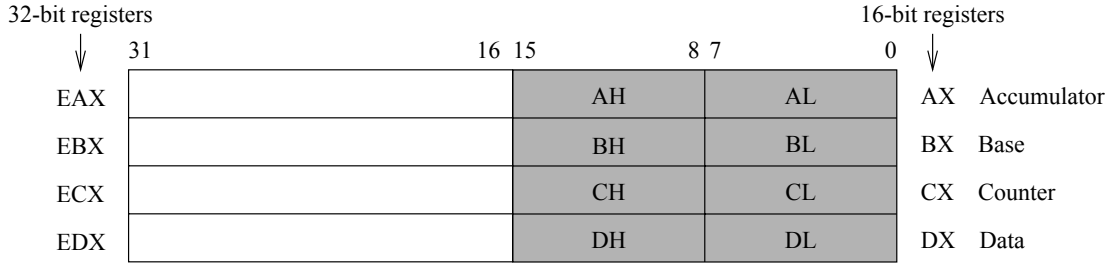


Figure 7.2 Data registers of the Pentium processor (16-bit registers are shown shaded).

As shown in Figure 7.2, it is possible to use a 32-bit register and access its lower half of the data by the corresponding 16-bit register name. For example, the lower 16 bits of EAX can be accessed by using AX. Similarly, the lower two bytes can be individually accessed by using the 8-bit register names. For example, the lower byte of AX can be accessed as AL and the upper byte as AH.

The data registers can be used without constraint in most arithmetic and logical instructions. However, some registers have special functions when executing specific instructions. For example, when performing a multiplication operation, one of the two operands should be in the EAX, AX, or AL register depending on the operand size. Similarly, the ECX or CX register is assumed to contain the loop count value for iterative instructions.

7.3.2 Pointer and Index Registers

Figure 7.3 shows the four 32-bit registers in this group. These registers can be used either as 16- or 32-bit registers. The two index registers play a special role in string processing instructions (discussed in Chapter 12). In addition, they can be used as general-purpose data registers.

The pointer registers are mainly used to maintain the stack. Even though they can be used as general-purpose data registers, they are almost exclusively used for maintaining the stack. The Pentium's stack implementation is discussed in Chapter 10.

7.3.3 Control Registers

This group of registers consists of two 32-bit registers: the instruction pointer register and the flags register. The processor uses the instruction pointer register to keep track of the location of the next instruction to be executed. The instruction pointer can be used either as a 16-bit register (IP), or as a 32-bit register (EIP). IP is used for 16-bit addresses and EIP for 32-bit addresses (see Sections 7.4 and 7.5 for details on the Pentium memory architecture).

When an instruction is fetched from memory, the instruction pointer is updated to point to the next instruction. This register is also modified during the execution of an instruction that transfers control to another location in the program (such as a jump, procedure call, or interrupt).

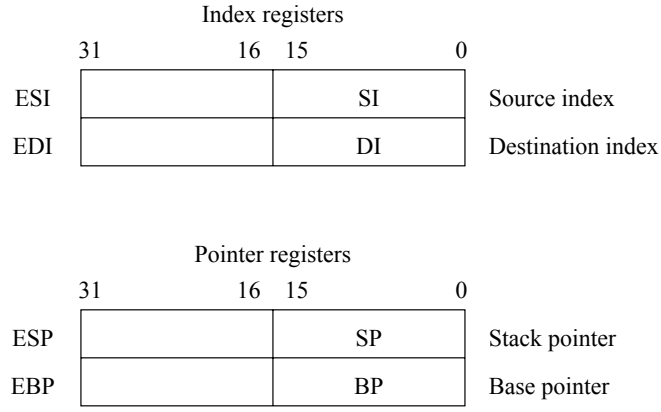


Figure 7.3 Index and pointer registers of the Pentium processor.

The flags register can be considered as either a 16-bit FLAGS register, or a 32-bit EFLAGS register. The FLAGS register is useful in executing 8086 processor code. The EFLAGS register consists of 6 *status* flags, 1 *control* flag, and 10 *system* flags, as shown in Figure 7.4. Bits of this register can be set (1) or cleared (0). The Pentium provides instructions to set and clear some flags. For example, the `clc` instruction clears the carry flag, and the `stc` instruction sets it.

The six status flags record certain information about the most recent arithmetic or logical operation. For example, if a subtract operation produces a zero result, the zero flag (ZF) bit would be set (i.e., ZF = 1). Chapter 12 discusses the status flags in detail.

The control flag is useful in string operations. This flag determines whether a string operation should scan the string in the forward or backward direction. The function of the direction flag is described in Chapter 12, which discusses the string instructions supported by the Pentium.

The 10 system flags control the operation of the processor. A detailed discussion of all 10 system flags is beyond the scope of this book. Here we briefly discuss a few flags in this group. The two interrupt enable flags—the trap enable flag (TF) and the interrupt enable flag (IF)—are useful in interrupt-related activities. For example, setting the trap flag causes the processor to single step through a program, which is useful in debugging programs. These two flags are covered in Chapter 20, which discusses the interrupt processing mechanism of the Pentium.

The ability to set and clear the identification (ID) flag indicates that the processor supports the CPUID instruction. The CPUID instruction provides information to software about the vendor (Intel chips use a "GenuineIntel" string), processor family, model, and so on. The virtual-8086 mode (VM) flag, when set, emulates the programming environment of the 8086 processor.

The last flag that we discuss is the alignment check (AC) flag. When this flag is set, the processor operates in alignment check mode and generates exceptions when a reference is made to an unaligned memory address. Section 16.7 provides further information on data alignment and its impact on application performance.

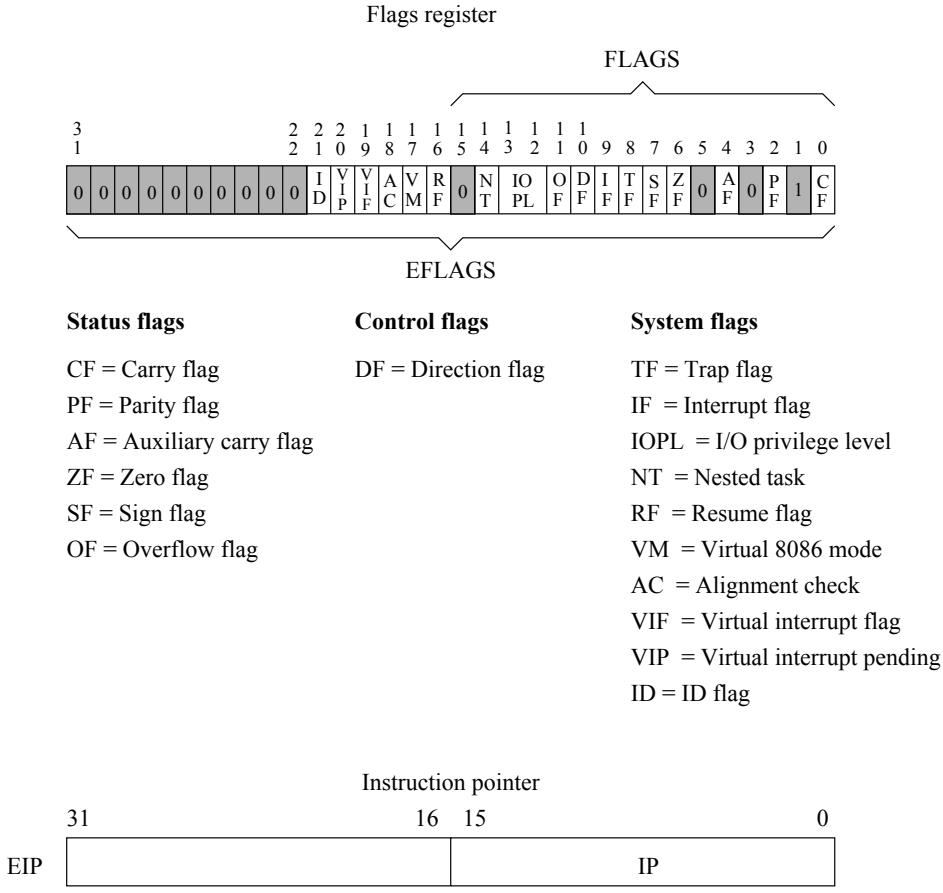


Figure 7.4 Flags and instruction pointer registers of the Pentium processor.

7.3.4 Segment Registers

The six 16-bit segment registers of the Pentium are shown in Figure 7.5. These registers support the segmented memory organization of the Pentium. In this organization, memory is partitioned into segments, where each segment is a small part of the memory. The processor, at any point in time, can only access up to six segments of the main memory. The six segment registers point to where these segments are located in the memory.

Your program is logically divided into two parts: a code part that contains only the instructions, and a data part that keeps only the data. The code segment (CS) register points to where your instructions are stored in the main memory, and the data segment (DS) register points to your data segment location. The stack segment (SS) register points to the program's stack segment (further discussed in Chapter 10).

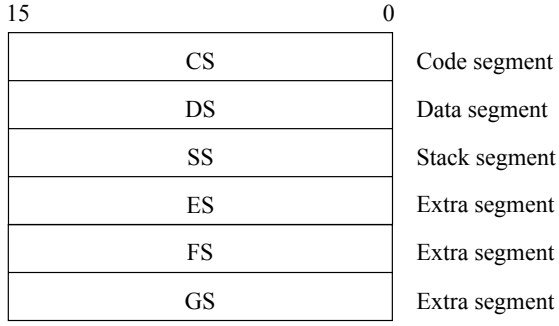


Figure 7.5 The six segment registers of the Pentium processor.

The last three segment registers—ES, GS, and FS—are additional segment registers that can be used in a similar way as the other segment registers. For example, if a program's data could not fit into a single data segment, we could use two data segment registers to point to the two data segments.

7.4 Real Mode Memory Architecture

The Pentium supports sophisticated memory architecture under real and protected modes. The real mode, which uses 16-bit addresses, is provided to run programs written for the 8086. In this mode, the Pentium supports the segmented memory architecture. The protected mode uses 32-bit addresses and is the native mode of the Pentium. In protected mode, the Pentium supports both segmentation and paging. Paging is useful in implementing virtual memory; it is transparent to the application program, but segmentation is not. We do not look at the paging features here. Paging details are presented in Chapter 18, which discusses virtual memory. We discuss protected mode memory architecture in the next section, and devote the rest of this section to describing the real-mode segmented memory architecture.

As mentioned, the Pentium behaves as a faster 8086 in the real mode. The memory address space of the 8086 processor is 1 MB. To address a memory location, we have to use a 20-bit address. The address of the first location is 00000H; the last addressable memory location is at FFFFFH. Recall that numbers expressed in the hexadecimal number system are indicated by suffix H (see Appendix A).

Since all registers in the 8086 are 16 bits wide, the address space is limited to 2^{16}, or 65,536 (64 K) locations. As a consequence, the memory is organized as a set of segments. Each segment of memory is a linear contiguous sequence of up to 64 K bytes. In this segmented memory organization, we have to specify two components to identify a memory location: a *segment base* and an *offset*. This two-component specification is referred to as the *logical address*. The segment base specifies the start address of a segment in memory and the offset specifies the address relative to the segment base. The offset is also referred to as the *effective address*. The relationship between the logical and physical addresses is shown in Figure 7.6.

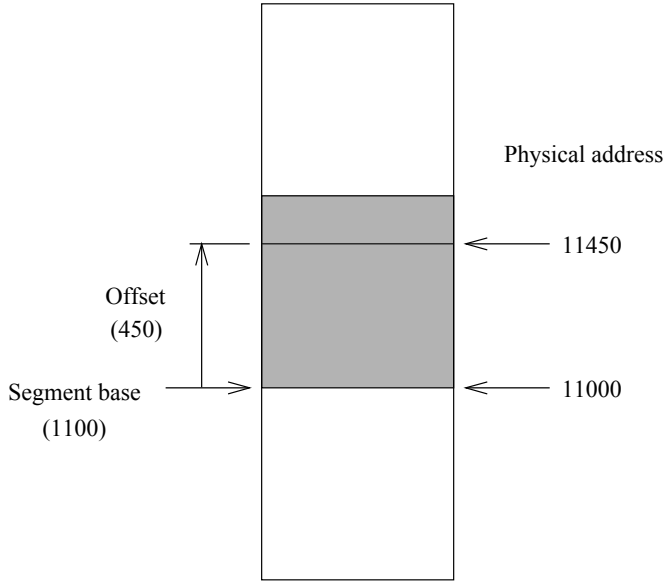


Figure 7.6 Relationship between logical and physical addresses of memory (all numbers are in hex).

Notice from Figure 7.6 that the segment base address is 20 bits long (11000H). So how can we use a 16-bit register to store the 20-bit segment base address? The trick is to store the most significant 16 bits of the segment base address and assume that the least significant four bits are all 0. In the example, we would store 1100H as the segment base. The implied four least significant zero bits are not stored. This trick works but imposes a restriction on where a segment can begin. Segments can begin only at those memory locations whose address has the least significant four bits as 0. Thus, segments can begin at 00000H, 00010H, 00020H, . . ., FFFE0H, FFFF0H. Segments, for example, cannot begin at 00001H or FFFEEH.

In the segmented memory organization, a memory location can be identified by its logical address. We use the notation *segment*:*offset* to specify the logical address. For example, 1100:450H identifies the memory location (i.e., 11450H), as shown in Figure 7.6. The latter value to identify a memory location is referred to as the *physical memory address*.

Programmers have to be concerned with logical addresses only. However, when the CPU accesses the memory, it has to supply the 20-bit physical memory address. The conversion of logical address to physical address is straightforward. This translation process, shown in Figure 7.7, involves adding four least significant zero bits to the segment base value and then adding the offset value. When using the hexadecimal number system, simply add a zero digit to the segment base address at the right and add the offset value. As an example, consider the logical address 1100:450H. The physical address is computed as follows:

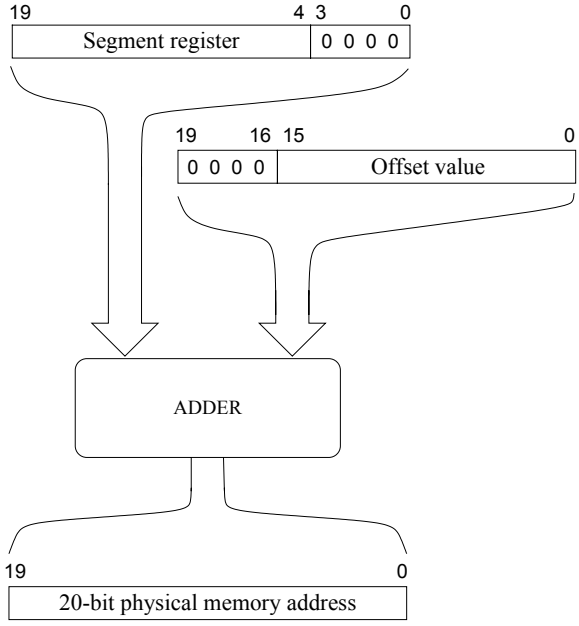


Figure 7.7 Physical address generation in the 8086.

```
  11000     (add 0 to the 16-bit segment base value)
+   450     (offset value)
  11450     (physical address).
```

For each logical memory address, there is a unique physical memory address. The converse, however, is not true. More than one logical address can refer to the same physical memory address. This is illustrated in Figure 7.8, where logical addresses 1000:20A9H and 1200:A9H refer to the same physical address 120A9H. In this example, the physical memory address 120A9H is mapped to two segments.

In our discussion of segments, we never said anything about the actual size of a segment. The main factor limiting the size of a segment is the 16-bit offset value, which restricts the segments to at most 64 K bytes in size. In the real mode, the Pentium sets the size of each segment to exactly 64 K bytes.

Programmers view the memory address space as a group of segments. At any instance, a program can access up to six segments. (The 8086 actually supported only four segments: segment registers FS and GS were not present in the 8086 processor.) Typically two of these segments contain code and data. The third segment is used for the stack. If necessary, other segments may be used, for example, to store data, as shown in Figure 7.9.

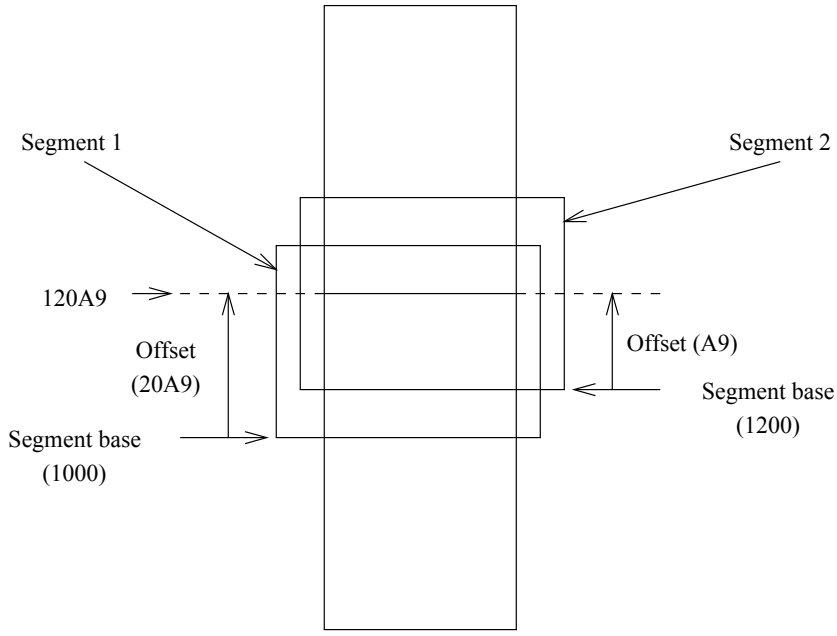


Figure 7.8 Two logical addresses map to the same physical address (all numbers are in hex).

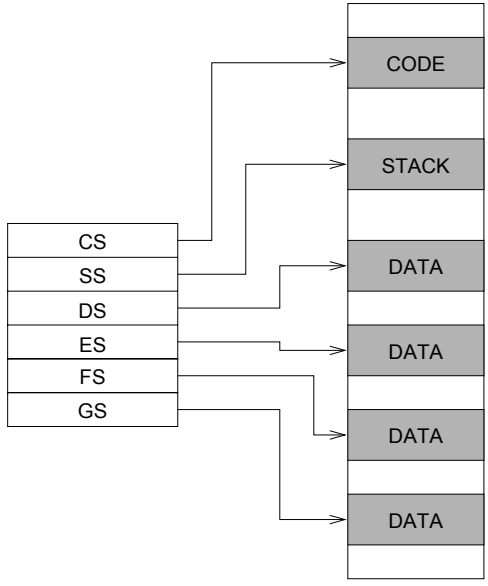


Figure 7.9 The six segments of the memory system.

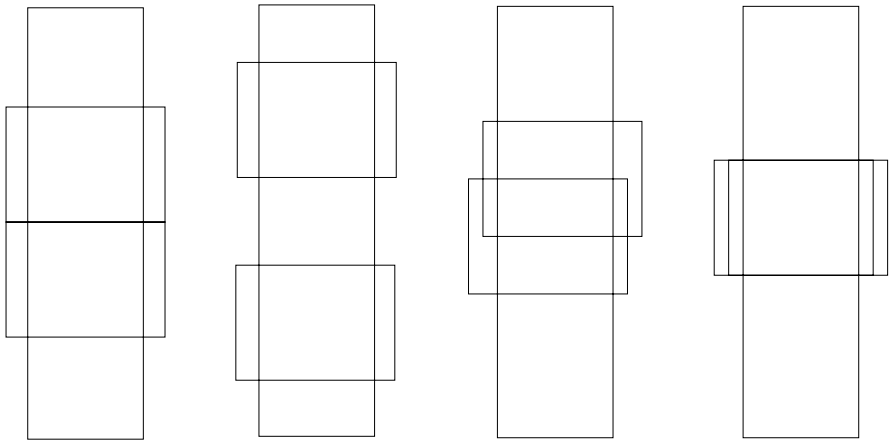


Figure 7.10 Various ways of placing segments in the memory.

Assembly language programs typically use at least two segments: code and stack segments. If the program has data (which almost all programs do), a third segment is also needed to store data. Those programs that require additional memory can use the other segments.

The six segment registers of the Pentium point to the six segments, as shown in Figure 7.9. As described earlier, segments must begin on 16-byte memory boundaries. Except for this restriction, segments can be placed anywhere in memory. The segment registers are independent and segments can be contiguous, disjoint, partially overlapped, or fully overlapped, as shown in Figure 7.10.

Even though programmers view memory as a group of segments and use the logical address to specify a memory location, all interactions between the processor and memory must use physical addresses. We have seen the process involved in translating a given logical address to the corresponding physical address (see page 261). The Pentium has dedicated hardware to perform the address translation, as illustrated in Figure 7.7.

Here is a summary of the real-mode memory architecture:

- Segments are exactly 64 K bytes in size.
- A segment register contains a pointer to the segment base.
- Default operand size and effective addresses are 16 bits long.
- Stack operations use the 16-bit SP register.
- Stack size is limited to 64 KB.
- Paging is not available. Thus, the processor uses the linear address as the physical address (see Figure 7.11).

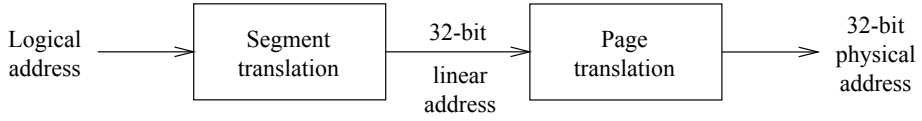


Figure 7.11 Logical to physical address translation process in the protected mode.

Keep in mind that this list gives the default attributes. It is, however, possible to change some of these defaults. Section 7.5.5 discusses how 32-bit operands and addresses can be used in the real mode.

7.5 Protected Mode Memory Architecture

In protected mode, the Pentium supports a more sophisticated segmentation mechanism in addition to paging. In this mode, the segment unit translates a logical address into a 32-bit linear address. The paging unit translates the linear address into a 32-bit physical address, as shown in Figure 7.11. If no paging mechanism is used, the linear address is treated as the physical address. In the remainder of this section, we focus on the segment translation process only. Paging is discussed in Chapter 18.

Protected mode segment translation is different from that in real mode. In real mode, the physical address is 20 bits long. The physical address is obtained directly from the contents of the selected segment register and the offset, as illustrated on page 261. In protected mode, contents of the segment register are taken as an index into a segment descriptor table to get a descriptor. The segment translation process is shown in Figure 7.12. Segment descriptors provide the 32-bit segment base address, its size, and access rights. To translate a logical address to the corresponding linear address, the offset is added to the 32-bit base address. The offset value can be either a 16-bit or 32-bit number.

7.5.1 Segment Registers

Every segment register has a "visible" part and an "invisible" part, as shown in Figure 7.13. When we talk about segment registers, we are referring to the 16-bit visible part. The visible part is referred to as the segment selector. There are direct instructions to load the segment selector. These instructions include `mov`, `pop`, `lds`, `les`, `lss`, `lgs`, and `lfs`. These instructions are discussed in later chapters and in Appendix I. The invisible part of the segment registers is automatically loaded by the processor from a descriptor table (described next).

As shown in Figure 7.12, the segment selector provides three pieces of information:

- *Index:* The index selects a segment descriptor from one of two descriptor tables: a local descriptor table or a global descriptor table. Since the index is a 13-bit value, it can select one of $2^{13} = 8192$ descriptors from the selected descriptor table. Since each descriptor, shown in Figure 7.14, is 8 bytes long, the processor multiplies the index by 8 and adds the result to the base address of the selected descriptor table.

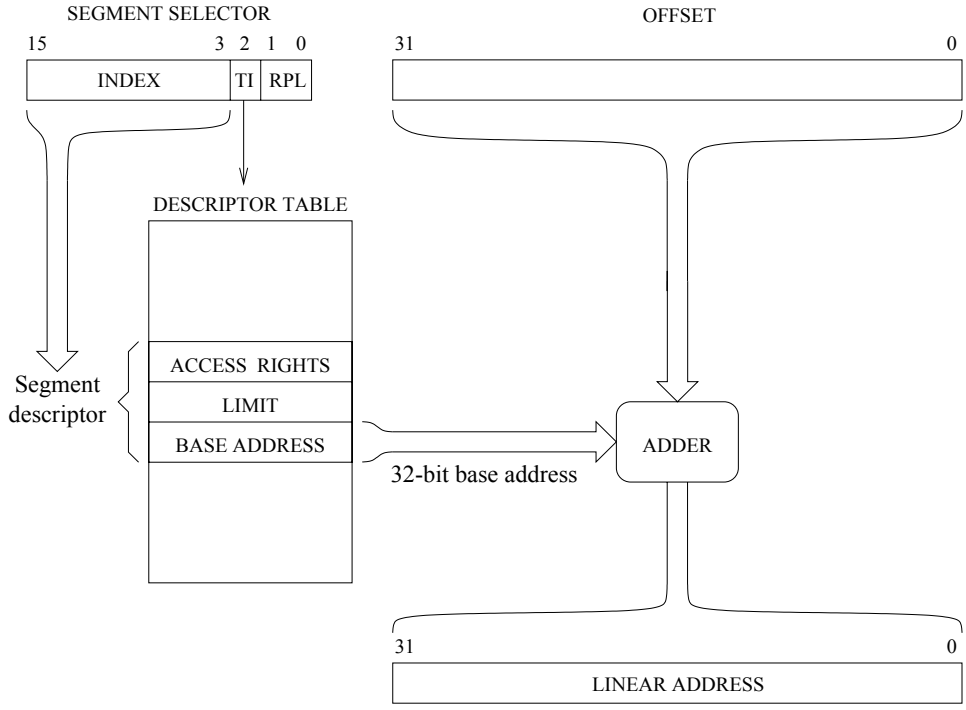


Figure 7.12 Protected mode address translation.

- *Table Indicator (TI):* This bit indicates whether the local or global descriptor table should be used.

  0 = Global descriptor table,
  1 = Local descriptor table.

- *Requester Privilege Level (RPL):* This field identifies the privilege level to provide protected access to data: the smaller the RPL value, the higher the privilege level.

7.5.2 Segment Descriptors

A segment descriptor provides the attributes of a segment. These attributes include its 32-bit base address, 20-bit segment size, as well as control and status information, as shown in Figure 7.14. Here we provide a brief description of some of the fields shown in this figure.

- *Base Address:* This 32-bit address specifies the starting address of a segment in the 4 GB physical address space. This 32-bit value is added to the offset value to get the linear address (see Figure 7.12).

Visible part	Invisible part	
Segment selector	Segment base address, size, access rights, etc.	CS
Segment selector	Segment base address, size, access rights, etc.	SS
Segment selector	Segment base address, size, access rights, etc.	DS
Segment selector	Segment base address, size, access rights, etc.	ES
Segment selector	Segment base address, size, access rights, etc.	FS
Segment selector	Segment base address, size, access rights, etc.	GS

Figure 7.13 Visible and invisible parts of segment registers.

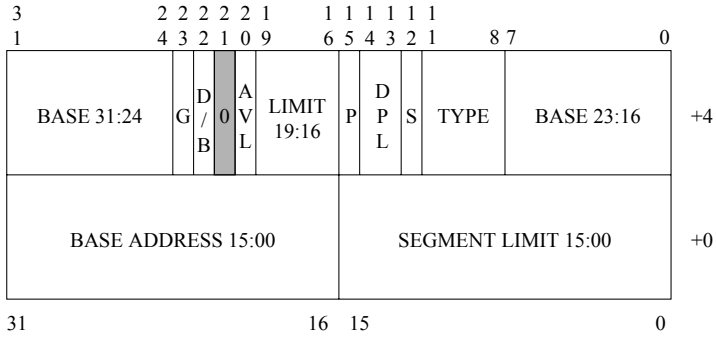


Figure 7.14 A segment descriptor.

- *Granularity (G):* This bit indicates whether the segment size value, described next, should be interpreted in units of bytes or 4 KB. If the granularity bit is zero, segment size is interpreted in bytes; otherwise, in units of 4 KB.
- *Segment Limit:* This is a 20-bit number that specifies the size of the segment. Depending on the granularity bit, two interpretations are possible:
  1. If the granularity bit is zero, segment size can range from 1 byte to 1 MB (i.e., 2^{20} bytes), in increments of 1 byte.
  2. If the granularity bit is 1, segment size can range from 4 KB to 4 GB, in increments of 4 KB.
- *D/B Bit:* In a code segment, this bit is called the D bit and specifies the default size for operands and offsets. If the D bit is 0, default operands and offsets are assumed to be 16 bits; for 32-bit operands and offsets, the D bit must be 1.

  In a data segment, this bit is called the B bit and controls the size of the stack and stack pointer. If the B bit is 0, stack operations use the SP register and the upper bound for the

stack is FFFFH. If the B bit is 1, the ESP register is used for the stack operations with a stack upper bound of FFFFFFFFH.

Typically, this bit is cleared for the real mode operation and set for the protected mode operation. Section 7.5.5 describes how 16- and 32-bit operands and addresses can be mixed in a given mode of operation.

- *S Bit:* This bit identifies whether the segment is a system segment or an application segment. If the bit is 0, the segment is identified as a system segment; otherwise, as an application (code or data) segment.
- *Descriptor Privilege Level (DPL):* This field defines the privilege level of the segment. It is useful in controlling access to the segment using the protection mechanisms of the Pentium processor.
- *Type:* This field identifies the type of segments. The actual interpretation of this field depends on whether the segment is a system or application segment. For application segments, the type depends on whether the segment is a code or data segment. For a data segment, type can identify it as a read-only, read-write, and so on. For a code segment, type identifies it as an execute-only, execute/read-only, and so on.
- *P bit:* This bit indicates whether the segment is present. If this bit is 0, the processor generates a segment-not-present exception when a selector for the descriptor is loaded into a segment register.

7.5.3 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors shown in Figure 7.14. There are three types of descriptor tables:

- The global descriptor table (GDT);
- Local descriptor tables (LDT);
- The interrupt descriptor table (IDT).

All three descriptor tables are variable in size from 8 bytes to 64 KB. The interrupt descriptor table is used in interrupt processing and is discussed in Chapter 20. Both LDT and GDT can contain up to $2^{13} = 8192$ 8-bit descriptors. As shown in Figure 7.12, the upper 13 bits of a segment selector are used as an index into the selected descriptor table. Each table has an associated register that holds the 32-bit linear base address and a 16-bit size of the table. LDTR and GDTR registers are used for this purpose. The LDTR and GDTR can be loaded using `lldt` and `lgdt` instructions. Similarly, the values of LDTR and GDTR registers can be stored by `sldt` and `sgdt` instructions. These instructions are typically used by the operating system.

The global descriptor table contains descriptors that are available to all tasks within the system. There is only one GDT in the system. Typically, the GDT contains code and data used by the operating system. The local descriptor table contains descriptors for a given program. There can be several LDTs, each of which may contain descriptors for code, data, stack, and so on. A program cannot access a segment unless there is a descriptor for the segment in either the current LDT or GDT.

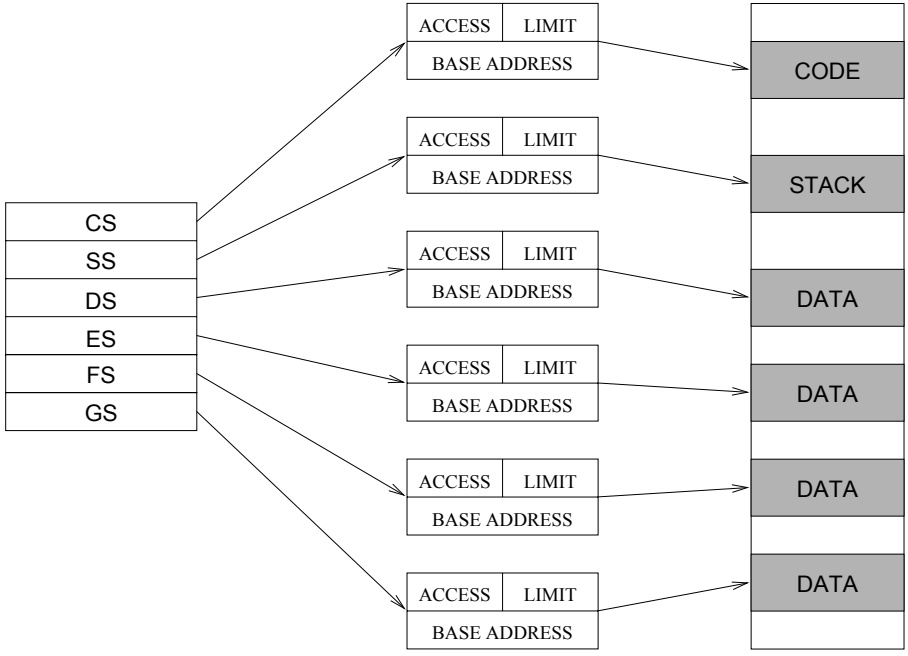


Figure 7.15 Segments in a multisegment model.

7.5.4 Segmentation Models

Remember that the 8086 segments are limited to 64 KB. However, in the Pentium, it is possible to span a segment over the entire physical address space. As a result, we can effectively make the segmentation invisible by mapping all segment base addresses to zero and setting the size to 4 GB. Such a model is called a *flat model* and is used in programming environments such as UNIX.

Another model that uses the capabilities of segmentation to the full extent is the *multisegment model*. Figure 7.15 shows an example mapping of six segments. A program, in fact, can have more than just six segments. In this case, the segment descriptor table associated with the program will have the descriptors loaded for all the segments defined by the program. However, at any time, only six of these segments can be active. Active segments are those that have their segment selectors loaded into the six segment registers. A segment that is not active can be made active by loading its selector into one of the segment registers, and the processor automatically loads the associated descriptor (i.e., the "invisible part" shown in Figure 7.13). The Pentium generates a general-protection exception if an attempt is made to access memory beyond the segment limit.

7.5.5 Mixed-Mode Operation

Our previous discussion of real and protected modes of operation suggests that we can use either 16-bit or 32-bit operands and addresses. The D/B bit indicates the default size. The question is: Is it possible to mix these two? For instance, can we use 32-bit registers in the 16-bit mode of operation? The answer is yes! The Pentium provides two size override prefixes—one for the operands and the other for the addresses—to facilitate such mixed mode programming. Details on these prefixes are provided in Chapter 11.

7.5.6 Which Segment Register to Use

This discussion applies to both real and protected modes of operation. In generating a physical memory address, the Pentium uses different segment registers depending on the purpose of the memory reference. Similarly, the offset part of the logical address comes from a variety of sources.

Instruction Fetch: When the memory access is to read an instruction, the CS register provides the segment base address. The offset part is supplied either by the IP or EIP register, depending on whether we are using 16-bit or 32-bit addresses. Thus, CS:(E)IP points to the next instruction to be fetched from the code segment.

Stack Operations: Whenever the processor is accessing the memory to perform a stack operation such as `push` or `pop`, the SS register is used for the segment base address, and the offset value comes from either the SP register (for 16-bit addresses) or the ESP register (for 32-bit addresses). For other operations on the stack, the BP or EBP register supplies the offset value. A lot more is said about the stack in Chapter 10.

Accessing Data: If the purpose of accessing memory is to read or write data, the DS register is the default choice for providing the data segment base address. The offset value comes from a variety of sources depending on the addressing mode used. Addressing modes are discussed in Chapter 11.

7.6 Summary

We described the architecture of the Pentium processor. The Pentium can address up to 4 GB of memory. It provides real and protected mode memory architectures. In the real mode, the Pentium supports 16-bit addresses and the memory architecture of the 8086 processor.

The protected mode is the native mode of the Pentium processor. In this mode, the Pentium supports both paging and segmentation. Paging is useful in implementing virtual memory and is not considered here. We discussed the segmented memory architecture in detail, as these details are necessary to program in the assembly language.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Address translation
- Effective address
- Flat segmentation model
- Instruction fetch
- Instruction pointer
- Linear address
- Logical address
- Memory architecture
- Mixed mode operation
- Multisegment segmentation model
- Override prefix
- Paging
- Pentium alignment check flag
- Pentium control registers
- Pentium data registers
- Pentium flags register
- Pentium index registers
- Pentium interrupt flag
- Pentium pointer registers
- Pentium registers
- Pentium trap flag
- Physical address
- Protected mode architecture
- Real mode architecture
- Segment descriptor
- Segment descriptor tables
- Segment registers
- Segmentation
- Segmentation models
- Segmented memory organization

7.7 Exercises

7–1 What is the purpose of providing various registers in a CPU?

7–2 What are the three address spaces supported by the Pentium?

7–3 What is a segment? Why does the Pentium support segmented memory architecture?

7–4 Why is segment size limited to 64 KB in the real mode?

7–5 In the real mode, segments cannot be placed anywhere in memory. Explain the reason for this restriction.

7–6 In the real mode, can a segment begin at the following physical addresses?

(a) 1235AH (b) 53535H
(c) 21700H (d) ABCD0H

7–7 What is the maximum size of a segment in the protected mode?

7–8 We stated that the Pentium can access up to six segments at a time. What is the hardware reason for this limitation?

7–9 In the protected mode, segment size granularity can be either 1 byte or 4 KB. Explain the hardware reason for this restriction.

7–10 What is the purpose of the TI field in the segment descriptor?

7–11 We looked at two descriptor tables: GDT and LDT. What is the maximum number of descriptors each table can have? Explain the hardware reason for this restriction.

7–12 Describe the logical to physical address translation process in the real mode.

7–13 Describe the logical to linear address translation process in the protected mode.

7–14 Discuss the differences between the segmentation architectures supported in the real and protected modes.

7–15 If a processor has 16 address lines, what is the physical memory address space of this processor? Give the address of the first and last addressable memory locations in hex.

7–16 Convert the following logical addresses to physical addresses. All numbers are in hexadecimal. Assume the real address mode.

(a) 1A2B:019A (b) 3911:200

(c) 2591:10B5 (d) 1100:ABCD

Chapter 8

Pipelining and Vector Processing

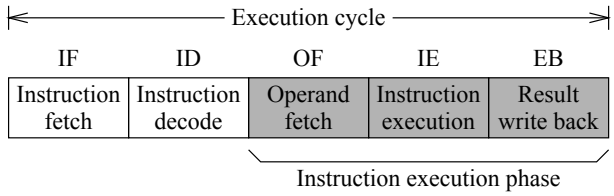
Objectives

- To introduce the principles of pipelining;
- To illustrate how resource conflicts are handled in pipelined systems;
- To give instruction pipeline details of some example processors;
- To describe vector processing basics;
- To present details on the Cray X-MP vector machine;
- To discuss performance of pipelined and vector processors.

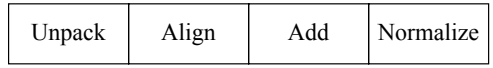
We briefly introduced the concept of pipelining in Chapter 1. Pipelining essentially allows overlapped execution of multiple operations. We begin this chapter with an introduction to the basic concepts of pipelining. Since pipelining is a highly coordinated activity, resource conflicts in the pipeline can seriously degrade its performance by causing pipeline stalls. We give some example scenarios that can potentially cause such stalls.

Several types of hazards cause pipeline stalls. We discuss three types: resource hazards, data hazards, and control hazards. When two or more instructions in the pipeline require the same resource, a resource hazard occurs. In Section 8.2, we describe some solutions for resource hazards. Data dependencies among the instructions lead to the data hazards. Section 8.3 gives some examples and possible solutions to minimize the impact of data dependencies on the pipeline performance.

Pipelining performs best when the execution is sequential. Pipeline performance deteriorates if the control flow is altered. Thus, it becomes important to devise ways to handle branches. Section 8.4 discusses several ways to handle such control hazards. The next section presents



(a) Instruction pipeline stages



(b) Floating-point add pipeline stages

Figure 8.1 For pipelining, we have to divide the work into several smaller, preferably equal, parts.

three performance enhancements including superscalar, superpipelining, and VLIW processors. Instruction pipeline implementations of the Pentium, PowerPC, SPARC, and MIPS processors are given in Section 8.6.

Section 8.7 gives details on vector processors. They use special vector registers. These registers can hold small arrays of floating-point numbers. Vector processors use pipelining in every aspect. They use pipelined transfer of data from memory to vector registers and overlapped execution of various integer and floating-point operations. These details are presented in Section 8.7. Performance of pipelined and vector processors is briefly discussed in Section 8.8. The last section gives a summary of the chapter.

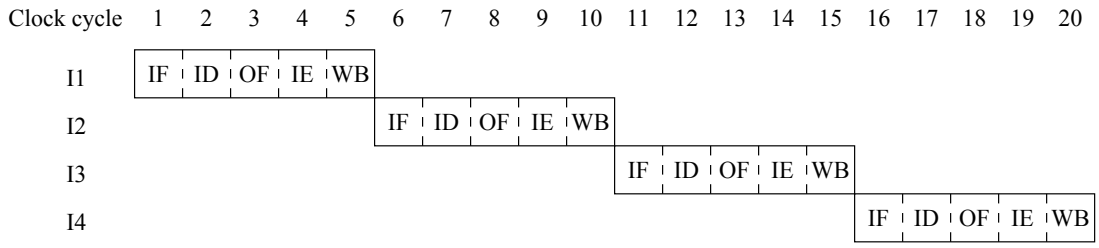
8.1 Basic Concepts

We briefly described the pipeline concept in Chapter 1. Pipelining allows overlapped execution to improve throughput. The pipeline concept can be applied to various functions of a computer system. We have introduced instruction pipelining in Chapter 1 through an example (see Section 1.5.1 on page 18). Before proceeding further, it is a good idea to refresh your memory by reviewing the material presented in that section.

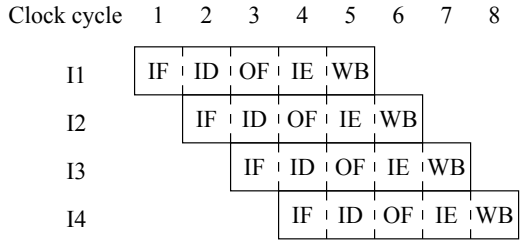
The key idea behind pipelining is to divide the work into smaller pieces and use assembly line processing to complete the work. In the instruction execution pipeline shown on page 17, we divided the execution cycle into five steps. For easy reference, we have reproduced these five steps in Figure 8.1*a*. In pipeline terminology, each step is called a *stage* because we will eventually have a dedicated piece of hardware to perform each step.

Pipelining can also be applied to arithmetic operations. As an example, we show a floating-point add pipeline in Figure 8.1*b*. The floating-point add unit has several stages:

1. *Unpack:* The unpack stage partitions the floating-point numbers into the three fields discussed in Section A.5.3: the sign field, exponent field, and mantissa field. Any special cases such as not-a-number (NaN), zero, and infinities are detected during this stage.



(a) Serial execution



(b) Pipelined execution

Figure 8.2 Pipelining overlaps execution among the stages. This improves instruction execution rate, even though each instruction takes the same amount of time.

2. *Align:* This stage aligns the binary points of the two mantissas by right-shifting the mantissa with the smaller exponent.
3. *Add:* This stage adds the two aligned mantissas.
4. *Normalize:* This stage packs the three fields of the result after normalization and rounding into the IEEE-754 floating-point format. Any output exceptions are detected during this stage.

Pipelining works best if we divide the work into equal parts so that each stage takes the same amount of time. Each stage performs its part by taking input from the previous stage. Figure 8.2 summarizes our discussion from Section 1.5.1.

Pipelining substantially reduces the execution time by overlapping execution of several instructions. In the example shown in this figure, serial execution takes 20 clock cycles to execute four instructions I1 through I4. On the other hand, pipelined execution takes only 8 clock cycles. However, pipelining requires hardware support. For the five-stage instruction pipeline, we need four buffers as shown in Figure 8.3. Each of these buffers holds only one value, the output produced by the previous stage. This is possible because the pipeline follows the just-in-time principle. In some processors, the ID stage uses the IF stage buffers by placing the decoded instruction back in the B1 buffer. As we show later, just-in-time arrival of input causes problems because any delay in one stage can seriously affect the entire pipeline flow.

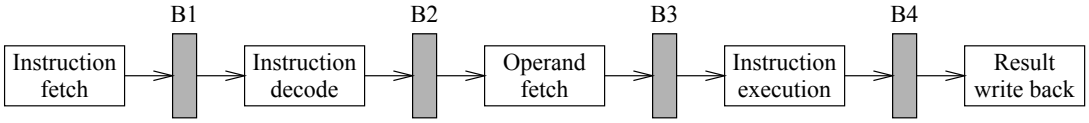


Figure 8.3 Pipelines typically do not require substantial buffering as they use the just-in-time concept.

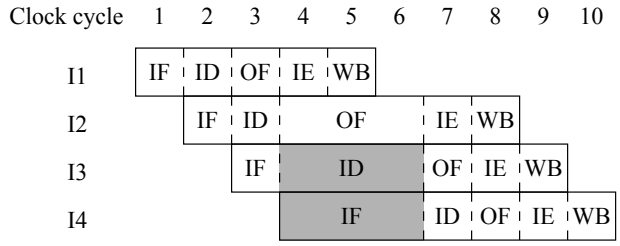


Figure 8.4 A delay in any stage can cause pipeline stalls.

What we have shown in Figure 8.2 is an ideal scenario. Several factors can adversely affect the performance of a pipeline. For starters, it is not always possible to divide the work into equal parts. In that case, the slowest stage determines the flow rate in the entire pipeline.

What are some of the reasons for not having equal work for all stages? Sometimes this is due to a complex step that cannot be subdivided conveniently. In some instances, it may be due to the operation that takes a variable amount of time to execute. For example, consider the operand fetch (OF) step in the instruction pipeline. The time required to execute this step depends on where the operands are located: in registers, cache memory, or main memory. If the operands are in registers rather than in memory, it takes less time. In some cases, the complexity of the operation depends on the type of operation performed by the instruction. For example, an integer addition operation may take only one cycle but a multiplication operation may take several cycles.

To illustrate the impact of the variable stage time on pipelined execution, let's assume that the operand fetch of the I2 instruction takes more time: three clock cycles rather than one. Figure 8.4 shows how the increased execution time for I2's OF stage causes the pipeline to *stall* for two clock cycles. As you can see from this figure, the stalled pipeline reduces the overall throughput. One of the goals in designing a pipeline is to minimize pipeline stalls.

Pipeline stalls can be caused by several factors. These are called *hazards*. There are three types of hazards: *resource*, *data*, and *control hazards*. Resource hazards result when two or more instructions in the pipeline want to use the same resource. Such resource conflicts can result in serialized execution, reducing the scope for overlapped execution. Resource hazards, sometimes referred to as *structural hazards*, are discussed in the next section.

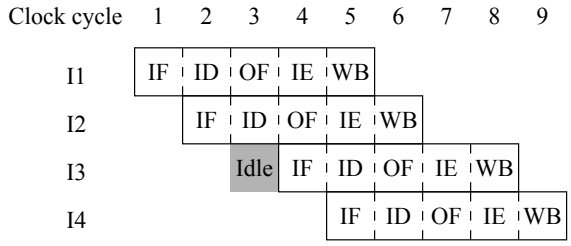


Figure 8.5 This scenario describes a memory conflict caused by the instruction fetch of I3 and memory-resident operand fetch of I1.

Data hazards are caused by data dependencies among the instructions in the pipeline. As a simple example, suppose that the result produced by instruction I1 is needed as an input to instruction I2. We have to stall the pipeline until I1 has written the result so that I2 reads the correct input. If the pipeline is not designed properly, data hazards can produce wrong results by using incorrect operands. Therefore, we have to worry about the correctness first. We can use a technique called *interlocking* to ensure that only correct operands are used. There is another technique called *forwarding* that actually improves the performance by reducing the amount of time the pipeline is stalled waiting for the result of a previous instruction. We discuss data hazards in Section 8.3.

Thus far, we assumed sequential flow control. What happens if the flow control is altered, say, due to a branch instruction? If the branch is not taken, we can proceed with the instructions in the pipeline. But, if the branch is taken, we have to throw away all the instructions that are in the pipeline and fill the pipeline with instructions at the branch target. These hazards are referred to as the control hazards, which are caused by control dependencies. We describe some possible solutions used to minimize the impact of control dependencies in Section 8.4.

8.2 Handling Resource Conflicts

To see why resource conflicts cause problems, consider the pipelined execution shown in Figure 8.2*b*. Let's ignore the cache and assume that both data and instructions are stored in a single-ported main memory. That is, memory supports only one read or write operation at a time. Thus, we can read from memory either an instruction or an operand (but not both at the same time). If an operand of I1 is in memory, the instruction fetch of I3 and operand fetch of I1 cause a memory conflict. The instruction fetch unit idles for one clock cycle, another example of a pipeline stall (see Figure 8.5).

You can also imagine resource conflicts if we have only one ALU and multiple instructions want to use it. The solution for the resource hazards is to minimize the conflicts by increasing the available resources. For example, if we have separate access paths to get instructions and data, we could have avoided the conflict shown in Figure 8.5: the Harvard architecture uses separate buses for data and instruction memories. Also, providing separate instruction and data

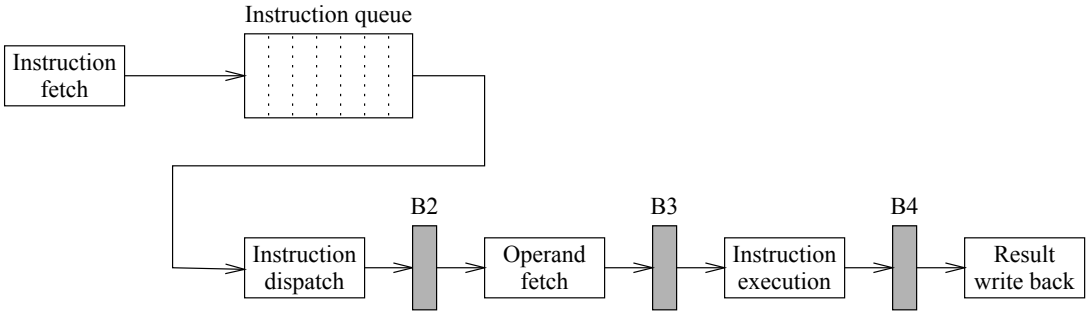


Figure 8.6 The impact of resource conflicts can be minimized by using a queue to buffer the instructions.

caches helps reduce the resource conflict. These topics are discussed in Chapter 17. We also later show that multiple resources are used to reduce resource conflicts.

Prefetching is another technique we can use to handle resource conflicts. As we have seen before, pipelining typically uses the just-in-time mechanism so that only a simple buffer is needed between the stages (see Figure 8.3). We can minimize the performance impact if we relax this constraint by allowing a queue instead of a single buffer. We illustrate this technique for our pipeline example. Suppose we replace buffer B1 by an *instruction queue*, as shown in Figure 8.6. The instruction fetch unit can prefetch instructions and place them in the instruction queue. The decoding unit will have ample instructions even if the instruction fetch is occasionally delayed because of a cache miss or resource conflict.

8.3 Data Hazards

Data dependencies can deteriorate performance of a pipeline by causing stalls. As a simple example, consider the following instruction sequence:

```
I1:    add     R2,R3,R4    /* R2 = R3 + R4 */
I2:    sub     R5,R6,R2    /* R5 = R6 – R2 */
```

The sum computed by the `add` instruction is used as input to the `sub` instruction. This data dependency between I1 and I2 causes the pipeline to stall, as shown in Figure 8.7.

When two instructions access registers or memory locations in a conflicting mode, data dependency exists. A conflicting access is one in which one or both instructions alter the data. Depending on the type of conflicting access, we can define the following dependencies:

- *Read-After-Write (RAW):* This dependency exists between two instructions if one instruction writes into a register or a memory location that is later read by the other instruction.
- *Write-After-Read (WAR):* This dependency exists between two instructions if one instruction reads from a register or a memory location that is later written by the other instruction.

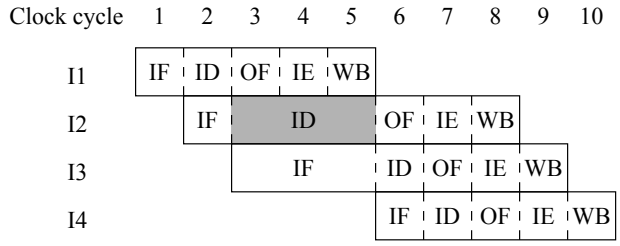


Figure 8.7 Data dependency between instructions I1 and I2 causes the pipeline to stall for two clock cycles.

- *Write-After-Write (WAW):* This dependency exists between two instructions if one instruction writes into a register or a memory location that is later written by the other instruction.

There is no conflict in allowing read-after-read (RAR) access. Data dependencies have two implications:

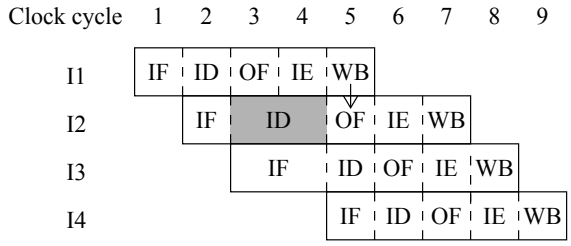
- *Correctness Issue:* We have to detect this dependency and stall the `sub` instruction from executing until `add` has written the sum into the R2 register. Otherwise, we end up using an incorrect R2 value in `sub`.
- *Efficiency Issue:* Pipeline stall can be long if we don't come up with a trick.

There are two techniques used to handle data dependencies: *register interlocking* and *register forwarding*. We first discuss the register forwarding method as most processors use this technique to reduce pipeline stalls. Later we show how the Pentium handles these three types of dependencies.

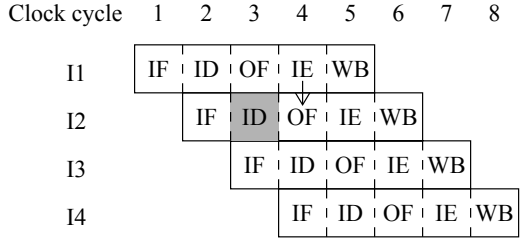
8.3.1 Register Forwarding

This technique, also called *bypassing*, works if the two instructions involved in the dependency are in the pipeline. The basic idea is to provide the output result as soon as it is available in the datapath. This technique is demonstrated in Figure 8.8. For example, if we provide the output of I1 to I2 as we write into the destination register of I1, we will reduce the number of stall cycles by one (see Figure 8.8*a*). We can do even better if we feed the output from the IE stage as shown in Figure 8.8*b*. In this case, we completely eliminate the pipeline stalls.

How do we implement this forwarding in hardware? To understand the implementation let's look at the single-bus datapath shown in Figure 6.7 (page 220). It is redrawn in Figure 8.9 to give details that are relevant to our discussion here. The `add` instruction execution involves moving the R3 contents to the A register and placing the R4 contents on the A bus. Once this is done, the ALU would produce the sum after the propagation delay through it. However, this result will not be available for a couple of more cycles, as it has to be latched into the C register and then into the final output register R2. If we provide feedback from the C register ("Forward



(a) Forward scheme 1



(b) Forward scheme 2

Figure 8.8 We can minimize the stalls if we provide the output of I1 to I2 as soon as possible.

1" path in Figure 8.9), we can save one clock cycle as in Figure 8.8*a*. We can further improve as in Figure 8.8*b* by providing a path to connect the output of the ALU to its input (the "Forward 2" path).

Register forwarding requires changes to the underlying hardware as shown in Figure 8.9. In this figure, the output of the ALU is fed back to its two inputs. Of course, we need a multiplexer and associated control circuit to properly route the ALU output. These details are not shown in this figure.

8.3.2 Register Interlocking

This is a general technique to solve the correctness problem associated with data dependencies. In this method, a bit is associated with each register to specify whether the contents are correct. If the bit is 0, the contents of the register can be used. Instructions should not read contents of a register when this interlocking bit is 1, as the register is locked by another instruction. Figure 8.10 shows how the register interlocking works for the `add/sub` example on page 278. I1 locks the R2 register for clock cycles 3 to 5 so that I2 cannot proceed reading an incorrect R2 value. Clearly, register forwarding is more efficient than the interlocking method.

The Intel Itanium processor associates a bit (called NaT—not-a-thing) similar to the interlocking bit with the general-purpose registers. The Itanium uses this to support speculative execution. We give more details in Chapter 14.

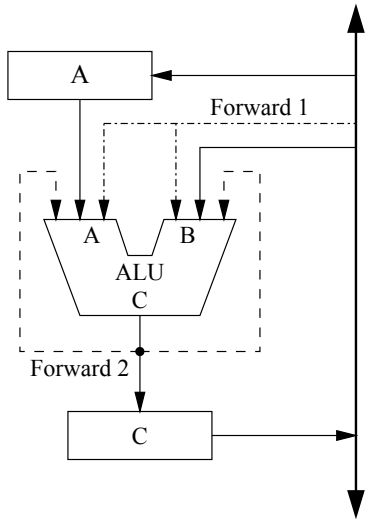


Figure 8.9 The forwarding technique requires support in hardware.

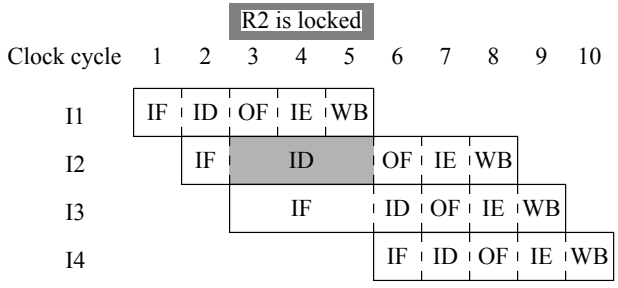
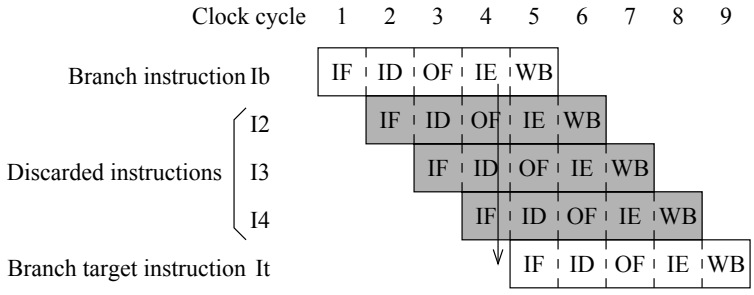


Figure 8.10 Register interlocking uses a lock bit to indicate whether the result can be used.

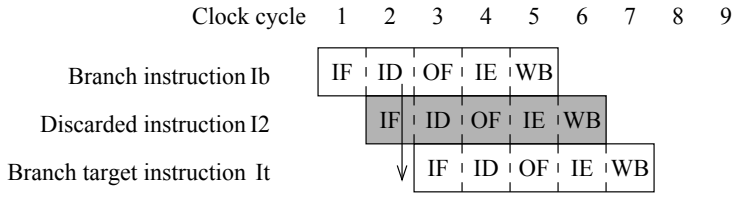
Register forwarding can be used only when the required values are already in the pipeline. Interlocking, however, can handle data dependencies of a general nature. For example, in the code

```
load    R3,count    ;R3 = count
add     R1,R2,R3    ;R1 = R2 + R3
```

the `add` instruction should not use the contents of R3 until the `load` has placed the `count` value in R3. Register forwarding is not useful for this scenario.



(a) Branch decision is known during the IE stage



(b) Branch decision is known during the ID stage

Figure 8.11 Impact of a branch instruction on the pipeline.

8.4 Handling Branches

Flow altering instructions such as branch require special handling in pipelined processors. Figure 8.11*a* shows the impact of a branch instruction on our pipeline. Here we are assuming that instruction Ib is a branch instruction; if the branch is taken, it transfers control to instruction It. If the branch is not taken, the instructions in the pipeline are useful. However, for a taken branch, we have to discard all the instructions that are in the pipeline at various stages. In our example, we have to discard instructions I2, I3, and I4. We start fetching instructions at the target address. This causes our pipeline to do wasteful work for three clock cycles. This is called the *branch penalty*.

How can we reduce this branch penalty? If you look at Figure 8.11 closely, you will notice that we wait until the execution (IE) stage before initiating the instruction fetch at the target address. We can reduce the delay if we can determine this earlier. For example, if we find whether the branch is taken along with the target address information during the decode (ID) stage, we would just pay a penalty of one cycle, as shown in Figure 8.11*b*.

In our example, only one instruction (I2) needs to be discarded. But can we get the necessary information at the decode stage? For most branch instructions, the target address is given as part of the instruction. So computation of the target address is relatively straightforward. But it may not be that easy to determine whether the branch is taken during the decode stage. For example, we may have to fetch the operands and compare their values to determine whether the branch is taken. This means we have to wait until the IE stage. We can use branch prediction strategies discussed in Section 8.4.2 to make an educated guess.

8.4.1 Delayed Branch Execution

We have shown in Figure 8.11*b* that we can reduce the branch penalty to one cycle. Delayed branch execution effectively reduces the branch penalty further. The idea is based on the observation that we always fetch the instruction following the branch before we know whether the branch is taken. Why not execute this instruction instead of throwing it away? This implies that we have to place a useful instruction in this instruction slot. This instruction slot is called the *delay slot*. In other words, the branching is delayed until after the instruction in the delay slot is executed. Some processors like the SPARC and MIPS use delayed execution for both branching and procedure calls.

When we apply this technique, we need to modify our program to put a useful instruction in the delay slot. We illustrate this by using an example. Consider the following code segment:

```
        add     R2,R3,R4
        branch  target
        sub     R5,R6,R7
        ...       . . .
target:
        mult    R8,R9,R10
        ...       . . .
```

If the `branch` is delayed, we can reorder the instructions so that the branch instruction is moved ahead by one instruction, as shown below:

```
        branch  target
        add     R2,R3,R4   /* Branch delay slot */
        sub     R5,R6,R7
        ...       . . .
target:
        mult    R8,R9,R10
        ...       . . .
```

Programmers do not have to worry about moving instructions into the delay slots. This job is done by the compilers and assemblers. When no useful instruction can be moved into the delay slot, a no operation (NOP) is placed.

We should also note that when the branch is not taken, we do not want to execute the delay slot instruction (i.e., we want to *nullify* the delay slot instruction). Some processors like the SPARC provide this nullification option. We give more details and examples of delayed execution in Appendix H, which describes the SPARC processor.

8.4.2 Branch Prediction

Branch prediction is traditionally used to handle the branch problem. We discuss three branch prediction strategies: fixed, static, and dynamic.

Table 8.1 Static branch prediction accuracy

Instruction type	Instruction distribution (%)	Prediction: Branch taken?	Correct prediction (%)
Unconditional branch	$70 \times 0.4 = 28$	Yes	28
Conditional branch	$70 \times 0.6 = 42$	No	$42 \times 0.6 = 25.2$
Loop	10	Yes	$10 \times 0.9 = 9$
Call/return	20	Yes	20
Overall prediction accuracy = 82.2%			

Fixed Branch Prediction

In this strategy, prediction is fixed. These strategies are simple to implement and assume that the branch is either never taken or always taken. The Motorola 68020 and VAX 11/780 use the branch-never-taken approach. The advantage of the never-taken strategy is that the processor can continue to fetch instructions sequentially to fill the pipeline. This involves minimum penalty in case the prediction is wrong. If, on the other hand, we use the always-taken approach, the processor would prefetch the instruction at the branch target address. In a paged environment, this may lead to a page fault, and a special mechanism is needed to take care of this situation. Furthermore, if the prediction were wrong, we would have done lot of unnecessary work.

The branch-never-taken approach, however, is not proper for a loop structure. If a loop iterates 200 times, the branch is taken 199 out of 200 times. For loops, the always-taken approach is better. Similarly, the always-taken approach is preferred for procedure calls and returns.

Static Branch Prediction

From our discussion, it is obvious that, rather than following a fixed strategy, we can improve performance by using a strategy that is dependent on the branch type. This is what the static strategy does. It uses instruction opcode to predict whether the branch is taken. To show why this strategy gives high prediction accuracy, we present sample data for commercial environments. In such environments, of all the branch-type operations, the branches are about 70%, loops are 10%, and the rest are procedure calls/returns. Of the total branches, 40% are unconditional. If we use a never-taken guess for the conditional branch and always-taken for the rest of the branch-type operations, we get a prediction accuracy of about 82% as shown in Table 8.1.

The data in this table assume that conditional branches are not taken about 60% of the time. Thus, our prediction that a conditional branch is never taken is correct only 60% of the time. This gives us $42 \times 0.6 = 25.2\%$ as the prediction accuracy for conditional branches. Similarly, loops jump back with 90% probability. Since loops appear about 10% of the time, the prediction

Table 8.2 Impact of using the knowledge of past n branches on prediction accuracy

	Type of mix		
n	Compiler	Business	Scientific
0	64.1	64.4	70.4
1	91.9	95.2	86.6
2	93.3	96.5	90.8
3	93.7	96.6	91.0
4	94.5	96.8	91.8
5	94.7	97.0	92.0

is right 9% of the time. Surprisingly, even this simple static prediction strategy gives us about 82% accuracy!

Dynamic Branch Prediction

Dynamic strategy looks at the run-time history to make more accurate predictions. The basic idea is to take the past n branch executions of the branch type in question and use this information to predict the next one. Will this work in practice? How much additional benefit can we derive over the static approach? The empirical study by Lee and Smith [25] suggests that we can get significant improvement in prediction accuracy. A summary of their study is presented in Table 8.2. The algorithm they implemented is simple: The prediction for the next branch is the majority of the previous n branch executions. For example, for $n = 3$, if two or more times branches were taken in the past three branch executions, the prediction is that the branch will be taken.

The data in Table 8.2 suggest that looking at the past two branch executions will give us over 90% prediction accuracy for most mixes. Beyond that, we get only marginal improvement. This is good from the implementation point of view: we need just two bits to take the history of the past two branch executions. The basic idea is simple: keep the current prediction unless the past two predictions were wrong. Specifically, we do not want to change our prediction just because our last prediction was wrong. This policy can be expressed using the four-state finite state machine shown in Figure 8.12.

In this state diagram, the left bit represents the prediction and the right bit indicates the branch status (branch taken or not). If the left bit is zero, our prediction would be branch "not taken"; otherwise we predict that the branch will be taken. The right bit gives the actual result of the branch instruction. Thus, a 0 represents that the branch instruction did not jump ("not taken"); 1 indicates that the branch is taken. For example, state 00 represents that we predicted that the branch would not be taken (left zero bit) and the branch is indeed not taken (right zero

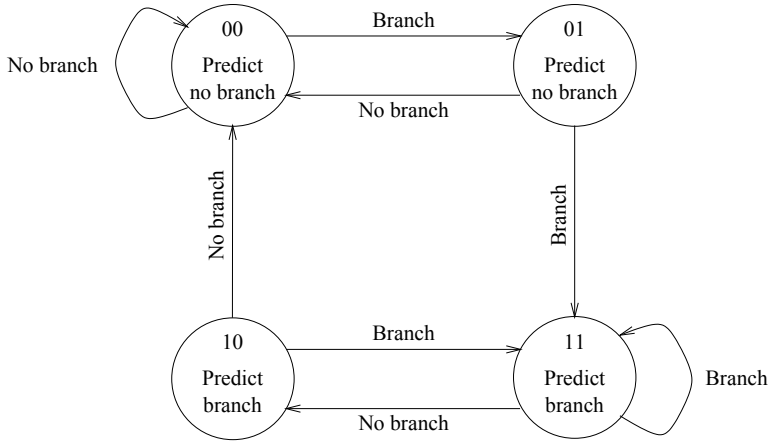


Figure 8.12 State diagram for branch prediction.

bit). Therefore, as long as the branch is not taken, we remain in state 00. If our prediction is wrong, we move to state 01. However, we will still predict "branch not taken" as we were wrong only once. If our prediction is right, we go back to state 00. If our prediction is wrong again (i.e., two times in a row), we change our prediction to "branch taken" and move to state 10. You can verify that it always takes two wrong predictions in a row to change our prediction.

Implementation of this strategy requires maintaining two bits for each branch instruction, as shown in Figure 8.13a. These two bits correspond to the two bits of the finite state machine in Figure 8.12. This works well for direct branch instructions, where the address of the target is specified as part of the instruction. However, in indirect branch instructions, the target is not known until instruction execution. Therefore, predicting whether the branch is taken is not particularly useful to fill the pipeline if we do not know the target address in advance. It is reasonable to assume that the branch instruction, if the branch is taken, jumps to the same target address as the last time. Thus, if we store the target address along with the branch instruction, we can use this target address to prefetch instructions to fill the pipeline. This scenario is shown in Figure 8.13b. In Section 8.6, we look at some processors that use the dynamic branch prediction strategy.

8.5 Performance Enhancements

We look at several techniques to improve performance of a pipelined system: (i) superscalar processors, (ii) superpipelined systems, and (iii) very long instruction word (VLIW) architectures. We start our discussion with the superscalar processors. Superpipelined systems improve the throughput by increasing the pipeline depth. VLIW architectures encode multiple operations into a long instruction word. The hardware can then schedule these operations on multiple functional units without any run-time analysis.

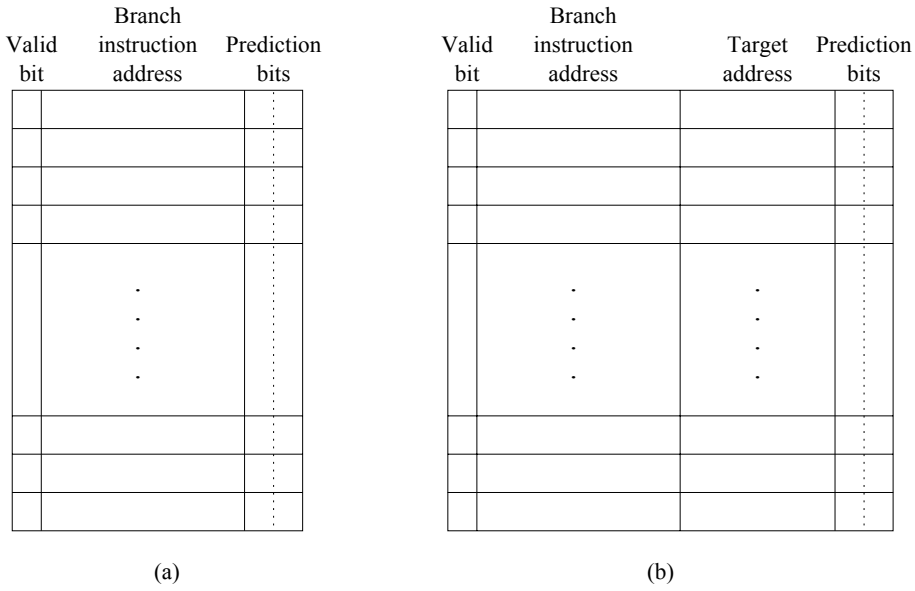


Figure 8.13 Implementation of dynamic branch prediction: (a) Using a 2-bit branch history; (b) Including the target address facilitates prefetching.

8.5.1 Superscalar Processors

Superscalar processors improve performance by replicating the pipeline hardware. One simple technique is to have multiple pipelines. Figure 8.14 shows a dual pipeline design, somewhat similar to that present in the Pentium. We discuss the Pentium instruction pipeline details in Section 8.6. The instruction fetch unit fetches two instructions each cycle and loads the two pipelines with one instruction each. Since these two pipelines are independent, instruction execution can proceed in parallel.

When we issue more than one instruction, we have to worry about the dependencies discussed before. Section 8.6 gives details on how the Pentium processor handles these data dependencies.

In our pipeline example, we assumed that all stages take the same amount of time. What if the instruction execution takes more time? In reality, this stage takes a variable amount of time. Although simple integer instructions can be executed in one cycle, complex instructions such as integer division and floating-point operations can take longer (often by several cycles). If we have only one execution unit at stage 4, execution of these complex instructions can bring the pipeline to a crawl! We can improve the situation by providing multiple execution units, linked to a single pipeline, as shown in Figure 8.15. In this figure, we are using four execution units: two integer units and two floating-point units. Such designs are referred to as *superscalar processors*. We discuss instruction pipeline details of four processors in Section 8.6.

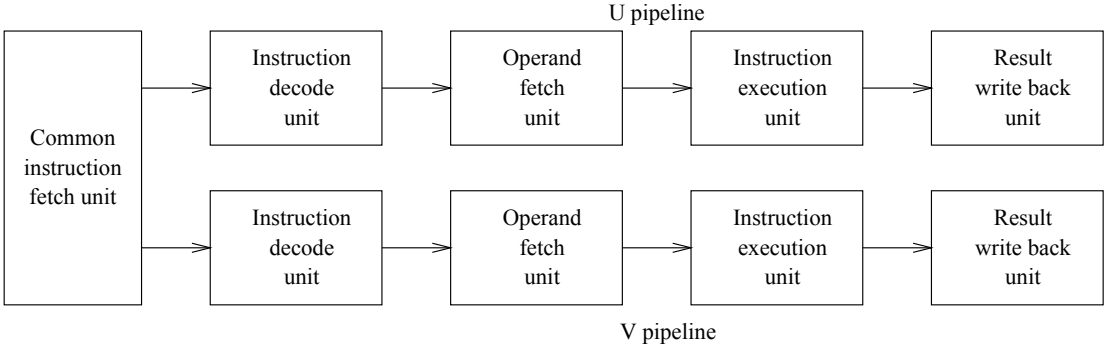


Figure 8.14 A dual pipeline based on the five stages used in Figure 8.1a.

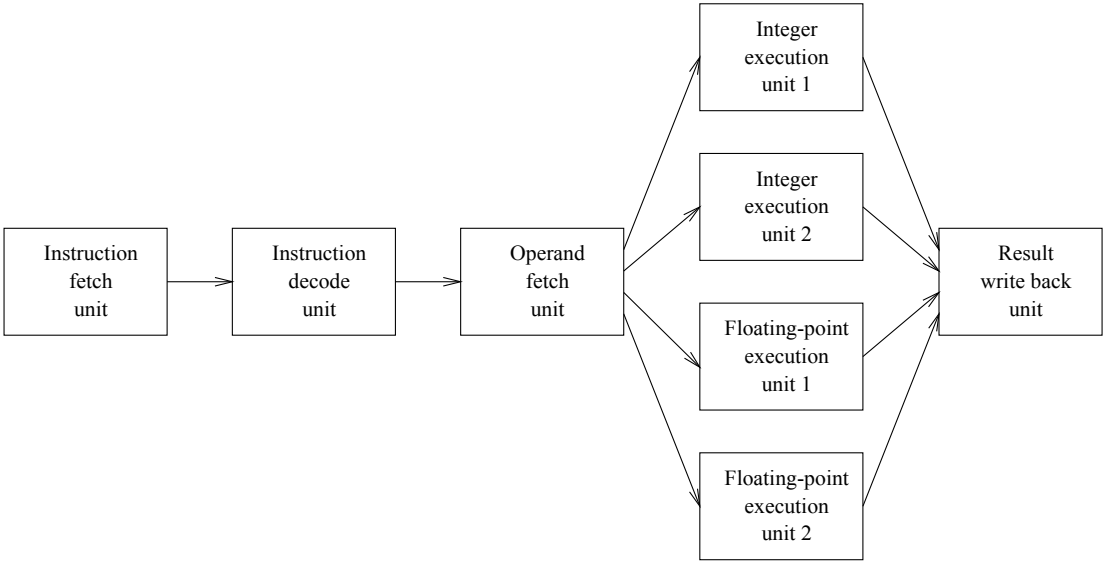
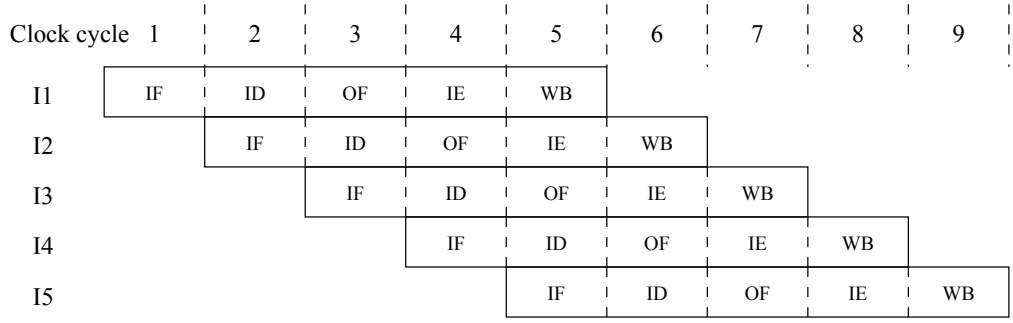


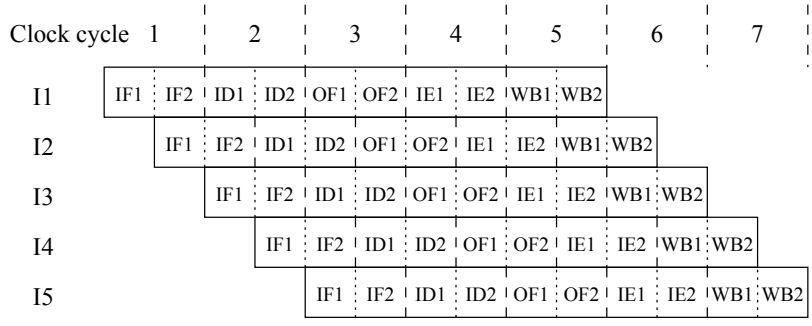
Figure 8.15 A superscalar pipeline with four functional units.

8.5.2 Superpipelined Processors

Superpipelined processors divide each processor cycle into two or more subcycles. A new instruction is fetched in each subcycle. The performance advantage of the superpipelined approach is shown in Figure 8.16. Part (a) of this figure is similar to that shown in Figure 8.2b. As you can see from Figure 8.16b, the five instructions complete two cycles earlier than in the pipelined version. After the pipeline is full, the superpipeline version executes two instructions per cycle (i.e., one instruction per subcycle).



(a) Pipelined execution



(b) Superpipelined execution

Figure 8.16 Superpipelined execution tends to achieve better performance by dividing each step into substeps.

As an example of a superpipelined processor, we present CPU pipeline details of the MIPS R4000 processor (see Table 8.3). It uses an eight-stage instruction pipeline; each stage takes half of the master clock cycle. Thus, execution of each instruction takes four master clock cycles.

The instruction fetch and memory access stages are split into two suboperations. The EX stage computes the operations. In load and store operations, the ALU computes the address during this stage. For the branch instructions, the ALU determines whether the branch is taken (and computes the target address) during the EX stage. The TC stage is used to check the tag field of the cache. The cache tag field is described in Chapter 17.

We have demonstrated that superpipelining increases the throughput of the pipeline. However, deeper pipelines increase the problems associated with data and control dependencies. We return to this topic in Section 8.8.1.

Table 8.3 Pipeline stages of the MIPS R4000 processor

IF1	Instruction fetch, first half
IF2	Instruction fetch, second half
RF	Decode instruction and fetch register operands
EX	Instruction execution
DF1	Data fetch (load/store), first half
DF2	Data fetch (load/store), second half
TC	Load/store check
WB	Write back

8.5.3 Very Long Instruction Word Architectures

One of the ways to improve performance of a processor is to increase the number of functional units. We have seen this principle in action with the superscalar processors. When we have multiple resources, scheduling instructions to keep these units busy is important to realize higher performance. There is no use in having a large number of functional units if we cannot keep them busy doing useful work. Thus instruction scheduling is very important. In most processors, instruction scheduling is done at run-time by looking at the instructions in the instruction queue. Instruction scheduling needs to take the available resources and instruction dependencies into account. For example, if we have one integer add unit and a single floating-point add unit, we can schedule one integer add instruction and another floating-point add instruction. Obviously, we cannot schedule two integer add instructions or two floating-point add instructions. In the code sequence

```
add     R1,R2,R3    ;R1 = R2+R3
sub     R5,R6,R7    ;R5 = R6-R7
and     R4,R1,R5    ;R4 = R1 AND R5
```

we cannot schedule these three instructions even if we have add, subtract, and logical AND functional units due to the dependency between the `and` and `add/sub` instructions. We can schedule `add` and `sub` in one instruction cycle and the `and` instruction can be scheduled in the next cycle.

Instruction scheduling is more complex than this case might suggest. For example, we can use *out-of-order* scheduling as in the following:

```
add     R1,R2,R3    ;R1 = R2+R3
sub     R5,R6,R7    ;R5 = R6-R7
and     R4,R1,R5    ;R4 = R1 AND R5
xor     R9,R9,R9    ;R9 = R9 XOR R9
```

We can schedule these four instructions in two cycles:

Cycle 1: `add`, `sub`, `xor`
Cycle 2: `and`

Even though we schedule the `xor` instruction ahead of the `and`, this out-of-order execution does not cause any semantic problems. Such out-of-order execution allows us to exploit instruction level parallelism (ILP) to improve performance. We further discuss ILP in Chapter 14.

Very long instruction word (VLIW) architectures move the job of instruction scheduling from run-time to compile-time. An advantage of such a scheme is that we can do more comprehensive offline analysis to determine the best scheduling, as we have complete knowledge of the program. Also notice that this shift from run-time to compile-time also means moving from hardware to software. This transfer of complexity from hardware to software leads to simpler, more efficient, and easier-to-design processors. At the same time, we get all the flexibility that we normally associate with a software-based approach to handle difficult tasks.

Each VLIW instruction consists of several primitive operations that can be executed in parallel. Each word in a VLIW processor may be tens of bytes wide. For example, the Multiflow TRACE system uses a 256-bit instruction word. It packs 7 different operations into each word. A more powerful model of the TRACE system uses 1024-bit instructions and packs as many as 28 operations.

Branches and memory access (load and store operations) invariably cause performance problems. These processors can minimize the branch prediction errors by executing all branch results and then selecting the result that corresponds to the taken branch.

Load/store latencies can be minimized by using what is known as *speculative loading*. Intel incorporated some of these concepts in their Itanium processor. The Itanium uses a 128-bit data bus. Thus, each memory read can bring 128 bits called instruction bundles. Each bundle consists of three 40-bit instructions. The remaining 8 bits are used to carry information on the three packed instructions. The compiler is responsible for packing instructions that do not have any conflicts into each bundle such that these instructions can be executed in parallel. There is no need to do run-time analysis to schedule these instructions. The Itanium also uses branch elimination and speculative loading techniques. These details are discussed in Chapter 14.

8.6 Example Implementations

In this section, we look at the instruction pipeline details of four processors: the Pentium, PowerPC, SPARC, and MIPS. Among these four processors, only the Pentium is the CISC processor; the other three are RISC processors.

8.6.1 Pentium

The Pentium uses a dual pipeline design, similar to the one shown in Figure 8.14, to achieve superscalar execution. Figure 8.17 shows the block diagram of the Pentium processor. It uses dual integer pipelines, called U- and V-pipes, and a floating-point pipeline. As shown in Figure 8.18, the integer pipeline has five stages. These five stages perform the following functions:

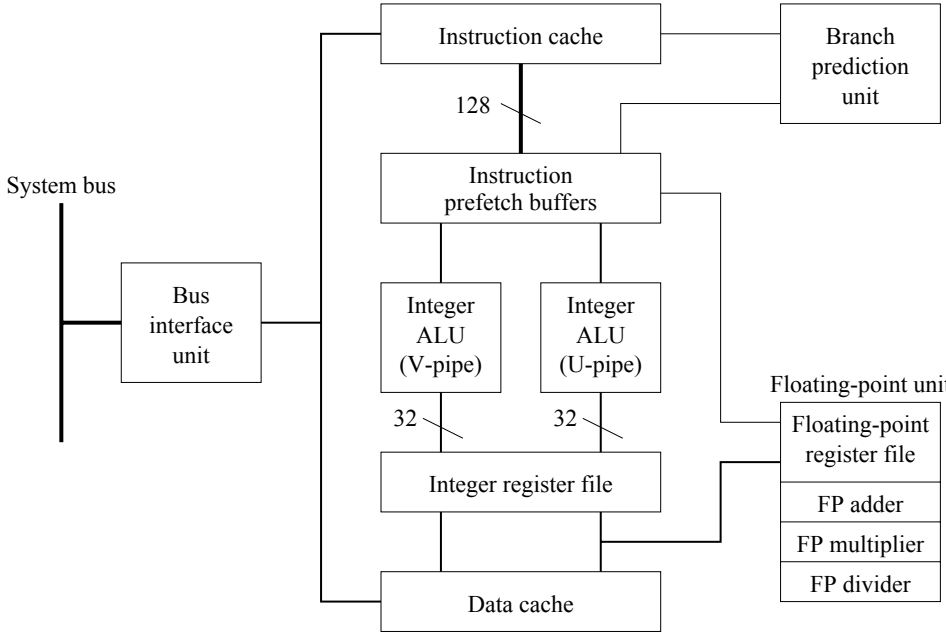


Figure 8.17 A block diagram of the Pentium processor (from [1]).

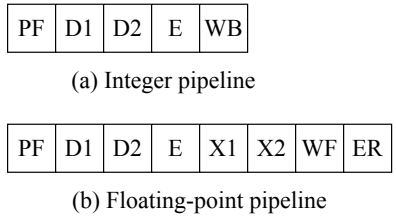


Figure 8.18 Integer and floating-point pipelines of the Pentium processor.

1. *Prefetch (PF):* This stage fetches instructions and stores them in the instruction buffer. Since the Pentium uses variable-length instructions, alignment of instructions to the cache line is taken care of by the instruction buffers.

2. *First Decode (D1):* This stage decodes the instruction and generates either a single control word (for simple operations) or a sequence of control words (for complex operations). Instructions that require only a single control word can be executed directly. These instructions are called "simple" instructions. Complex instructions require several control words. As we have seen in Chapter 6, these control word sequences are generated by a microprogrammed control unit.

3. *Second Decode (D2):* The control words generated in the D1 stage are decoded in this stage. This stage also generates the necessary operand addresses.
4. *Execute (E):* This stage either accesses the data cache to get operands or executes instructions in the ALU and other functional units. Which action is taken depends on the type of instruction. If the operands of an instruction are in registers, the D2 stage is executed and the operation is performed during the E stage and the result is written back to the register set. For a memory operand, D2 calculates the operand address and the E stage fetches the operand from the data cache. In the case of a cache hit, the data are available at the end of this stage. Another E stage is inserted to execute the operation, after which the result is written back. In the case of a cache miss, data need to be fetched from memory. Chapter 17 describes how cache misses are handled.
5. *Write Back (WB):* This stage essentially writes the result back into the register set or data cache.

The Pentium's floating-point pipeline has eight stages (see Figure 8.18). The first three stages are the same as in the integer pipeline. The next five stages are described below:

- *Operand Fetch (OF):* During this stage, the FPU accesses the data cache and the floating-point register file to fetch the necessary operands for the floating-point operation.
- *First Execute (X1):* During this step, the initial operation is performed. If the data are read from the data cache, they are written to the floating-point register file.
- *Second Execute (X2):* The X2 stage continues the floating-point operation initiated during the X1 stage.
- *Write Float (WF):* The FPU completes the floating-point operation and writes the result to the floating-point register file.
- *Error Reporting (ER):* This stage is used for error detection and reporting. Additional processing may be required to complete execution.

The dual pipeline allows execution of two instructions in the U- and V-pipelines. The U-pipeline is called the main pipeline. This pipeline can execute any Pentium instruction. The V-pipeline can only execute simple instructions. Thus, the Pentium can issue two instructions per clock cycle under certain conditions. Its instruction issue uses the following algorithm to resolve dependencies between the two instructions [1]:

```
Decode two consecutive instructions I1 and I2
if (I1 and I2 are simple instructions) AND
   (I1 is not a branch instruction) AND
   (destination of I1 ≠ source of I2) AND
   (destination of I1 ≠ destination of I2)
then
      Issue I1 to U-pipe and I2 to V-pipe
else
      Issue I1 to U-pipe.
```

Since this algorithm issues only simple instructions to the U- and V-pipes, it eliminates most *resource dependencies*. We can also avoid read-after-write (RAW) and write-after-write (WAW) dependencies as the source and destination registers of the V-pipe instruction differ from the destination register of the U-pipe instruction. We don't have to worry about the write-after-read (WAR) dependency because reads occur in an earlier pipeline stage than writes. The pipeline avoids *control dependencies* by not issuing an instruction to the V-pipe whenever a branch instruction is issued to the U-pipe.

There can be problems with resource and data dependencies between memory references. For details on how these conflicts are resolved, see [1].

The Pentium uses dynamic branch prediction. It maintains a 256-entry branch target buffer with a structure similar to the one shown in Figure 8.13*b*.

8.6.2 PowerPC

The PowerPC 604 processor has 32 general-purpose registers (GPRs) and 32 floating-point registers (FPRs). It has three basic types of execution units: integer, floating-point, and load/store units. In addition, it has a branch processing unit and a completion unit. The PowerPC 604 follows the superscalar design mentioned earlier. It can issue up to four instructions per clock. The general block diagram is shown in Figure 8.19.

The integer units consist of two single-cycle integer units (SCIUs) and a multicycle integer unit (MCIU). Most integer instructions are executed by the two SCIUs and take only a single cycle to execute. Integer multiplication and division operations are executed by the MCIU. A multiplication of two 32-bit integers takes 4 cycles whereas the division operation takes as many as 20 cycles. The floating-point unit (FPU) handles both single- and double-precision floating-point operations.

The load/store unit (LSU) provides a single-cycle, pipelined access to the cache. It has a dedicated hardware adder to perform effective address (EA) calculations. It also performs alignment and precision conversion for floating-point numbers and alignment and sign-extension of the integers. As shown in Figure 8.19, the LSU uses a 4-entry load miss buffer and a 6-entry store buffer.

The branch processing unit (BPU) uses dynamic branch prediction. It maintains a 512-entry branch history table with two prediction bits (see Figure 8.13*a*). It also keeps a 64-entry branch target address cache. In other words, the two attributes in Figure 8.13*b* are maintained in two separate tables.

It uses a six-stage instruction pipeline as shown in Figure 8.20:

1. *Fetch (IF):* As the name implies, this stage is responsible for instruction fetch and determining the address of the next instruction. The fetch unit maintains an 8-entry instruction buffer between the fetch and dispatch units (see Figure 8.19). This 6-entry buffer is divided into two 4-entry decode and dispatch buffers. The instruction fetch unit can fetch up to four instructions (128 bits) from the instruction cache per cycle.
2. *Decode (ID):* This stage performs all time-critical instruction decoding of the instructions in the instruction buffer. It moves instructions from the four-instruction decode buffer into the dispatch buffer as space becomes available.

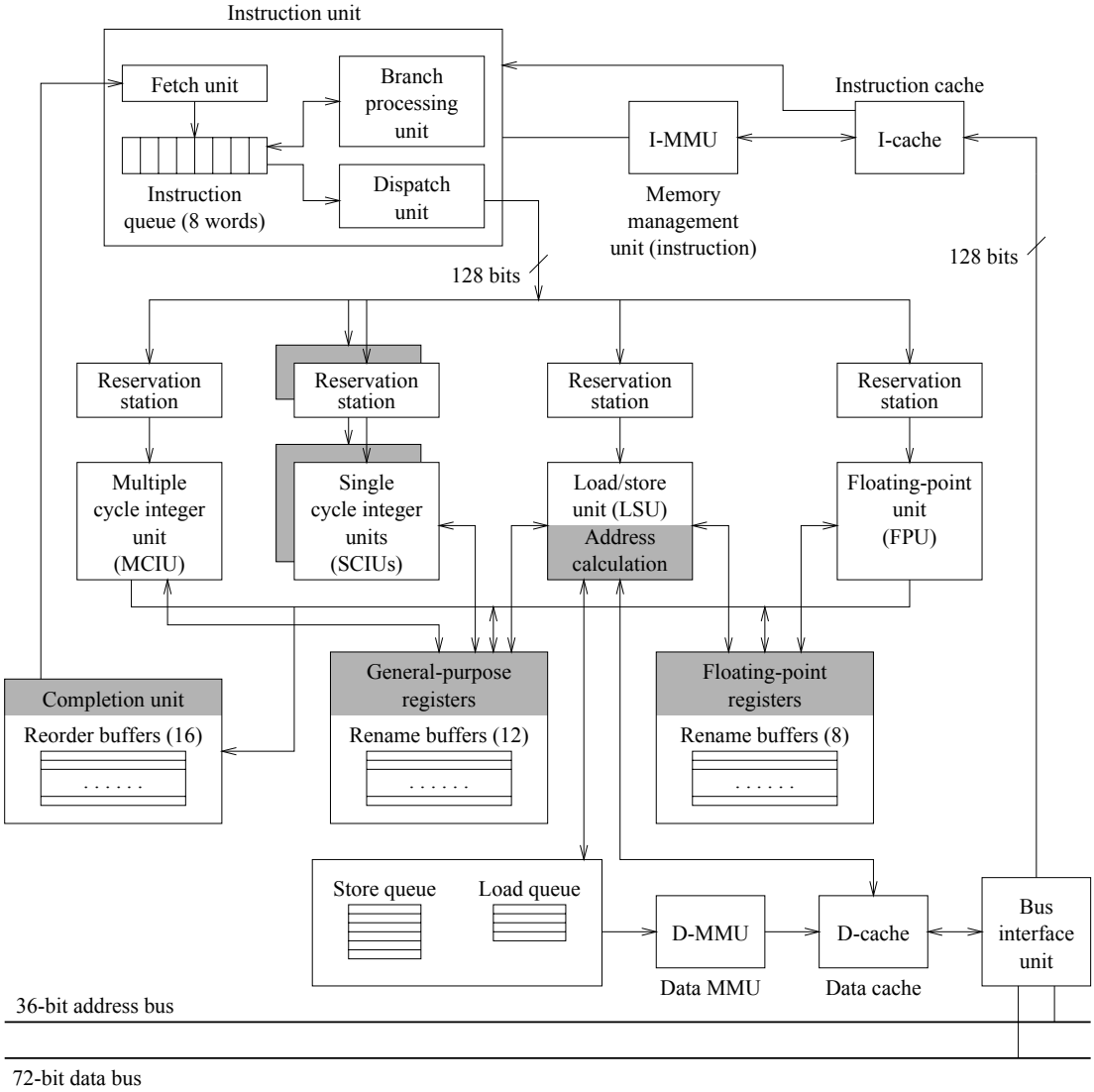


Figure 8.19 A block diagram of the PowerPC 604 processor.

3. *Dispatch (DS):* The dispatch unit performs non-time-critical decoding of instructions. Its main job is to see which of these instructions can be scheduled. It also fetches the source operands from the appropriate registers and dispatches the operands with the instruction to the execution unit.

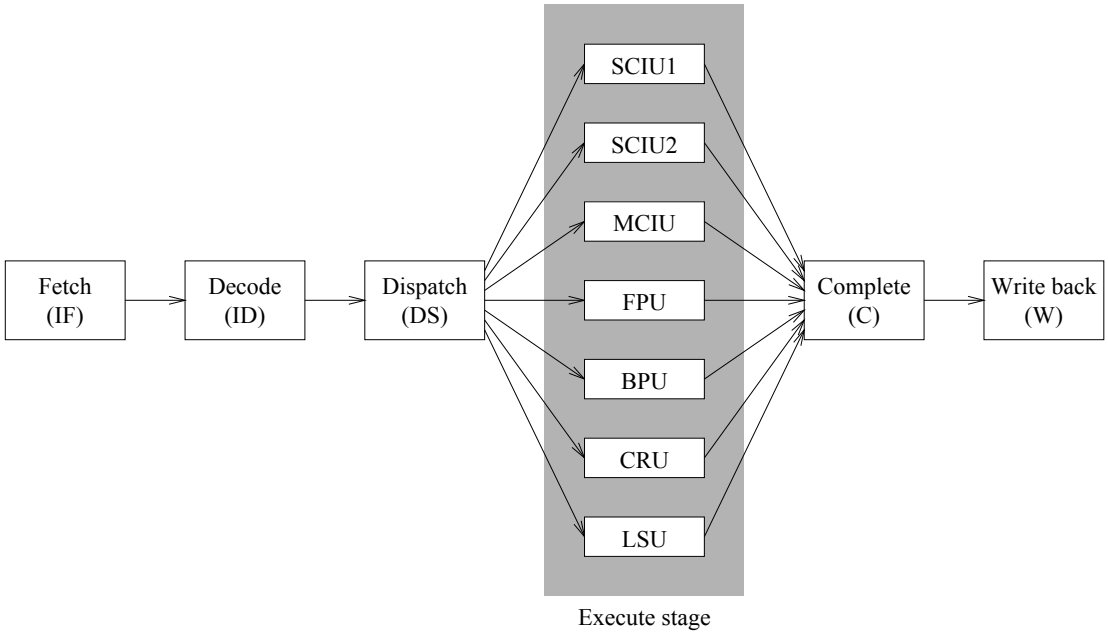


Figure 8.20 PowerPC instruction pipeline.

4. *Executed (E):* Time spent in the execution stage is determined by the type of operation performed. Potentially, there can be up to seven instructions in execution at the seven execution units shown in Figure 8.20.
5. *Complete (C):* The completion stage is responsible for maintaining the correct instruction order of execution. We give more details shortly.
6. *Write-Back (W):* This stage writes back data from the rename buffers that were written by the complete stage. Details about the rename buffers are given next.

To understand how these resources are usefully utilized to improve performance, let us see the general instruction flow. The instruction fetch unit gets up to four instructions from the instruction cache into the instruction buffer. Branch instructions are forwarded to the branch processing unit. Dynamic branch prediction is done during the fetch, decode, and dispatch stages.

The dispatch unit takes instructions from the end of the instruction queue. If the instructions can be dispatched, it fetches the operands and sends them either to the associated reservation station or directly to the execution unit. The PowerPC 604 processor maintains a 2-entry reservation station for each execution unit. The reservation stations act as buffers between the dispatch and execution units. Reservation stations allow the dispatch unit to dispatch instructions even if the operands are not yet available. Integer units allow out-of-order execution of

instructions within an integer unit as well as among the three integer units. Reservation stations for other execution units allow only in-order execution.

Since multiple instructions are in execution, some taking more time than others, the results of these instructions cannot be directly written to the destination registers. Instead, we have to hold these results in temporary registers until it is safe to write to the destination processor registers. These temporary registers are called the *rename registers* because they pretend to be a processor register given in the instruction. As shown in Figure 8.19, the PowerPC 604 processor provides 12 and 8 rename registers for the general-purpose and floating-point registers, respectively.

When an instruction is dispatched, the dispatch unit reserves space for the instruction in the 16-entry completion buffer. The completion buffer is organized as a FIFO buffer. Thus, it examines the instructions in this buffer in the order they were dispatched. This ensures that the instructions are retired strictly in program order. The process of retiring an instruction involves writing the value from the rename register to the appropriate processor register and freeing the rename register for use by other instructions. The completion unit can retire up to four instructions in a clock cycle.

8.6.3 SPARC Processor

Sun's UltraSPARC is a superscalar processor that implements the 64-bit SPARC-V9 architecture. It is capable of executing up to four instructions per cycle. Figure 8.21 shows the main components of the UltraSPARC processor.

The prefetch and dispatch unit (PDU) performs the standard instruction fetch and dispatch function discussed before. Like the PowerPC, UltraSPARC has an instruction buffer that can store up to 12 instructions. In addition, the PDU also contains branch prediction logic that implements a dynamic branch prediction scheme. Dynamic branch prediction is done using a two-bit history as shown in Figure 8.13*a*.

The integer execution unit has two ALUs, a multicycle integer multiplier, and a multicycle divider. In addition, this unit contains eight register windows and four sets of global registers. Details on these registers are presented in Appendix H.

The floating-point unit has add, multiply, and divide/square root subunits. It can issue and execute two floating-point instructions per cycle. Most floating-point instructions are pipelined with a throughput of one per cycle. The latency is independent of the precision (single or double). The divide and square root operations are not pipelined and take 12 (for single-precision) or 22 (for double-precision) cycles. These long latency instructions do not stall the processor. The floating-point instructions that follow the divide/square root instruction can be issued, executed, and retired.

UltraSPARC also supports graphics instructions and provides hardware support to execute these instructions quickly. The graphics unit (GRU) supports operations such as single-cycle pixel distance and data alignment.

The load/store unit is similar to the one we have seen in the PowerPC processor. Like the PowerPC, it also has load and store buffers. One load or store can be issued per cycle.

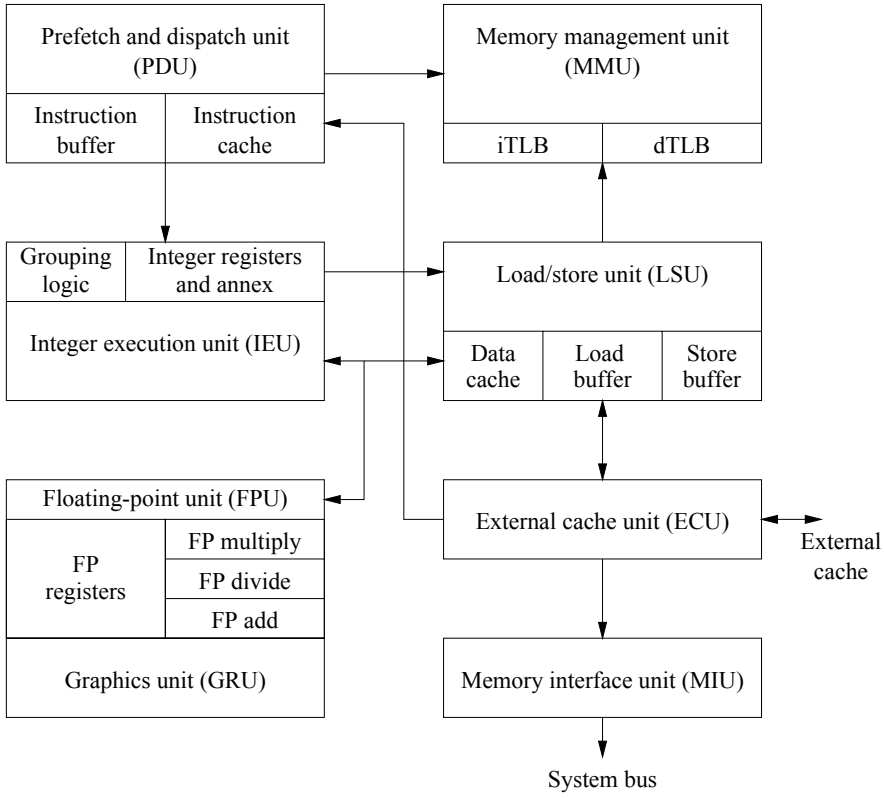


Figure 8.21 A block diagram of the UltraSPARC processor.

UltraSPARC uses the 9-stage instruction pipeline shown in Figure 8.22. Three additional stages are added to the integer pipeline in order to simplify pipeline synchronization with the floating-point pipeline.

The first two stages are the standard fetch and decode stages. The UltraSPARC fetches and decodes four instructions per cycle. The decoded instructions are placed back in the instruction buffer. A pair of pointers is used to manage the instruction buffer so that instructions are issued in order to the next stage.

The grouping stage groups and dispatches up to four instructions per cycle. From these four valid instructions, it can send up to two floating-point or graphics instructions. The grouping stage is also responsible for integer data forwarding (see Section 8.3.1) and for handling pipeline stalls due to interlocks.

The cache access stage is used by the load and store operations to get data from the data cache. ALU operations generate condition code values during this stage. Floating-point and graphics instructions start their execution during this phase. The next two stages N_1 and N_2 are

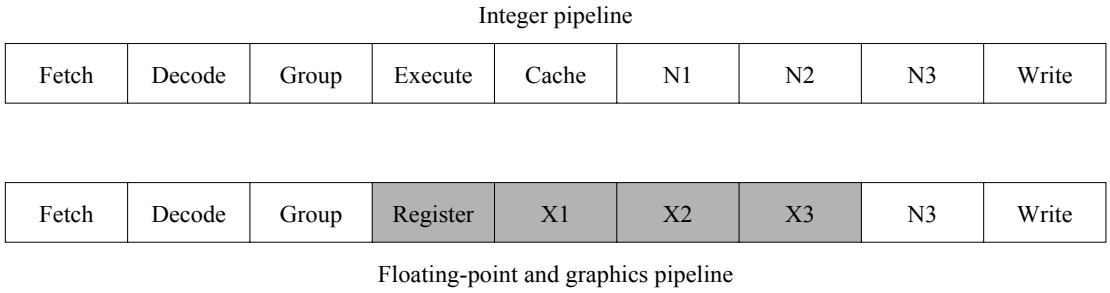


Figure 8.22 UltraSPARC's integer and floating-point pipelines.

used to complete load and store operations. These two stages also perform other tasks that are not covered here. The N_3 stage is used to resolve traps. Traps and interrupts are discussed in Chapter 20. The last stage (write stage) is used to write the results to the integer and floating-point registers.

8.6.4 MIPS Processor

The MIPS R4000 processor internal details are shown in Figure 8.23. We look at its instruction set and registers in Chapter 15. As mentioned before, the R4000 uses superpipelined design for the instruction pipeline. Its pipeline runs at twice the processor clock.

Like the SPARC processor, it uses an 8-stage instruction pipeline for both integer and floating-point instructions. The floating-point unit has three functional units: adder, multiplier, and divider. The multiplier and divider units use the adder during the final stages. Other than this restriction, the adder operations can overlap the multiplier and divider instructions. The divider unit is not pipelined; it allows only one operation at a time. The multiplier is pipelined and allows up to two instructions.

The multiplier unit can start a new double-precision multiplication every four cycles; single-precision operations can begin every three cycles. The adder can start a floating-point operation every three cycles. We have described its instruction pipeline details in Section 8.5.2.

8.7 Vector Processors

Vector machines are special-purpose systems targeted for high-performance scientific computations in which matrix and vector arithmetic operations are quite common. What is a vector? It is a linear array of numbers: integers or floating-point numbers. From a programming language perspective, a vector is a one-dimensional array. That does not mean that vector processors work only one-dimensional arrays. As we show later in this section, we can treat a column of a two-dimensional matrix as a vector.

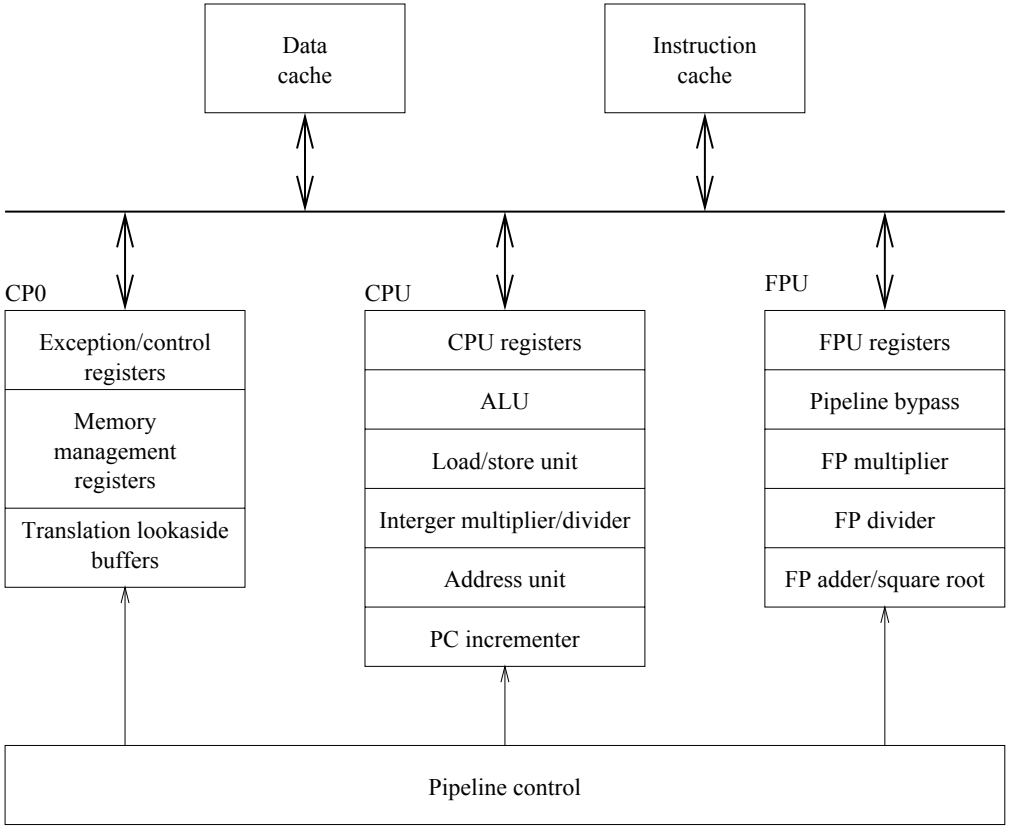


Figure 8.23 A block diagram of the MIPS R4000 processor.

8.7.1 What Is Vector Processing?

Vector machines are designed to operate at the vector level. To illustrate vector operations, let's assume that we have two vectors **A** and **B**, each with 64 elements. The number of elements in a vector is called the vector size. In our example, the vector size is 64. We write our vectors **A** and **B** as

$$\mathbf{A} = a_0, a_2, \ldots, a_{n-1}\,,$$
$$\mathbf{B} = b_0, b_2, \ldots, b_{n-1}\,.$$

Suppose that we want to add these two vectors and place the result in vector **C**. Note that adding two vectors involves adding the corresponding elements as shown below:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} = a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1}\,.$$

In high-level programming languages, we perform the vector addition by using a loop that iterates n times, where n is the vector size. In the C language, we can write this code as

```
for(i=0; i<n; i++)
    C[i] = A[i] + B[i];
```

This `for` loop iterates n times. The addition operation in this code is called the *scalar operation*. Vector processor instructions specify vector operations. For example, we just need one vector instruction to add vectors **A** and **B**. A typical vector instruction specifies four fields—three registers and an operation:

VOP	Vd	Vs1	Vs2

which performs `Vd = Vs1 VOP Vs2`. As you can see from this format, it uses the three-address format discussed in Chapter 6. Here is what a vector addition instruction $\mathbf{V3} = \mathbf{V2} + \mathbf{V1}$ looks like in the Cray X-MP assembly language:

```
V3    V2 + V1
```

We need some more details before we can use vector instructions for $\mathbf{A} + \mathbf{B}$. We discuss the necessary Cray machine details later. As this example illustrates, a single vector instruction replaces the entire loop. There is no need to increment the index variable i and test it for the boundary condition (in our example, whether it is less than 64) to jump back. Due to this and their exploitation of pipelining, vector processors give superior performance for scientific computations.

8.7.2 Architecture

A vector machine consists of a scalar unit and a vector unit. The scalar unit works on scalars and has an architecture similar to that in the traditional processors. The vector unit is responsible for performing vector operations. Similar to the move from CISC to RISC designs, vector architectures have also progressed from the memory–memory architecture to the vector–register architecture.

The first vector machines used the memory–memory architecture. In this architecture, all vector operations receive the input operands from memory and store the result in memory. The first vector machine CDC Star 100 used this architecture.

Most of the current vector machines including the machines from Cray, NEC, Fujitsu, Hitachi, and Convex use the vector–register approach. This architecture is analogous to the RISC approach. All vector operations are done on vectors located in vector registers. The result is stored in a vector register as well. As with the RISC processors, special load and store instructions move vectors between vector registers and memory.

Architecture of a vector–register machine is shown in Figure 8.24, which is based on the Cray 1 system. It consists of five components: vector registers, scalar registers, vector functional units, vector load/store unit, and main memory.

Vector Registers: We have already mentioned the need for vector registers to hold the input and result vectors. The Cary 1 and most vector processors have eight vector registers. Each register can hold 64 elements of 64 bits each.

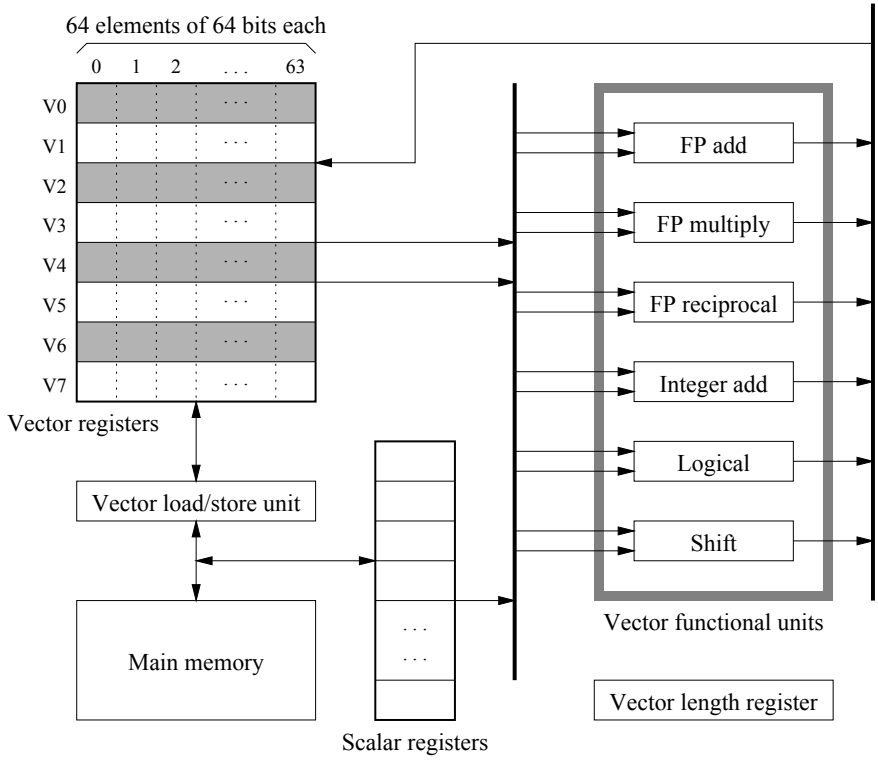


Figure 8.24 A typical vector processor architecture (based on the Cray 1).

Some processors such as the Fijitsu VP 200 allow the total available register space of 8 K elements into a programmable set of vector registers ranging from 8 to 256. When using eight vector registers, each register can hold 1024 64-bit elements; at the other end, with 256 registers, each can hold only 32 elements.

As shown in Figure 8.24, each register has two read ports and one write port to allow a high degree of overlap among vector operations on different vector registers.

Scalar Registers: The scalar registers provide scalar input to vector operations. For instance, when multiplying all elements of a matrix by a constant, a scalar register supplies the constant. For example, to compute

$$\mathbf{B} = 5 * \mathbf{X} + \mathbf{Y}$$

the constant 5 is stored in a scalar register, and vectors **X** and **Y** are stored in two vector registers. The scalar registers are also used to compute addresses for the vector load/store unit.

Vector Load/Store Unit: This unit is responsible for movement of data between vector registers and memory. This unit is pipelined to facilitate overlapped read and write operations from memory. As well, it will mask the high latency associated with main memory access. This unit supports two basic operations: a load vector operation to move a vector from memory to a vector register and a store vector to move a vector to the memory. One load/store unit is typical, however, several vector processors have more than one such unit. For example, the NEC SX/2 has eight vector load/store units.

Vector Functional Units: Vector processors contain several vector functional units for floating-point, integer, and logical operations. For example, the Cray 1 has 6 functional units as shown in Figure 8.24. The NEC SX/2 has 16 functional units: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, and 4 shift units. Note that these processors have other functional units for scalar operations, which are not our focus here.

Memory: The memory unit is different in organization than the ones we use in other processors. The memory organization should allow pipelined transfer of data to and from memory. Interleaved memory is used to support pipelined transfer of data from memory. We describe interleaved memories in Section 16.8 (see page 684).

8.7.3 Advantages of Vector Processing

Even though vector processing is not for general-purpose computation, it does offer significant advantages for scientific computing applications. To illustrate the performance advantages offered by vector systems, let's look at the simple vector addition example we have discussed before.

- *Flynn's bottleneck* can be reduced by using vector instructions as each vector instruction specifies a lot of work. As we have seen in the previous example, vector instructions can specify the work equivalent of an entire loop. Thus, fewer instructions are required to execute programs. This reduces the bandwidth required for instruction fetch.
- *Data hazards* can be eliminated due to the structured nature of the data used by vector machines. We can determine the absence of a data hazard at compile-time, which not only improves performance but also allows for planned prefetching of data from memory. The next point elaborates on this advantage.
- *Memory latency* can be reduced by using pipelined load and store operations. For example, when we fetch the first element in a 64-element addition operation, we can schedule fetching the remaining 63 elements from memory. By using interleaved memory designs, vector machines can amortize the high latency associated with memory access over the entire vector. We discuss interleaved memories on page 684.
- *Control hazards* are reduced as a result of specifying a large number of iterations in a single vector instruction. The number of iterations depends on the size of the vector registers. For example, if the machine has 64-element vector registers, each vector instruction can specify work equivalent to 64 loop iterations.

- *Pipelining* can be exploited to the maximum extent. This is facilitated by the absence of data and control hazards. Vector machines not only use pipelining for integer and floating-point operations but also to feed data from one functional unit to another. This process, known as chaining, is discussed on page 311. In addition, as mentioned before, load and store operations also use pipelining.

As you can see from this discussion, vector machines restrict the type of data to vectors, which exhibit known access patterns and reduce data and control hazards. This characteristic of the input data is heavily exploited by using pipelining in every aspect of the system operation.

8.7.4 The Cray X-MP

The first prototype of the Cray X-MP was introduced in 1982. It can support up to four CPUs. The processors communicate with each other through shared communication register clusters. In this section, we present an overview of its CPU architecture. For more details, see [30].

The Cray X-MP shares many of the RISC characteristics discussed in Chapter 1 (page 19). Vector processors use the load/store architecture. In this architecture, all operations are done on operands that are in either scalar or vector registers. Special load and store instructions are responsible for moving data between main memory and the processor registers.

As the Cray machine is targeted for scientific computations, all data in memory are 64 bits wide. It uses 22-bit addresses, where each address specifies a 64-bit word. Instructions are encoded into either a 16- or 32-bit format. In the Cray's terminology, a 16-bit instruction encoding is called *one parcel*, and the 32-bit encoding is *two parcels*.

The Cray X-MP has three types of registers: *address*, *scalar*, and *vector registers*. It has eight 24-bit address registers A0 to A7. These registers are mainly used for holding memory addresses for load and store operations. The use of 24-bit address registers may seem strange as the X-MP uses 22-bit addresses. The reason is that up to four instructions can be at the addressed location. Thus, we need two additional bits to identify one of the four instructions that could be packed into a 64-bit word.

The Cray X-MP uses pipelined execution for address, scalar, and vector operations. It has several address, scalar, and vector functional units. The address section of the processor has two functional units to perform address arithmetic operations. Details about the address functional units are shown below:

Address functional unit	Number of stages
24-bit integer ADD	2
24-bit integer MULTIPLY	4

The Cray provides several instructions to manipulate addresses. Here are some example address instructions in the Cray assembly language (CAL) format:

```
Ai  Aj+Ak    (Ai = Aj+Ak)
Ai  Aj*Ak    (Ai = Aj*Ak)
```

Table 8.4 Sample Cray X-MP scalar functional units

Scalar functional unit	Number of stages
Integer ADD (64-bit)	3
64-bit SHIFT	2
128-bit SHIFT	3
64-bit LOGICAL	1
POP/PARITY (population or parity)	4
POP/PARITY (leading zero count)	3

```
Ai  Aj−Ak     (Ai = Aj−Ak)
Ai  −Ak       (Ai = −Ak)
Ai  Aj+1      (Ai = Aj+1)
Ai  Aj−1      (Ai = Aj−1)
Ai  expr      (Ai = expr)
```

The last instruction can be used to assign constant values (`expr`) to A registers. As we show later, address registers are also used for holding short integers. For example, address registers are used to hold the shift count for shift instructions.

The scalar section of Cary X-MP has eight 64-bit scalar registers (S0 to S7) and four types of functional units: ADD, LOGICAL, SHIFT, and POP/PARITY. The first three operations are the standard addition, logical, and shift operations. The POP/PARITY unit counts the number of ones or zeros or determines parity. Table 8.4 shows the number of stages used in each scalar functional unit.

Some example scalar instructions are given below:

```
Si  Sj+Sk     (Si = Sj+Sk)
Si  Sj!Sk     (Si = Sj OR Sk)
Si  Si>Ak     (Si = Si>>Ak)
```

The first instruction places the sum of `Sj` and `Sk` in the `Si` register. The second instruction performs the logical bitwise OR operation, and the third one right-shifts the contents of `Si` by `Ak` positions.

Like the Cray-1 shown in Figure 8.24, the X-MP has eight 64-element vector registers V0 to V7. Each vector register can hold 64 words (each word is 64 bits wide). Table 8.5 summarizes the details of the vector functional units. For now, ignore the last two columns in this table.

A sample of the Cray X-MP vector instructions is shown in Table 8.6. Each vector instruction works on the first VL elements. The VL value is in the vector length register (see Figure 8.24). The first group of instructions in this table shows the types of addition instructions provided by the X-MP. Similar instructions are available for multiplication and subtraction operations. Multiplication and subtraction instructions use * and − operators, respectively. There

Table 8.5 Sample Cray X-MP vector functional units

Vector functional unit	Number of stages	Available to chain	Vector results
Integer ADD (64-bit integer)	3	8	VL + 8
64-bit SHIFT (64-bit logical)	3	8	VL + 8
128-bit SHIFT (128-bit logical)	4	9	VL + 9
Full vector 64-bit LOGICAL	2	7	VL + 7
Second vector 64-bit LOGICAL	4	9	VL + 9
POP/PARITY	5	10	VL + 10
Floating ADD	6	11	VL + 11
Floating MULTIPLY	7	12	VL + 12
Reciprocal approximation	14	19	VL + 19

is no division operation. Instead, X-MP provides the reciprocal operation as shown in Table 8.6. We have shown only the logical AND operation in this table. The Cray X-MP also provides instructions to perform logical OR (!) and XOR (\) operations.

8.7.5 Vector Length

In our discussion so far, we have not said anything about the size of the actual vector. We conveniently assumed that the size of the vector register is equal to the size of the vector we have. What happens if this is not true? In particular, we have to handle two cases: the vector size is less than the vector register size, and the vector size is larger than the vector register size. For concreteness, we assume 64-element vector registers as provided by the Cray systems. We first look at the simpler of these two problems.

Handling Smaller Vectors

If our vector size is smaller than 64, we have to let the system know that it should not operate on all 64 elements in the vector registers. This is fairly simple to do by using the vector length register. The VL register holds the valid vector length. All vector operations are done on the first VL elements (i.e., elements in the range 0 to VL $-$ 1). There are two instructions to load values into the VL register:

```
VL   1        (VL = 1)
VL   Ak       (VL = Ak where k≠0)
```

For example, if the vector length is 40, the following code can be used to add two vectors in registers V3 and V4:

Table 8.6 Sample Cray X-MP instructions

Instruction	Meaning	Description
`Vi Vj+Vk`	`Vi = Vj+Vk` Integer add	Add corresponding elements (in the range 0 to VL − 1) from `Vj` and `Vk` vectors and place the result in vector `Vi`
`Vi Sj+Vk`	`Vi = Sj+Vk` Integer add	Add the scalar `Sj` to each element (in the range 0 to VL−1) of `Vk` vector and place the result in vector `Vi`
`Vi Vj+FVk`	`Vi = Vj+Vk` Floating-point add	Add corresponding elements (in the range 0 to VL − 1) from `Vj` and `Vk` vectors and place the floating-point result in vector `Vi`
`Vi Sj+FVk`	`Vi = Sj+Vk` Floating-point add	Add the scalar `Sj` to each element (in the range 0 to VL − 1) of `Vk` vector and place the floating-point result in vector `Vi`
`Vi ,A0,Ak`	`Vi = M(A0)+Ak` Vector load with stride `Ak`	Load into elements 0 to VL − 1 of vector register `Vi` from memory starting at address `A0` and incrementing addresses by `Ak`
`Vi ,A0,1`	`Vi = M(A0)+1` Vector load with stride 1	Load into elements 0 to VL − 1 of vector register `Vi` from memory starting at address `A0` and incrementing addresses by 1
`,A0,Ak Vi`	`Vi = M(A0)+Ak` Vector store with stride `Ak`	Store elements 0 to VL − 1 of vector register `Vi` in memory starting at address `A0` and incrementing addresses by `Ak`
`,A0,1 Vi`	`Vi = M(A0)+1` Vector store with stride 1	Store elements 0 to VL − 1 of vector register `Vi` in memory starting at address `A0` and incrementing addresses by 1
`Vi Vj&Vk`	`Vi = Vj&Vk` Logical AND	Perform bitwise-AND operation on corresponding elements (in the range 0 to VL − 1) from `Vj` and `Vk` vectors and place the result in vector `Vi`
`Vi Sj&Vk`	`Vi = Sj&Vk` Logical AND	Perform bitwise-AND operation on 0 to VL − 1 elements of `Vk` and scalar `Sj` and place the result in vector `Vi`
`Vi Vj>Ak`	`Vi = Vj>>Ak` Right-shift by `Ak`	Right-shift 0 to VL − 1 elements of `Vj` by `Ak` and place the result in vector `Vi`
`Vi Vj<Ak`	`Vi = Vj<<Ak` Left-shift by `Ak`	Left-shift 0 to VL − 1 elements of `Vj` by `Ak` and place the result in vector `Vi`

```
A1  40      (A1 = 40)
VL  A1      (VL = 40)
V2  V3+FV4  (V2 = V3 + V4)
```

Since we cannot write

```
VL  40
```

we have to use the two-instruction sequence to load 40 into the VL register. The last instruction specifies floating-point addition of vectors V3 and V4. Since the VL is 40, only the first 40 elements are added.

Handling Larger Vectors

The VL register is useful for handling smaller vector sizes, but it does not help for larger vector sizes. Suppose we have 200-element vectors (i.e., $N = 200$). How do we use the vector instructions to add two such vectors? The case of larger vectors is handled by a technique known as *strip mining*.

In strip mining, the vector is partitioned into strips of 64 elements. This leaves one odd-size piece that may be less than 64 elements. The size of this piece is given by (N mod 64). We load each strip into a vector register and apply the vector addition instruction. The number of strips is given by $(N/64) + 1$. For our example, we divide the 200 elements into four pieces: three pieces with 64 elements and one odd piece with 8 elements. We use a loop that iterates four times: in one of the iterations, we set VL to 8 and the remaining three iterations will set the VL register to 64.

8.7.6 Vector Stride

To understand vector stride, we have to consider how the elements are stored in memory. Let's first look at vectors. Since vectors are one-dimensional arrays, storing a vector in memory is straightforward: vector elements are stored as sequential words in memory. If we want to fetch 40 elements, we read 40 contiguous words from memory. These elements are said to have a stride of 1. That is, to get to the next element, we add 1 to the current element. Note that the distance between successive elements is not measured in bytes; rather in number of elements.

We will need nonunit vector strides for multidimensional arrays. To see why, let's focus on two-dimensional matrices. If we want to store a two-dimensional matrix in memory, we have to linearize it. We can do this in one of two ways: *row-major* or *column-major* order. Most languages, except FORTRAN, use the row-major order. In this ordering, elements are stored in row order: row 0, row 1, row 2, and so on. In the column-major order, which is used by FORTRAN, elements are stored column by column: column 0, column 1, and so on. As an example, consider the following 4×4 matrix:

$$\mathbf{A} = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{bmatrix}.$$

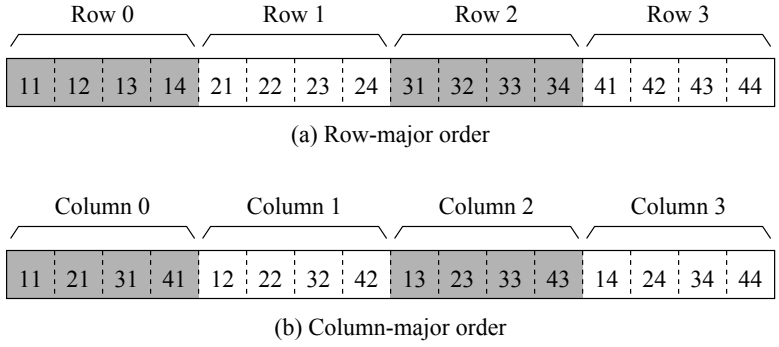


Figure 8.25 Memory layout of vector **A**.

This matrix is stored in memory as shown in Figure 8.25. For more details on accessing individual elements, see Section 11.4.2 (page 450).

Assuming row-major order for storing, how do we access all elements of column 0? Clearly, these elements are not stored contiguously. We have to access 0, 4, 8, and 12 elements in the memory array. Since successive elements are separated by 4 elements, we say that the stride is 4. Vector machines provide load and store instructions that take the stride into account. We can see from Table 8.6 that X-MP supports both unit and nonunit stride access. For example, the instruction

```
Vi  ,A0,Ak
```

loads vector register `Vi` with stride `Ak`. Since unit stride is very common, a special instruction

```
Vi  ,A0,1
```

is provided. Similar instructions are available for storing vectors in memory.

8.7.7 Vector Operations on the Cray X-MP

All vector units produce a result on each clock once the pipeline is full. Pipelined operations on the Cray X-MP proceed in three phases: *setup*, *execution*, and *shutdown*. During the setup phase, the functional unit is programmed to perform the appropriate operation and the connections to the source and destination vector registers are established. The setup time for all vector units is three clock cycles. The pipeline performs the computation during the execution phase. The number of cycles required to complete the computation depends on the vector length and the number of stages in the functional unit. The final shutdown phase represents the time difference between when the final result comes out of the pipeline and when the destination register can be used for another operation. For the X-MP, the shutdown phase also takes three clock cycles.

With this information, it is relatively straightforward to estimate the number of clock cycles required to execute a piece of code. We illustrate the process by means of examples.

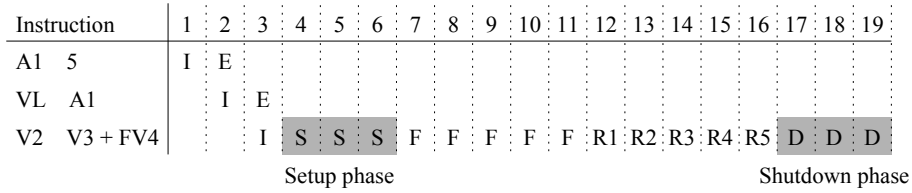


Figure 8.26 Timing analysis of a simple vector addition operation.

Example 8.1 *A simple vector add operation.*
As a first example, consider the following code that adds the first five elements of vectors **V**2 and **V**3 and places the result in **V**1:

```
A1   5
VL   A1          (VL = 5)
V1   V2+FV3
```

The timing for this code is shown in Figure 8.26. The first instruction takes two clock cycles: one for instruction fetch and the other for execution. The second instruction fetch is done during the second clock cycle. It also takes two clocks. The vector instruction is fetched in clock cycle 3. This is followed by a three-cycle setup phase (shown by S). The floating-point add unit is a six-stage pipeline (see Table 8.5). Thus, for the next five clocks, the pipeline gets filled (shown by F). The first result comes out during clock 12. We get one result out for the next four clock cycles. The last three cycles are for the shutdown (D) phase. □

From this example, we can see that the add operation is completed in 16 clocks, excluding the instruction fetch. That is, 16/5 = 3.2 clocks per addition. We reduce the overhead by increasing the vector length. In general, the number of clocks required to add VL size vectors is

$$3 + 6 + (\text{VL} - 1) + 3\,.$$

The first three cycles are for the startup phase. The six cycles are the number of stages in the add pipeline. Thus, by nine clocks, the result is out. The remaining (VL − 1) results will require (VL − 1) clocks as we get one result per clock. The last three cycles are for the shutdown phase. For example, if we use 64 element vectors, the number of clocks for each addition reduces to 75/64 ≈ 1.2 clocks.

By generalizing the previous expression to an n-stage pipeline functional unit, we get

$$\text{Number of clock required} = 3 + n + (\text{VL} - 1) + 3 = \text{VL} + n + 5\,.$$

This matches with the expression given in the last column of Table 8.5.

The next example shows how overlapped execution among multiple functional units improves the performance.

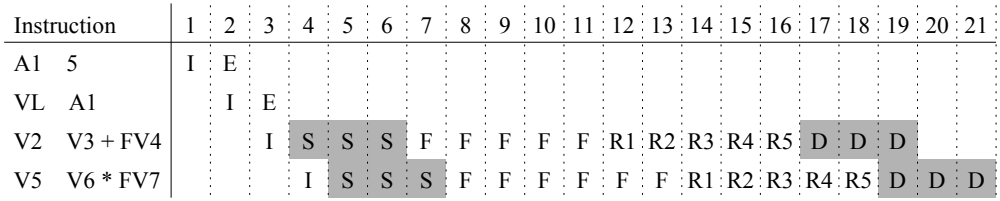


Figure 8.27 Overlapped execution of floating-point addition and multiplication operations.

Example 8.2 *Overlapped execution of vector operations.*
In this example, we consider two vector operations that are independent. These instructions can be executed in any order.

```
A1   5
VL   A1          (VL = 5)
V1   V2+FV3      (Floating-point addition of V2 and V3)
V4   V5*FV6      (Floating-point multiplication of V5 and V6)
```

The timing for this code is shown in Figure 8.27. The first three rows are exactly the same as in the previous example. The floating-point multiply unit is a seven-stage pipeline. Thus, the first result is available during clock cycle 14. As in the add instruction, there are three clocks for the setup phase and three for the shutdown phase.

This example illustrates the overlapped execution of the floating-point add and multiply units. We get significant performance gains by this overlapped execution. Without overlapping these two operations, we would have required 16 clocks for the addition and 17 clocks for the multiplication. Using overlapped execution, we complete both operations in 18 clocks. □

8.7.8 Chaining

We have demonstrated in the last example that overlapped execution is important to get performance gains. Since there is no conflict on the vector registers used by the addition and multiplication operations, we could overlap the execution of the two operations. What happens if there is a conflict as in the following example?

```
A1   5
VL   A1          (VL = 5)
V1   V2+FV3      (Floating-point addition of V2 and V3)
V4   V5*FV1      (Floating-point multiplication of V5 and V1)
```

The multiplication operation takes values from **V**1, which are produced by the addition operation. Due to this dependency, we cannot independently schedule these two instructions. As we have seen, sequential execution of these two instructions takes 16 + 17 = 33 clocks. This is where chaining is useful.

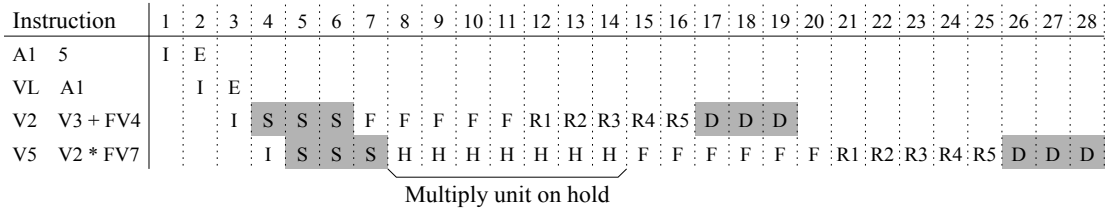


Figure 8.28 A vector chaining example.

Chaining allows feeding of data from one operation to the next without waiting for the first operation to complete. For example, once the first result of addition is placed in **V**1, we could make that value available for the multiplication instruction. The Cray X-MP allows using the first result after two clock cycles of delay.

Example 8.3 *A chaining example.*
To understand how chaining speeds up execution, let's consider the example code given before. The execution timing is shown in Figure 8.28. The first three rows are the same as in the previous examples. The vector multiplication instruction uses chaining to get values from the **V**1 vector register. The addition operation produces the first result R1 during clock cycle 12. This result is available for use by the multiplication two clocks later. Until that time, the multiplication pipeline is in the hold (H) state. From that point onward, execution proceeds in the normal fashion. Thus, by using chaining, we complete both operations in 25 cycles. This is significantly faster than executing without chaining: 33 cycles versus 25 cycles, about 24% improvement. □

8.8 Performance

This section briefly discusses the performance of pipelined and vector processors.

8.8.1 Pipeline Performance

Performance of pipelined execution depends on the number of stages in the pipeline as well as the number of computations performed (e.g., vector length). Speedup is used to measure the relative performance of the pipelined system with the corresponding nonpipelined system. The speedup is defined as

$$\text{Speedup} = \frac{\text{Nonpipelined execution time}}{\text{Pipelined execution}}.$$

Ideally, an n-stage pipeline should give a speedup of n. However, the pipeline would not operate at full capacity all the time due to the following:

- *Pipeline Fill:* When the computation is initiated, the pipeline is empty. It takes n cycles to fill an n-stage pipeline. Until that time, all stages of the pipeline are not busy.

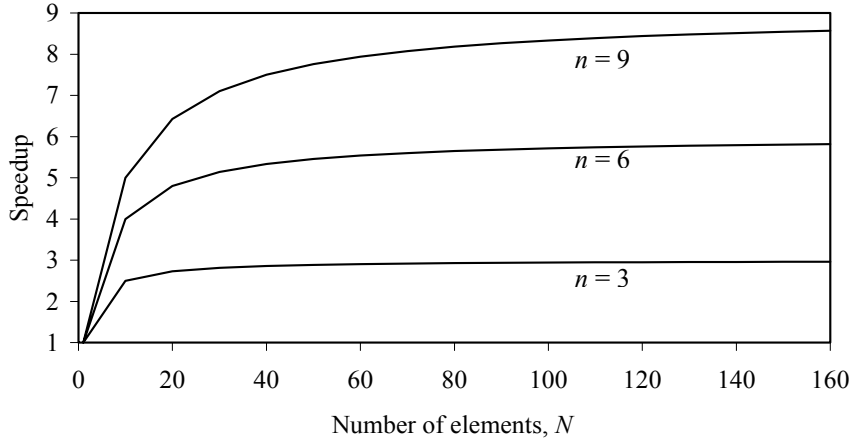


Figure 8.29 Pipeline performance as a function of the vector size N.

- *Pipeline Drain:* At the trailing end of computation, no further input enters the pipeline. However, we have to drain the pipeline to get out all the results. Again, during this draining phase, all stages are not busy.

To compute speedup, let us assume that we perform N computations. Each computation takes $n \times T$ time units. Thus, on a nonpipelined system, it takes $N \times n \times T$ time. If we use an n-stage pipeline, we get the first result out in $n \times T$ time. But after that the pipeline is full and produces one result every T time units. Thus, we need an additional $(N-1)T$ time for the remaining $(N-1)$ results. Therefore, on a pipelined system, we need

$$n \times T + (N-1)T = (n+N-1)T\,.$$

We can now compute the speedup as

$$\text{Speedup} = \frac{n \times N \times T}{(n+N-1)T} = \frac{n \times N}{n+N-1}\,.$$

By rewriting the above expression, we get

$$\text{Speedup} = \frac{1}{\frac{1}{N} + \frac{1}{n} - \frac{1}{n \times N}}\,.$$

We get the ideal speedup of n as $N \to \infty$. This condition implies that we are ignoring the "fill" and "drain" effects on the pipeline performance. At the other extreme, if $N = 1$, we get a speedup of 1, which agrees with our assertion that pipelining does not improve the execution time of a single computation. Figure 8.29a shows the speedup as a function of N in an n-stage pipeline.

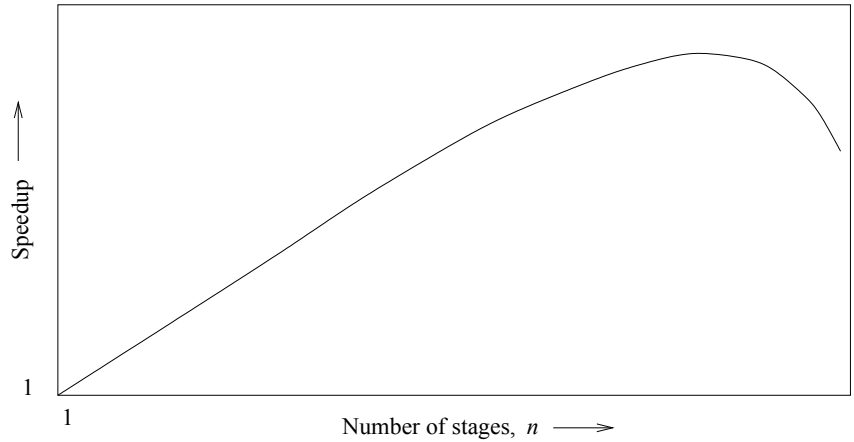


Figure 8.30 Pipeline speedup as a function of pipeline depth n.

This expression also tells us that we can improve the speedup by increasing the number of stages n. Instead of using an n-stage pipeline, we can design a $2n$-stage pipeline where each stage does half of the work done by one stage in the previous design. The speedup is than given by

$$\text{Speedup} = \frac{1}{\frac{1}{N} + \frac{1}{2n} - \frac{1}{2n \times N}}.$$

If we ignore other factors, we improve speedup as we increase the number of pipeline stages. The number of pipeline stages is also called the *pipeline depth*. This can be seen from Figure 8.29. For example, when we increase the pipeline depth n from 3 to 6, the speedup nearly doubles. Further increase in speedup is obtained by increasing the pipeline depth to 9 stages.

It may appear from this that by increasing the pipeline depth, we can improve the performance. However, in practice, larger pipeline depth reduces performance (see Figure 8.30). This reduction is mainly contributed by the data and control hazards. The probability of a pipeline stall occurring due to these dependencies increases with the pipeline depth. Both these dependencies will cause the work in the pipeline to be thrown out. In addition, longer pipelines mean more time for branching when control hazards occur.

8.8.2 Vector Processing Performance

As we have seen in the Cray X-MP machine, vector systems use pipelining for address, scalar, vector, and load/store operations. Thus, the performance gains we get are based on the performance improvements due to pipelining. In this section, we look at the impact of vector register length on the performance of vector machines. As mentioned on page 308, we have to resort to strip mining to handle any mismatch between the actual vector and the vector register size VL.

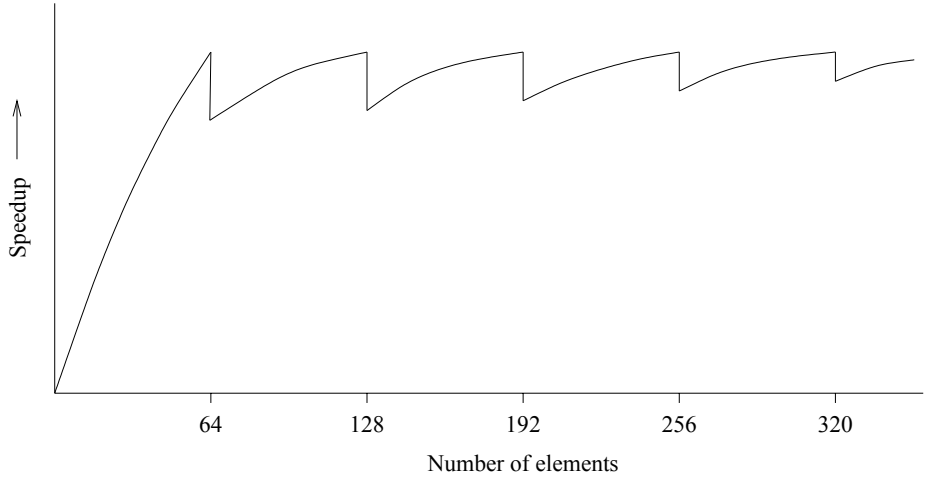


Figure 8.31 Vector processors exhibit "saw-tooth" shaped performance sensitivity to vector length.

We get improved speedup as the vector size increases from 1 to VL. This is due to the amortization of the pipeline fill cost. As an example, consider a 10-stage pipeline with 64-element vector registers (VL = 64). Neglecting all other influences, nonpipelined operation on 64 elements takes 640 time units. On a pipelined unit, it takes 10 + 63 = 73 time units giving us a speedup of 640/73 = 8.77. Now look at what happens to the speedup when we increase the vector size by one element. On the nonpipelined unit, it takes 650 time units, whereas the pipelined unit takes 10 + 63 + 10 = 83. Thus, the speedup drops to 650/83 = 7.83. Similarly, you can verify that the speedup reaches 8.77 for 128-element vectors but drops to 8.27 for 129 elements. Thus, the performance peaks at multiples of VL but drops immediately after that, leading to the "saw-tooth" shaped performance shown in Figure 8.31. As with the pipeline performance, practical issues such as the time needed to load the vector registers would give much worse performance than that we can get from an ideal pipeline. For more details on pipeline and vector performance, see [19, 31, 30].

8.9 Summary

We have demonstrated that pipelining can significantly improve the performance of a processor. The key factor is the overlapped execution of multiple operations. Although pipelining does not reduce the time needed to perform a single operation, it does increase the throughput using overlapped execution. Pipelining involves highly coordinated movement of data from stage to stage. Any delay in one stage can seriously affect the performance of the whole pipeline.

We have discussed three types of resource conflicts—resource, data, and control hazards—that can cause pipeline stalls. We have looked at potential solutions to these resource conflicts. All processors use pipelining to derive better performance. We have presented details about four processors—the Pentium, PowerPC, SPARC, and MIPS.

Performance of a pipelined processor can be improved by using superscalar, superpipelined, and VLIW techniques. The MIPS processor uses the superpipeline design whereas the other two processors use superscalar designs. It should, however, be noted that processors tend to use some features of each of these techniques. In Chapter 14, we discuss the architecture of Intel's 64-bit Itanium processor that uses some VLIW design features.

Vector processors exploit pipelining fully to get substantial performance improvement. These machines are best suited for scientific computations wherein the structured data allow pipelined designs for most operations including the load and store operations. These machines use vector registers that can hold a small vector. The strip mining technique can be used to process larger vectors. We have presented details about the Cray X-MP processor. Performance of pipelined and vector processors is briefly discussed in the last section.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Branch prediction
- Bypassing
- Column-major order
- Completion buffer
- Control hazards
- Data hazards
- Delay slot
- Delayed branch execution
- Dual pipeline
- Dynamic branch prediction strategy
- Fixed branch prediction strategy
- Floating-point addition pipeline
- Flynn's bottleneck
- Hazards
- Instruction execution
- Load/store unit
- Memory–memory architecture
- Nullification
- Pipeline stages
- Pipeline stalls
- Read-after-write (RAW) dependency
- Register forwarding
- Register interlocking
- Rename registers
- Resource hazards
- Retiring instructions
- Row-major order
- Scalar registers
- Static branch prediction strategy
- Strip mining
- Structural hazards
- Superpipelined processor
- Superscalar
- Vector chaining
- Vector length
- Vector processor architecture
- Vector–register architecture
- Vector registers
- Vector stride
- VLIW
- Write-after-read (WAR) dependency
- Write-after-write (WAW) dependency

8.10 Exercises

8–1 What is the disadvantage of having heterogeneous pipeline stages (i.e., not all stages require the same amount time to process)?

8–2 Describe the three major resource conflicts that can potentially stall the pipeline.

8–3 Explain how instruction prefetching can minimize stalls due to resource conflicts.

8–4 What is the reason for data hazards? How can we minimize their adverse impact on pipeline performance?

8–5 Explain why branching affects performance of a pipeline.

8–6 What are some of the techniques used to reduce the penalty associated with branching?

8–7 What is the motivation for introducing delayed branch execution in processors like the MIPS and SPARC?

8–8 Why do we need nullification in delayed branch execution?

8–9 Discuss the three branch prediction strategies presented in this chapter.

8–10 Why does the static prediction strategy give improved prediction accuracy over the fixed strategy?

8–11 Compared to the static branch prediction strategy, dynamic strategy leads to better prediction accuracy. Explain why.

8–12 Discuss the advantages and disadvantages of static and dynamic branch prediction strategies.

8–13 What is the reason for selecting 2-bit states in the finite state machine shown in Figure 8.12?

8–14 Explain the basic features of a superscalar processor.

8–15 Explain why superscalar designs give better performance.

8–16 What is the difference between the superscalar and superpipelined processors?

8–17 How are vector processors different from the traditional processors?

8–18 What are the reasons for getting substantial performance improvements with vector processing?

8–19 What is the purpose of the vector length register in vector processors?

8–20 Explain how larger vectors (that exceed the vector register size) are processed to take advantage of vector processing.

8–21 Why is it necessary to have vector load and store instructions that support vector stride?

8–22 Explain the chaining techniques used in vector processors.

8–23 For the Cray X-MP machine, give the timing diagram (similar to that in Figure 8.27 on page 311) in executing the following code:

```
A1   10
VL   A1          (VL = 10)
V1   V2 + V3     (Integer addition of V2 and V3)
V4   V5 + FV6    (Floating-point addition of V5 and V6)
```

For details on the integer and floating-point addition pipelines, see Table 8.5 on page 306.

8–24 For the Cray X-MP machine, give the timing diagram (similar to that in Figure 8.28 on page 312) in executing the following code:

```
A1   10
VL   A1          (VL = 10)
V1   V2 + V3     (Integer addition of V2 and V3)
V4   V5 + FV6    (Floating-point addition of V5 and V6)
V7   V4 * FV1    (Floating-point multiplication of V4 and V1)
```

For details on the integer and floating-point pipelines, see Table 8.5 on page 306.

8–25 Explain why increasing the pipeline depth beyond a certain point deteriorates its performance.

8–26 Explain the basic reason for the saw-tooth shaped performance of vector processors.

Chapter 9

Overview of Assembly Language

Objectives

- To introduce the basics of the Pentium assembly language;
- To discuss data allocation statements of the assembly language;
- To describe the Pentium data transfer instructions;
- To provide an overview of the Pentium instruction set;
- To examine how constants and macros are defined in the assembly language;
- To present examples of the Pentium assembly language programs.

The objective of this chapter is to review the basics of the Pentium assembly language. Assembly language statements can either instruct the CPU to perform a task, or direct the assembler during the assembly process. The latter statements are called assembler directives. Section 9.2 discusses the format and types of assembly language statements.

Assemblers provide several directives to reserve storage space for variables. These directives are discussed in Section 9.3. A typical assembly language instruction consists of an operation code to indicate the type of operation to be performed and the specification of required operands. Section 9.4 describes some basic addressing modes used to specify operands.

The Pentium instruction set can be divided into several groups of instructions. Section 9.5 discusses the instructions that transfer data, including `mov`, `xchg`, *and* `xlat` *instructions. Section 9.6 provides an overview of some of the Pentium instructions belonging to the other groups. Later chapters discuss other instructions supported by Pentium.*

Section 9.7 describes the assembler directives to define constants, numeric as well as string. Assemblers also provide a mechanism to define macros. These details are presented in Section 9.8. Several assembly language program examples are given in Section 9.9. The chapter concludes with a summary.

9.1 Introduction

Assembly language is a low-level programming language that is directly influenced by the instruction set and architecture of a processor. Assembly language code must be translated into the processor's machine language. This translation is done by a piece of software called the *assembler*. MASM (Microsoft assembler), TASM (Borland turbo assembler), and NASM (Netwide Assembler) are some of the popular assemblers available for the Intel processors.

Compared to assembly language, high-level languages (HLLs) such as C offer several advantages in program development, as discussed in Chapter 1 (see page 10). The three main advantages are: faster program development, easier program maintenance, and portability. We have also noted that there are two main reasons why programming is still done in assembly language: efficiency and accessibility to system hardware.

Our objective here is to use the assembly language as a tool to study computer organization. In this part of the book, we look at the Pentium assembly language to study the Pentium processor. We have described the basic architecture of the Pentium in Chapter 7. In a later part, we study the MIPS assembly language. Thus, you will have exposure to both CISC and RISC processors and their assembly languages.

In this part of the book, we write assembly language programs using the MASM/TASM syntax. NASM, however, uses a slightly different syntax. We describe NASM in detail in Appendix E.

9.2 Assembly Language Statements

Assembly language programs are created out of three different classes of statements. Statements in the first class tell the CPU what to do. These instructions are called *executable instructions*, or *instructions* for short. Each executable instruction consists of an *operation code* (*opcode* for short). Executable instructions cause the assembler to generate machine language instructions. Each executable statement typically generates one machine language instruction.

The second class of statements provides information to the assembler on various aspects of the assembly process. These instructions are called *assembler directives* or *pseudoops*. Assembler directives are nonexecutable and do not generate machine language instructions.

The last class of statements, called *macros*, is used as a shorthand notation for a group of statements. Macros permit the assembly language programmer to name a group of statements and refer to the group by the macro name. During the assembly process, each macro is replaced by the group of statements that it represents and is assembled in place. This process is referred to as *macro expansion*. We use macros to provide the basic input and output capabilities to standalone assembly language programs. We briefly discuss macros in Section 9.8. A more detailed discussion is in [11].

Assembly language statements are entered one per line in the source file. Even though up to 128 characters can be used in a line, it is good practice to limit a line to 80 characters so that it can be displayed on the screen. Except for a few statements, most assembly language statements require far fewer characters than 80.

All three classes of the assembly language statements use the same format:

```
[label]     mnemonic     [operands]     [;comment]
```

The fields in the square brackets are optional in some statements. As a result of this format, it is a common practice to align the fields to aid readability.

Label: This is an optional field. The label field serves two distinct purposes: it is used to represent either an identifier or a constant. When a label appears in an executable instruction, it is used as a marker to identify the instruction. Then, for example, you can make the program execution jump to the labeled instruction. In this case, the label represents the memory address of the instruction. When used with certain assembler directives such as `EQU`, the label represents a constant.

Mnemonic: This is a required field and identifies the purpose of the statement. In certain lines of code, this field is not required. Examples include lines consisting of a comment, or a label, or a label and a comment.

Operands: Operands specify the data to be manipulated by the statement. The number of operands required depends on the specific statement or directive. For instance, executable statements may have zero, one, two, or three operands.

Comment: This is an optional field and serves the same purpose as comments in a high-level language. Comments play a more important role in assembly language, as it is a low-level language. Assembler ignores all comments. Comments begin with a semicolon (;) and extend until the end of the line. Since the readability of assembly language programs is poor, comments should be generously added to improve readability. It is good programming practice to explain the functionality of a group of statements by several lines of comments and then add brief comments to selected code lines within the group. This is the practice followed in this book.

Now let us look at some sample assembly language statements.

```
repeat:   inc      result   ;increment result by 1
```

The label `repeat` can be used to refer to this particular statement. The mnemonic `inc` indicates increment operation, which is done on the data stored in memory at `result`. The comment simply explains what the instruction is doing. We avoid such self-explanatory comments. The following assembler directive defines the constant `CR`. The ASCII carriage return value is assigned to it by the EQU directive.

```
CR    EQU    0DH      ;carriage return character
```

In the previous two examples, the label field has two different forms. The label in the executable instruction is followed by a colon (:) but not in the directive statement. Labels and other names can be formed from upper and lowercase letters (a to z, A to Z), digits (0 through 9), and special characters (_, %, ?, $, ., @).

A name may not begin with a digit and if a period is used, it must be the first character. For example, `jump2` and `repeat` are valid but not `go.back` and `2_jump`. Other characters may be used in any position. Among the special characters, the underscore character is frequently used to aid readability. Certain reserved words that have special meaning to the assembler are not allowed as names. These include mnemonics such as `inc` and `EQU`.

The assembler is normally case insensitive. For example, labels `repeat` and `REPEAT` are treated the same. The assembler can be made case sensitive by using an option (e.g., the `/ml` option with TASM). We follow the convention that the source code is normally in lowercase except for the directive mnemonics and constants defined in the program.

The fields in a statement must be separated by at least one space or tab character. If you want, you can use more spaces and tabs, but the assembler ignores them. It is good programming practice to use blank lines and spaces to improve the readability of assembly language programs. As a result, you rarely see in this book a statement containing all four fields in a single line. In particular, we almost always write labels on a separate line unless doing so destroys the program structure. Thus, our first example statement would be written as

```
repeat:
     inc     result
```

9.3 Data Allocation

In high-level languages, allocation of storage space for variables is done indirectly by specifying the data types of each variable used in the program. For example, in C the following declarations allocate different amounts of storage space for each variable.

```
char    response;       /* 1 byte is allocated   */
int     value;          /* 2 bytes are allocated */
float   total;          /* 4 bytes are allocated */
double  average_value;  /* 8 bytes are allocated */
```

These variable declarations not only specify the amount of storage required, but also indicate how the stored bit pattern should be interpreted. As an example, consider the following two statements in C:

```
unsigned   value_1;
int        value_2;
```

Both variables will have two bytes reserved for storage. However, the bit pattern stored in them would be interpreted differently. For instance, the bit pattern (8DB9H)

```
1000 1101 1011 1001
```

stored in `value_1` is interpreted as representing 36,281, whereas the same bit pattern at `value_2` would be interpreted as −29,255.

In assembly language, allocation of storage space is done by the `define` assembler directive. The define directive can be used to reserve and initialize one or more bytes. However, no interpretation is attached to the contents of these bytes. It is entirely up to the program to interpret the bit pattern.

The general format of a storage allocation statement is

```
[variable-name] define-directive initial-value [,initial-value],...
```

The square brackets indicate optional items. The `variable-name` is used to identify the storage space allocated. The assembler associates an offset value for each variable name defined in the data segment. Note that no colon (:) follows the variable name (unlike a label identifying an executable statement).

The define directive takes one of the five basic forms:

```
DB    Define Byte         ;allocates 1 byte
DW    Define Word         ;allocates 2 bytes
DD    Define Doubleword   ;allocates 4 bytes
DQ    Define Quadword     ;allocates 8 bytes
DT    Define Ten Bytes    ;allocates 10 bytes
```

Let us look at some examples now. The statement

```
sorted    DB    'y'
```

allocates a single byte of storage and initializes to character `y`. Our assembly language program can refer to this data location by its name `sorted`. If you just want to reserve storage space without initialization, you can write

```
sorted    DB    ?
```

You can also use numbers to initialize. For example,

```
sorted    DB    79H
```

or

```
sorted    DB    1111001B
```

is equivalent to

```
sorted    DB    'y'
```

Note that the ASCII value for `y` is 79H. We use the suffix "H" to indicate a hexadecimal number and "B" for a binary number. The following data definition statement allocates two bytes of contiguous storage and initializes to 25159:

```
value    DW    25159
```

The decimal value 25159 is automatically converted to its 16-bit binary equivalent (6247H). Since the Pentium uses little-endian byte ordering, this 16-bit number is stored in memory as

```
 address:   x    x+1
contents:   47   62
```

You can also use negative values, as in the following example:

```
balance    DW    -29255
```

Since 2's complement representation is used to store negative values, −29,255 is converted to 8DB9H and is stored as

```
 address:   x    x+1
contents:   B9   8D
```

The statement

```
total    DD    542803535
```

would allocate four contiguous bytes of memory and initialize it to 542803535 (205A864FH), as shown below:

```
 address:   x      x+1    x+2    x+3
contents:   4F     86     5A     20
```

9.3.1 Range of Numeric Operands

The numeric operand of define directives can take both signed and unsigned numbers. The valid range depends on the number of bytes allocated. The following table shows the valid range for the numeric operands:

Directive	Valid range
DB	-128 to 255 (i.e., -2^7 to $2^8 - 1$)
DW	$-32{,}768$ to 65,535 (i.e., -2^{15} to $2^{16} - 1$)
DD	$-2{,}147{,}483{,}648$ to 4,294,967,295 (i.e., -2^{31} to $2^{32} - 1$) or a short floating-point number (32 bits)
DQ	-2^{63} to $2^{64} - 1$ or a long floating-point number (64 bits)

Using a constant that is outside the specified range can result either in an assembler error, or in assigning a wrong value. For example, the statement

```
byte1    DB    256
```

causes an assembly time error. In general, the assembler can accept a value in the range -256 to $+255$. However, 8 bits are not sufficient for the values between -256 and -129. Therefore, the assembler converts the number into 2's complement representation using 16 bits and stores the lower byte. For example,

```
byte2    DB    -200   ; stores 38H
```

stores 38H because the 2's complement representation of -200 is FF38H.

Similarly, the statement

```
word1    DW    -60000  ; stores 15A0H
```

assigns 15A0H because -60000 is outside the range of signed numbers that can be represented using 16 bits. Therefore, as in the last example, -60000 is converted to its 2's complement equivalent using 32 bits (FFFF15A0H), and the lower word is stored.

Short and long floating-point numbers are represented using 32 or 64 bits, respectively. See Appendix A for details. We can use DD and DQ directives to assign real numbers, as shown in the following examples:

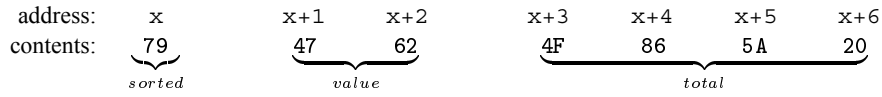
```
float1    DD    1.234
real2     DQ    123.456
```

9.3.2 Multiple Definitions

Assembly language programs typically contain several data definition statements. For example, look at the following assembly language program fragment:

```
sorted    DB    'y'         ; ASCII of y = 79H
value     DW    25159       ; 25159D = 6247H
total     DD    542803535   ; 542803535D = 205A864FH
```

When several data definition statements are used as above, the assembler allocates contiguous memory locations for the variables. The memory layout for the three variables is



Multiple data definitions can be abbreviated. For example, the following sequence of eight DB directives

```
message     DB     'W'
            DB     'E'
            DB     'L'
            DB     'C'
            DB     'O'
            DB     'M'
            DB     'E'
            DB     '!'
```

can be abbreviated as

```
message     DB     'W','E','L','C','O','M','E','!'
```

or even more compactly as

```
message     DB     'WELCOME!'
```

Here is another example showing how abbreviated forms simplify data definitions. The definition

```
message     DB     'B'
            DB     'y'
            DB     'e'
            DB     0DH
            DB     0AH
```

can be written as

```
message     DB     'Bye',0DH,0AH
```

Similar abbreviated forms can be used with the other define directives. For instance, an eight-element `marks` array can be defined and initialized to zero by

```
marks       DW     0
            DW     0
            DW     0
            DW     0
            DW     0
            DW     0
            DW     0
            DW     0
```

which can be abbreviated as

```
marks       DW     0, 0, 0, 0, 0, 0, 0, 0
```

9.3.3 Multiple Initializations

In the previous example, if the class size is 90, it is inconvenient to define the array as described. The `DUP` directive allows multiple initializations to the same value. Using `DUP`, the `marks` array can be defined as

```
marks    DW    8 DUP (0)
```

The `DUP` directive is useful in defining arrays and tables. Here are some examples using the `DUP` directive.

```
table1   DW  10 DUP (?)       ;10 words,uninitialized
name1    DB  30 DUP ('?')     ;30 bytes,each byte
                              ; initialized to ?
name2    DB  30 DUP (?)       ;30 bytes,uninitialized
message  DB  3  DUP ('Bye!') ;12 bytes,initialized
                              ; to Bye!Bye!Bye!
```

The `DUP` directive may also be nested. For example, to allocate storage space containing

```
***??!!!!!***??!!!!!***??!!!!!***??!!!!!
```

we can write

```
stars  DB  4 DUP (3 DUP ('*'), 2 DUP ('?'), 5 DUP ('!'))
```

A two-dimensional 10×5 matrix (10 rows, 5 columns) can be defined as

```
matrix    DW    10 DUP (5 DUP (0))
```

The initialization values of define directives can also be expressions, as shown in the following example:

```
max_marks    DW    7*25
```

This statement is equivalent to

```
max_marks    DW    175
```

The assembler evaluates such expressions at assembly time and assigns the resulting value. Use of expressions to specify initial values is not preferred because it affects the readability of the program. However, there are certain situations where using an expression actually helps clarify the code. In our example, if `max_marks` represents the sum of seven assignment marks where each assignment is marked out of 25 marks, it is preferable to use the expression $7 * 25$ rather than 175.

Table 9.1 Symbol table for the example data segment

Name	Offset
value	0
sum	2
marks	6
message	26
char1	40

Symbol Table

When we allocate storage space using a data definition directive, we usually associate a symbolic name with it for reference. The assembler, during the assembly process, assigns an offset value to each symbolic name. For example, consider the following data definition statements:

```
.DATA
value     DW   0
sum       DD   0
marks     DW   10 DUP (?)
message   DB   'The grade is:',0
char1     DB   ?
```

As we have indicated, the assembler assigns contiguous memory space for the variables. The assembler uses the same ordering of variables that is present in the source code. Thus, finding the offset value of a variable is a simple matter of counting the number of bytes allocated to the variables preceding it. For example, the offset of `marks` is 6 because `value` and `sum` are allocated 2 and 4 bytes, respectively. The symbol table for this data segment is shown in Table 9.1.

9.3.4 Correspondence to C Data Types

The correspondence between the data definition directives and Turbo C data types is shown in Table 9.2. Some examples using the DB, DW, and DD directives are shown in Table 9.3.

Two consecutive apostrophes can be used in a string to specify a single apostrophe, as in

```
message     DB     'John''s'
```

This statement reserves six bytes of storage and initializes it to `John's`. TASM and MASM also allow the use of double quotation marks to specify a string of characters, as in

```
message     DB     ''John's''
```

In a string that is delineated by double quotation marks, two consecutive double quotation marks can be used to stand for a single one. Since double quotation marks are used to specify strings in C (which is different from the sense used here to specify a string of characters), we exclusively use only apostrophes in our programs.

Table 9.2 Correspondence between Turbo C data types and data definition directives

Directive	C data type
DB	char
DW	int, unsigned
DD	float, long
DQ	double
DT	Not used to specify a data type but used to store intermediate float values

Table 9.3 Some example data definition declarations

C declaration	Assembly language data definition
`char ch_1;`	`ch_1 DB ?`
`char string1[30];`	`string1 DB 30 DUP (?)`
`char name1[25] = ''John'';`	`name1 DB 'John',0,20 DUP (?)`
`int value = 50;`	`value DW 50;`
`int array[20];`	`array DW 20 DUP (?)`
`long total = 0;`	`total DD 0`

9.3.5 LABEL Directive

The LABEL directive provides another way to name a memory location *without* actually defining any data. The syntax is

```
name    LABEL    type
```

where `type` specifies the variable type. The standard types BYTE, WORD, DWORD, QWORD, and TBYTE can be used to label 1-, 2-, 4-, 8-, and 10-byte data.

In the example

```
.DATA
count    LABEL    WORD
Lo_count    DB    0
Hi_count    DB    0
.CODE
          . . .
    mov    Lo_count,AL
    mov    Hi_count,CL
          . . .
```

the two bytes of memory `Lo_count` and `Hi_count` can also be referenced as a 16-bit number `count`. We can also individually manipulate the lower and upper halves of `count`.

The LABEL directive is also useful in creating an alias of another data type, as shown in the following example:

```
.DATA
byte_count   LABEL   BYTE
count    DW    0
.CODE
          . . .
    mov    byte_count,CL
          . . .
```

If the LABEL directive is not used in this example, we have to use the PTR directive (discussed in Section 9.5.1) to rewrite the `mov` statement as

```
mov    BYTE PTR count,CL
```

9.4 Where Are the Operands?

Most assembly language instructions require specification of operands. The Pentium assembly language provides several ways to specify the operands. These are called *addressing modes*. This section introduces some of the basic addressing modes required to write simple assembly language programs.

An operand may be in one of the following locations:

- In a register internal to the CPU;
- In the instruction itself;
- In main memory (usually in the data segment);
- At an I/O port (discussed in Chapter 19).

Specification of an operand that is in a register can be done by *register addressing mode*, whereas *immediate addressing mode* refers to specifying an operand that is part of the instruction. We describe two basic addressing modes to specify an operand located in memory. These addressing modes are called memory addressing modes. Chapter 11 discusses the remaining memory addressing modes in detail.

9.4.1 Register Addressing Mode

In this addressing mode, CPU registers contain the operands. For example, the instruction

```
mov    EAX,EBX
```

requires two operands and both are in the CPU registers. The syntax of the move (`mov`) instruction is

```
mov     destination,source
```

The `mov` instruction copies the contents of `source` to `destination`. The contents of `source`, however, are not destroyed. Thus,

```
mov     EAX,EBX
```

copies the contents of the EBX register into the EAX register. In this example, `mov` is operating on 32-bit data. However, we can also use the `mov` instruction to copy 16- and 8-bit data, as shown in the following examples:

```
mov     BX,CX
mov     AL,CL
```

The register addressing mode is the most efficient way of specifying operands for two reasons:

- The operands are in the registers and no memory access is required.
- Instructions using the register mode tend to be shorter, as only 3 bits are needed to identify a register. In contrast, we need at least 16 bits to identify a memory location.

As a consequence, good compilers attempt to place frequently accessed data items in registers. As an example, consider the following pseudocode:

```
total := 0
for (i = 1 to 400)
    total = total + marks[i]
end for
```

Even if the compiler allocates memory for variables `i` and `total`, it should move these variables to registers for the duration of the `for` loop. At the end of the loop, we can copy the values of these registers to memory. This arrangement is more efficient than accessing variables `i` and `total` directly from memory during each iteration.

9.4.2 Immediate Addressing Mode

In this addressing mode, data are specified as part of the instruction. As a result, even though the data are in memory, they are located in the code segment, not in the data segment. This addressing mode is typically used to specify a constant, either directly or via the EQU directive (discussed in Section 9.7). In the example

```
mov     AL,75
```

the source operand 75 is specified in the immediate addressing mode and the destination operand is specified in the register addressing mode. Such instructions are said to use mixed mode addressing.

Immediate addressing mode is also faster because the operand is fetched into the instruction queue along with the instruction during the instruction fetch cycle. This prefetch, therefore, reduces the time required to get the operand from memory.

9.4.3 Direct Addressing Mode

Operands specified in a memory addressing mode require access to the main memory, usually to the data segment. As a result, they tend to be slower than either of the last two addressing modes.

Recall that to locate a data item in a data segment, we need to specify two components: the data segment base address and an offset value within the segment. Recall that the offset is sometimes referred to as the *effective address*. The start address of the segment is typically found in the DS register. Thus, various memory addressing modes differ in the way the offset is specified.

In direct addressing mode, the offset is specified directly as part of the instruction. In assembly language programs, the value is usually indicated by the variable name of the data item referenced. The assembler will translate the name to its offset value during the assembly process using the symbol table as described on page 330.

This addressing mode is the simplest of all the memory addressing modes. The examples that follow assume the following data definition statements:

```
response  DB    'Y'         ;reserves one byte and
                           ;initializes with y

table1    DW    20 DUP (0)  ;reserves 40 bytes and
                           ;initializes to 0

name1     DB    'Jim Ray'   ;reserves 7 bytes and
                           ;initializes to Jim Ray
```

Here are some example `mov` instructions:

```
mov   AL,response    ;copies character y into AL register

mov   response,'N'   ;N is written into the byte represented
                     ;by response (Y is lost)

mov   name1,'K'      ;writes K as the first character of
                     ;name1, which now reads Kim Ray

mov   table1,56      ;56 is written in the first two bytes
                     ;of table1, which contains 56 and 19 zeros
```

This last statement is equivalent to `table1[0]=56` in C.

The Pentium instruction set does not allow both operands to be located in memory. As a result, instructions in high-level languages involving more than one variable would require a sequence of assembly language instructions. For example, the C code

```
total_marks = assign_marks + test_marks + exam_marks
```

is translated into a sequence of four directly addressed assembly language instructions:

```
mov     EAX,assign_marks
add     EAX,test_marks
add     EAX,exam_marks
mov     total_marks,EAX
```

Even though we are using variable names in the above examples, the assembler will actually replace these variables by their offset values during the assembly process.

We look at the `add` instruction later in this chapter. For now, it is sufficient to understand that this instruction adds the two operands and stores the result in the first operand (i.e., the EAX register in our `add` instructions).

9.4.4 Indirect Addressing Mode

The direct addressing can be used to access simple variables. The main drawback of this addressing mode is that it is not useful for accessing complex data structures such as arrays and records that are used in high-level languages. For example, it is not useful for accessing the second element of `table1`, as in

```
table1[1] = 99
```

The indirect addressing mode remedies this deficiency. In this addressing mode, offset of the data is in one of the general registers. For this reason, this addressing mode is sometimes called the register indirect addressing mode.

The indirect addressing mode is not required for variables having only a single element (e.g., `response`). But for variables such as `table1` containing several elements, the starting address of the data structure can be loaded into, say, the BX register so that BX can act as a pointer to an element in `table1`. By manipulating the contents of the BX register, we can access different elements of `table1`. Note that we use 16-bit segments in which the offset is 16 bits long (see Chapter 7).

The fact that a register is holding the offset is indicated by enclosing it within square brackets [ ], as in

```
mov     AX,[BX]
```

This is different from

```
mov     AX,BX
```

which implies that the second operand is in BX.

For 16-bit segments, only BX, BP, SI, and DI registers are allowed to hold the offset. For 32-bit segments, all eight 32-bit registers (i.e., EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) can be used. However, even with 16-bit segments, we can use all eight 32-bit registers by using the address size override prefix (more details on page 437). For instance,

```
mov     AX,[ECX]
```

is valid, but not

```
mov     AX,[CX]   ; not valid
```

Loading the Offset

How do we get the starting address of `table1`? A statement like

```
mov    BX,table1
```

will not work because this statement copies the first element of `table1` into the BX register. Remember that the symbolic name `table1` refers to the offset of the first element of `table1`. The `OFFSET` directive should be used whenever the offset of a variable is needed. Thus,

```
mov    BX,OFFSET table1
```

copies the offset of `table1` into the BX register. The following code assigns 100 to the first element and 99 to the second element of `table1`. Note that BX is incremented by 2 because each element of `table1` requires two bytes.

```
mov    BX,OFFSET table1  ; copy address of table1 to BX
mov    [BX],100          ; table1[0] := 100
add    BX,2              ; BX := BX + 2
mov    [BX],99           ; table1[1] := 99
```

Chapter 11 discusses other memory addressing modes that can perform this task in a better way. In summary, we have discussed the following four addressing modes:

Addressing mode	Valid example	Invalid example
Register	`mov EAX,EBX`	`mov AX,EBX`
Immediate	`mov ECX,155`	`mov 155,ECX`
Direct	`mov table1,DX`	`mov response,name1`
Indirect	`mov [BX],EAX`	`mov [BX],[AX]`

Referenced Default Segments

The register indirect addressing mode can be used to specify data items that are located either in the data segment or in the stack segment.

16-Bit Addresses: By default, effective address in registers BX, SI, or DI is taken as the offset value into the data segment (i.e., relative to the DS segment register). On the other hand, if BP and SP registers are used, the offset is used to access a data item from the stack segment (i.e., relative to the SS segment register). As we show in Chapter 10, BP is used in the register indirect addressing mode to access arguments from the stack in procedure calls.

32-Bit Addresses: By default, effective address in registers EAX, EBX, ECX, EDX, ESI, and EDI is relative to the DS data segment. The SS stack segment is used if EBP and ESP registers are used.

In all cases, stack operations such as `push` and `pop` refer to the stack segment. In addition, the destination of string instructions uses the ES data segment register (see Chapter 12 for details on string instructions).

Overriding Default Segments

We can override the default segment association by a segment override prefix. For example,

```
add     AX,SS:[BX]
```

can be used to access a data item from the stack whose offset relative to the SS register is given in the BX register. For an example that uses the SS overriding prefix, see Program 10.6 on page 419. Similarly, the BP register can be used as an offset into the data segment by

```
add     AX,DS:[BP]
```

The CS, ES, FS, and GS segment registers can also be used to override the default association, even though the CS register is not used frequently. For example, the program on page 501 uses the CS overriding prefix. To summarize, the Pentium provides the following segment override prefixes:

Opcode	Segment override prefix
2EH	CS
36H	SS
3EH	DS
26H	ES
64H	FS
65H	GS

We cannot use these override prefixes to affect the default segment association in the following cases:

- Destination of string instructions always uses the ES segment.
- Stack `push` and `pop` instructions always use the SS segment.
- Instruction fetches always use the CS segment.

Another Way to Load Effective Address

We have seen that the `OFFSET` directive can be used to load the effective address. The effective address can also be loaded into a register by the `lea` (load effective address) instruction. The syntax of this instruction is

```
lea     register,source
```

It copies the address of `source` into `register`. Thus,

```
lea     BX,table1
```

can be used instead of the

```
mov     BX,OFFSET table1
```

instruction. The difference is that `lea` computes the offset values at run-time, whereas `mov` with `OFFSET` resolves the offset value at assembly time. For this reason, we try to use the latter whenever possible. However, `lea` offers more flexibility as to the types of `source` operands. For example, we can write

```
lea     BX,array[SI]
```

to load BX with the address of an element of `array` whose index is in the SI register. However,

```
mov     BX,OFFSET array[SI]
```

does not make sense.

9.5 Data Transfer Instructions

We now discuss some of the data transfer instructions supported by the Pentium. Specifically, we describe `mov`, `xchg`, and `xlat` instructions. Other data transfer instructions such as `movsx` and `movzx` are discussed in Chapter 12.

9.5.1 The mov Instruction

We have already introduced the `mov` instruction, which requires two operands and has the syntax

```
mov     destination,source
```

The data are copied from `source` to `destination`, and the `source` operand remains unchanged. Both operands should be of the same size. The `mov` instruction can take one of the following five forms:

```
mov     register,register
```

Restrictions:

- Destination register cannot be CS or (E)IP registers.
- Both registers cannot be segment registers.

```
mov     register,immediate
```

Restriction: Register cannot be a segment register.

```
mov     memory,immediate
mov     register,memory
mov     memory,register
```

There is no move instruction to transfer data from memory to memory, as the Pentium does not allow it. However, as we show in Chapter 12, memory-to-memory data transfer is possible if we use string instructions.

Ambiguous Moves: PTR Directive

Moving an immediate value into memory sometimes causes ambiguity as to the size of the operand. For example, in the statements

```
mov     BX,OFFSET table1
mov     SI,OFFSET name1
mov     [BX],100
mov     [SI],100
```

it is not clear whether a word (2 bytes) or a byte equivalent of 100 is to be written in the memory. The PTR directive can be used to clarify. WORD PTR can be used to identify a word operation and BYTE PTR for a byte operation. Using the PTR directive, we can write

```
mov     WORD PTR [BX],100
mov     BYTE PTR [SI],100
```

WORD and BYTE are called *type specifiers*. Some of the type specifiers available are

Type specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

9.5.2 The xchg Instruction

The `xchg` instruction exchanges 8-, 16-, or 32-bit source and destination operands. The syntax is similar to that of the `mov` instruction. Here are some examples:

```
xchg    EAX,EDX
xchg    response,CL
xchg    total,DX
```

As in the `mov` instruction, both operands cannot be located in memory. Thus,

```
xchg    response,name1    ; illegal
```

is invalid.

The `xchg` instruction is convenient because we do not need a third register to hold a temporary value in order to swap two values. For example, we need three `mov` instructions

```
mov     ECX,EAX
mov     EAX,EDX
mov     EDX,ECX
```

to perform `xchg EAX,EDX`. Thus, `xchg` is the most efficient way to exchange two 8-, 16-, or 32-bit values. This instruction is especially useful in sorting applications. The `xchg` instruction is also useful in implementing semaphores for process synchronization, as well as for swapping the two bytes of 16-bit data to perform conversions between little-endian and big-endian forms, as in the following example:

```
xchg    AL,AH
```

The Pentium provides the `bswap` (byte swap) instruction to perform such conversions on 32-bit data. The format is

```
bswap  32-bit register
```

This instruction works only on the data located in a 32-bit register. It can be used to convert numbers from big-endian to little-endian format and vice versa.

9.5.3 The xlat Instruction

The `xlat` (translate) instruction can be used to perform character translation. For example, it can be used to translate character codes from ASCII to EBCDIC and vice versa. The `xlat` has the form

```
xlatb
```

To use the `xlat` instruction, the BX register must be loaded with the starting address of the translation table and AL must contain an index value into the table. The `xlat` instruction adds contents of the AL to the BX and reads the byte at the resulting address. This byte replaces the index value in the AL register. Since the 8-bit AL register provides the index into the translation table, the number of entries in the table is limited to 256. An application of `xlat` is given in Example 9.8 on page 374.

9.6 Pentium Assembly Language Instructions

This section discusses some of the remaining assembly language instructions. The instructions presented here allow you to write meaningful assembly language programs. Other Pentium instructions are discussed in Chapter 12.

9.6.1 Arithmetic Instructions

The Pentium provides several instructions to perform simple arithmetic operations. In this section, we describe five instructions to perform addition, subtraction, and comparison. Arithmetic instructions update the status flags, shown on page 259, to record the result of the operation. In Chapter 12, we discuss how the arithmetic instructions affect the status flags. That chapter also looks at the Pentium multiplication and division instructions.

Increment/Decrement Instructions

These instructions can be used to either increment or decrement the operands. The `inc` (increment) instruction adds one to its operand, and the `dec` (decrement) instruction subtracts one from its operand. Both instructions take a single operand. The operand can be either in a register or in memory. It does not make sense to use an immediate operand such as `inc 55` or `dec 109`.

The general format of these instructions is

```
inc    destination
dec    destination
```

where `destination` may be an 8-, 16-, or 32-bit operand.

```
inc    BX      ;increment the 16-bit register
dec    DL      ;decrement the 8-bit register
```

Let us assume that BX and DL have 1057H and 5AH, respectively. After executing the above two instructions, BX and DL will have 1058H and 59H, respectively. If the initial values of BX and DL were FFFFH and 00H, after executing the two statements the contents of BX and DL would be changed to 0000H and FFH, respectively.

Consider the following program:

```
.DATA
count    DW    0
value    DB    25

.CODE
      inc    count              ;unambiguous
      dec    value              ;unambiguous
      move   BX,OFFSET count
      inc    [BX]               ;ambiguous
      mov    SI,OFFSET value
      dec    [SI]               ;ambiguous
```

In the above example,

```
inc    count
dec    value
```

are unambiguous because the assembler knows from the definition of `count` and `value` that they are WORD and BYTE operands. However,

```
inc    [BX]
dec    [SI]
```

are ambiguous because BX and SI registers merely point to an object in memory but the actual object type (whether a WORD or BYTE) is not clear. We have to resort to the PTR directive to clarify, as shown below:

```
inc    WORD PTR [BX]
dec    BYTE PTR [SI]
```

Add Instructions

The add instruction can be used to add two 8-, 16-, or 32-bit operands. The syntax is

```
add     destination,source
```

As with the mov instruction, add can also take the five basic forms depending on how the two operands are specified. The semantics of the add instruction are

```
destination = destination + source
```

As a result, destination loses its contents before the execution of add but the contents of source remain unchanged. The examples given in the table below assume the following data definitions:

```
.DATA
value    DB    0F0H
count    DW    3746H
```

Instruction	Before add		After add
	Source	Destination	Destination
add AX,DX	DX = AB62H	AX = 1052H	AX = BBB4H
add BL,CH	BL = 76H	CH = 27H	BL = 9DH
add value,10H	—	value = F0H	value = 00H
add DX,count	count = 3746H	DX = C8B9H	DX = FFFFH

Note: If we are specifying a hex number that begins with A to F, we have to prefix a zero so that the assembler does not think it is a label. This is the reason for using 0F0H instead of F0H in the DB directive to define count.

In general,

```
inc    EAX
```

is preferred to

```
add    EAX,1
```

as the inc version requires less memory space to store the instruction. However, both instructions typically execute at about the same speed.

The second version of the addition instruction is the adc (add with carry) instruction. This instruction has the general format

```
adc    destination, source
```

and performs

```
destination = destination + source + CF
```

The only difference between `add` and `adc` is that the `adc` instruction adds the contents of the carry flag (CF). The `adc` instruction is useful in performing addition of long multiword numbers (i.e., numbers that take more than 32 bits).

The Pentium provides three instructions to manipulate the carry flag, which are useful with instructions like `adc`.

`stc`	set carry flag	sets CF (CF = 1)
`clc`	clear carry flag	clears CF (CF = 0)
`cmc`	complement carry flag	inverts CF value

All three instructions affect only the carry flag and have no effect on the other flags. These three instructions are also useful in conjunction with the rotate instructions we discuss later. An example that uses `stc` and `clc` instructions is given on page 483.

Example 9.1 *An example 64-bit addition.*
Assume that EBX:EAX and EDX:ECX contain two 64-bit numbers. The notation EBX:EAX is used to indicate that the higher-order 32 bits are in EBX and the lower-order 32 bits in EAX. We can perform this 64-bit addition as shown below:

```
add     EAX,ECX     ; add the lower 32 bits
adc     EBX,EDX     ; add the higher 32 bits with carry
```

The 64-bit result of this addition will be in EBX:EAX registers. □

Subtract Instructions

The `sub` (subtract) instruction can be used to subtract two 8-, 16-, or 32-bit numbers. The syntax is

```
sub     destination,source
```

The `source` operand is subtracted from the `destination` operand and the result is placed in the `destination`.

```
destination = destination – source
```

Some examples are given below:

	Before `sub`		After `sub`
Instruction	Source	Destination	Destination
`sub AX,DX`	`DX = AB62H`	`AX = 1052H`	`AX = 64F0H`
`sub BL,CH`	`CH = 27H`	`BL = 76H`	`BL = 4FH`
`sub value,10H`	—	`value = F0H`	`value = E0H`
`sub DX,count`	`count = 3746H`	`DX = C8B9H`	`DX = 9173H`

The second subtract instruction sbb (subtract with borrow) is the adc counterpart. The syntax is

```
sbb    destination, source
```

and performs

```
destination = destination – source – CF
```

The second subtract operation is done only if CF is 1. As with the adc instruction, sbb is useful in performing a subtract operation on numbers that are longer than 32 bits.

Next we look at the negate (neg) instruction. This single-operand instruction

```
neg    destination
```

subtracts the destination operand from 0. Thus, this instruction effectively reverses the sign of an integer and is meaningful only with signed numbers.

```
destination = 0 – destination
```

The neg instruction updates all six status flags. The carry flag is always set except when the operand is zero, in which case it is cleared.

There is a slight problem when negating the smallest number that can be represented. Remember that, with 8 bits, we can represent signed integers in the range −128 to +127. What happens if we try to negate −128, as in the following example:

```
mov    AL,–128
neg    AL
```

Since +128 is out of range, the overflow flag will be set and there will be no change in the operand value (it remains −128). A similar situation arises with 16- and 32-bit operands.

With add and neg instructions, we can implement

```
sub    destination, source
```

as a sequence of

```
neg    source
add    destination, source
```

However, the former is more efficient and convenient. Furthermore, it aids in program readability.

Compare Instruction

The cmp (compare) instruction is used to compare two operands (equal, not equal, and so on). The compare instruction

```
cmp    destination, source
```

subtracts the source operand from the destination operand but does not alter any of the two operands, as shown below:

```
destination – source
```

The flags are updated as if the `sub` operation were performed. The main purpose of the `cmp` instruction is to update the flags so that a subsequent conditional jump instruction can test these flags. Although both `sub` and `cmp` instructions take the same number of clocks in most cases, `cmp` requires one less clock if the destination is memory. This is because `cmp` does not write the result in memory, whereas the `sub` instruction does.

The `cmp` instruction is used in conjunction with conditional jump instructions for decision making. This is the topic of the next section.

9.6.2 Conditional Execution

The Pentium instruction set contains several branching and looping instructions to construct programs that require conditional execution. In this section, we present a subset of these instructions. Other instructions are discussed in Chapter 12.

Unconditional Jump

The unconditional jump instruction `jmp`, as its name implies, tells the processor that the next instruction to be executed is located at the label that is given as part of the instruction. This jump instruction has the form

```
jmp    label
```

where `label` identifies the next instruction to be executed. We can specify the target either *directly* or *indirectly* leading to direct and indirect jumps. The majority of the jumps are of the direct type. Here, we focus our attention on the direct jump instruction. Indirect jumps are discussed in Section 12.3.1.

In the code fragment

```
            . . .
      mov    CX,10
      jmp    CX_init_done
init_CX_20:
      mov    CX,20
CX_init_done:
      mov    AX,CX
repeat1:
      dec    CX
            . . .
      jmp    repeat1
            . . .
```

both the `jmp` instructions directly specify the target. Recall that, in assembly language programs, we only specify the target address by using a label and let the assembler figure out the exact value by using its symbol table.

The instruction

```
jmp   CX_init_done
```

transfers control to an instruction that follows it. This is called the *forward jump*. On the other hand, the instruction

```
jmp    repeat1
```

is a *backward jump*, as the control is transferred to an instruction that precedes the jump instruction.

Relative Address

The address specified in a jump instruction is not the absolute address of the target instruction. Rather, it specifies the relative displacement in bytes between the target instruction and the instruction following the jump instruction.

In order to see why this is so, we have to understand how jumps are executed. Recall that the IP register always points to the next instruction to be executed (see Chapter 7). Thus, after fetching the `jmp` instruction, the IP is automatically advanced to point to the instruction following the `jmp` instruction. Execution of `jmp` involves changing the IP from where it is currently pointing to the target instruction location. This is achieved by adding the difference (i.e., relative displacement) to the IP contents. This works fine because the relative displacement is a signed number: a positive displacement implies a forward jump, and a negative displacement indicates a backward jump.

The specification of the relative address as opposed to the absolute address of the target instruction is appropriate for dynamically relocatable code (i.e., for position-independent code). This way we could move the code to a different location without changing the code.

The code

```
forever:  jmp forever
```

is a valid statement that results in an infinite loop. Incidentally, this is a backward jump.

Where Is the Target?

If the target of a jump instruction is located in the same segment as the jump instruction itself, it is called an *intrasegment jump*; if the target is located in another segment, it is called an *intersegment jump*.

Our previous discussion has assumed an intrasegment jump. In this case, the `jmp` simply performs the following action:

$$\text{IP} = \text{IP} + \text{relative-displacement}.$$

In the case of an intersegment jump, also called the *far jump*, the CS is also changed to point to the target segment, as shown below:

CS = target-segment,
IP = target-offset.

Both target-segment and target-offset are specified directly in the instruction. Thus, for 16-bit segments, the instruction encoding for the intersegment jump takes five bytes: one byte for the specification of the opcode, two bytes for the target-segment, and another two bytes for the target-offset specification.

The majority of jumps are of the intrasegment type. Therefore, the Pentium provides two ways to specify intrasegment jumps depending on the distance of the target location, that is, depending on the value of the relative displacement.

If the relative displacement, which is a signed number, can fit in a byte, a jump instruction can be encoded by using just two bytes: one byte for the opcode and the other for the relative displacement. This means that the relative displacement should be within −128 to +127 (the range of a signed 8-bit number). This form is called the *short jump*.

If the target is outside this range, two or four bytes are used to specify the relative displacement. A two-byte displacement is used for 16-bit segments, and a four-byte displacement for 32-bit segments. As a result, the jump instruction requires either three or five bytes to encode in the machine language. This form is called the *near jump*.

If we want to use the short jump form, we can inform the assembler by using the operator `SHORT`, as shown below:

```
jmp     SHORT CX_init_done
```

The question that naturally arises at this point is: What if the target is not within −128 or +127 bytes? The assembler will inform us with an error message that the target can't be reached with a short jump.

In fact, specification of `SHORT` for the jump instruction

```
jmp     SHORT repeat1
```

on page 345 is redundant, as the assembler can automatically select the SHORT jump, if appropriate, for all backward jumps. However, for forward jumps, the assembler needs our help. This is because the assembler does not know the relative displacement of the target when it must decide whether to use the short form. Therefore, when appropriate, we use the `SHORT` operator for forward jumps.

Example 9.2 *Example encodings of short and near jumps.*
Figure 9.1 shows some example encodings for short and near jump instructions. This listing consists of four columns:

```
                        . . .
 8 0005  EB 0C              jmp    SHORT CX_init_done
 9 0007  B9 000A            mov    CX,10
10 000A  EB 07 90           jmp    CX_init_done
11                        init_CX_20:
12 000D  B9 0014            mov    CX,20
13 0010  E9 00D0            jmp    near_jump
14                        CX_init_done:
15 0013  8B C1              mov    AX,CX
16                        repeat1:
17 0015  49                 dec    CX
18 0016  EB FD              jmp    repeat1
                        . . .
                        . . .
84 00DB  EB 03              jmp    SHORT short_jump
85 00DD  B9 FF00            mov    CX, 0FF00H
86                        short_jump:
87 00E0  BA 0020            mov    DX, 20H
88                        near_jump:
89 00E3  E9 FF27            jmp    init_CX_20
                        . . .
```

Figure 9.1 Example encoding of jump instructions.

Column 1	Line number;
Column 2	Address in hex;
Column 3	Machine language encoding;
Column 4	Assembly language statement.

For example, for the first `jmp` statement, the line number is 8, its address is 0005H, and the `jmp` instruction is encoded as `EB0C`. Next we discuss the machine language encoding of the jump instruction.

The forward short jump on line 8 is encoded in the machine language as `EB 0C`, where `EB` represents the opcode for the short jump. The relative offset to target `CX_init_done` is `0CH`. From the code, you can see that this is the difference between the address of the target (address 0013H) and the instruction on line 9 (address 0007H). Another example of a forward short jump is given on line 84.

The backward instruction on line 18 also uses the short jump form. In this case, the assembler can decide whether the short or near jump is appropriate. The relative offset is given by FDH (= −3D), which is the offset from the instruction at address 18H to `repeat1` at 15H.

For near jumps, the opcode is E9H, and the relative offset is a 16-bit signed integer. The relative offset of the forward near jump on line 13 is 00D0H, which is equal to 00E3H − 0013H. The relative offset of the backward near jump on line 89 is given by 000DH − 00E6H = FF27H, which is equal to −217D.

The jump instruction encoding on line 10 requires some explanation. Since this is a forward jump and we have not specified that it could be a short jump, the assembler reserves three bytes for a near jump (the worst case scenario). At the time of actual encoding, the assembler knows the target location and therefore uses the short jump version. Thus, `EB 07` represents the encoding, and the third byte is not used and contains a `nop` (no operation) instruction (opcode = 90H). □

Conditional Jumps

In conditional jump instructions, program execution is transferred to the target instruction only when the specified condition is satisfied. The general format is

```
j<cond>   label
```

where `<cond>` identifies the condition under which the target instruction at `label` should be executed. Usually, the condition being tested is the result of the last arithmetic/logic operation. For example, the code

```
read_char:
      mov    DL,0
           . . .
      (code for reading a character into AL)
           . . .
      cmp    AL,0DH      ;compare the character to CR
      je     CR_received ;if equal, jump to CR_received
      inc    CL          ;otherwise, increment CL and
      jmp    read_char   ;go back to read another
                         ; character from keyboard
CR_received:
      mov    DL,AL
          . . .
```

reads characters from the keyboard until the carriage return (CR) key is pressed. The character count is maintained in the CL register. The two instructions

```
cmp    AL,0DH       ;0DH is ASCII for carriage return
je     CR_received  ;je stands for jump on equal
```

perform the required conditional execution. How does the processor remember the result of the `cmp` operation when it is executing the `je` instruction? One of the purposes of the flags register is to provide such short-term memory between instructions. Let us look at the actions taken by the processor in executing these two instructions.

Remember that the `cmp` instruction subtracts 0DH from the contents of the AL register. Although the result is not saved anywhere, the operation sets the zero flag (ZF = 1) if the two operands are the same. If not, ZF = 0. The ZF retains this value until another instruction that affects the ZF is executed. Note that not all instructions affect all the flags. In particular, the `mov` instruction does not affect any of the flags.

Thus, when the `je` instruction is executed, the processor checks the ZF and program execution jumps to the labeled instruction if ZF = 1. To cause the jump, the Pentium loads the (E)IP register with the target instruction address. Recall that the (E)IP register points to the next instruction to be executed. Therefore, when the character read is CR, instead of fetching the instruction

```
inc    CL
```

it will fetch the

```
mov    DL,AL
```

instruction.

Here are some of the conditions tested by conditional jump instructions:

```
je       jump if equal
jg       jump if greater
jl       jump if less
jge      jump if greater or equal
jle      jump if less than or equal
jne      jump if not equal
```

Conditional jumps can also test the values of flags. Some examples are as follows:

```
jz       jump if zero (i.e., if ZF = 1)
jnz      jump if not zero (i.e., if ZF = 0)
jc       jump if carry (i.e., if CF = 1)
jnc      jump if not carry (i.e., if CF = 0)
```

Note that `je` is synonymous with the `jz` instruction. Similarly, `jne` is synonymous with the `jnz` instruction.

As an example, consider the following code:

```
go_back:
      inc    AL
           . . .
           . . .
      cmp    AL,BL
      statement-1
      mov    BL,77H
```

Table 9.4 shows the actions taken depending on `statement-1`.

The conditional jump instructions we have discussed so far treat the operands as signed numbers. There is another set of conditional jump instructions for operands that are unsigned numbers. But until these instructions are discussed in Chapter 12, the conditional jump instructions introduced here are sufficient to write simple assembly language programs.

Table 9.4 Some examples of conditional jump instructions

statement-1	AL	BL	Action taken
`je go_back`	56H	56H	Program control is transferred to `inc AL`
`jg go_back`	56H	55H	Program control is transferred to `inc AL`
`jg go_back` `jl go_back`	56H	56H	No jump; executes the next instruction `mov BL,77H`
`jle go_back` `jge go_back`	56H	56H	Program control is transferred to `inc AL`

`jne go_back`
`jg go_back`
`jge go_back`
	27H	26H	Program control is transferred to `inc AL`

A Note on Conditional Jumps

All conditional jump instructions are encoded into the machine language using only 2 bytes (like the short jump instruction). As a consequence, all these jumps should be short jumps. That is, the target instruction of a conditional jump must be 128 bytes before or 127 bytes after the instruction following the conditional jump instruction itself.

What if the target is outside this range?
If the target is not reachable by using a short jump, we can use the following trick to overcome this limitation. In the instruction sequence

```
            . . .
target:
            . . .
      cmp    AX,BX
      je     target  ; target is not a short jump
      mov    CX,10
            . . .
```

if `target` is not reachable by short jump, it should be replaced by

```
            . . .
target:
            . . .
      cmp    AX,BX
      jne    skip1   ; skip1 is a short jump
```

```
        jmp     target
skip1:
        mov     CX,10
                . . .
```

We have negated the test condition (`je` becomes `jne`) and used an unconditional jump to transfer control to the target. Recall that `jmp` has *short* as well as *near* versions.

9.6.3 Iteration Instructions

Iteration can be implemented with jump instructions. For example, the following code can be used to execute `<loop body>` 50 times.

```
        mov     CL,50
repeat1:
              . . .
        <loop body>
              . . .
        dec     CL
        jnz     repeat1  ;jumps back to repeat1
              . . .      ; as long as dec  CL does
              . . .      ; not result in CL = 0
```

The Pentium, however, provides a group of `loop` instructions to support iteration. The syntax of the basic `loop` instruction is

```
loop    target
```

where `target` is a label that identifies the target instruction of the jump (`repeat1` in our example).

This instruction assumes that the CX register contains the loop count. When executing the `loop` instruction, it decrements the CX register and jumps to the `target` instruction if CX $\neq$ 0. Using this instruction, we can write the previous example as

```
            mov     CX,50
repeat:
               . . .
            <loop body>
               . . .
            loop    repeat
loop_exit:
               . . .
```

A problem with the above code is that if we set CX to 0, the loop iterates 2^{16} times, not zero times. This is not what we expect!

The instruction `jcxz` provides a remedy for this situation by testing the CX register. The syntax of this instruction is

```
jcxz    target
```

which tests the CX register and if it is zero, transfers control to the target instruction. Thus, it is equivalent to

```
cmp     CX,0
jz      target
```

except that `jcxz` does not affect any of the flags, whereas the `cmp/jz` combination affects the status flags. If the operand size is 32 bits, we can use the `jecxz` instruction instead of `jcxz`. Both instructions, however, use the same opcode E3H. The operand size determines the register—CX or ECX—used.

By using this instruction, the previous example can be written as

```
          mov    CX,50
          jcxz    loop_exit
repeat:
               . . .
            <loop body>
               . . .
          loop   repeat
loop_exit:
               . . .
```

Notes on Execution Times:

1. The functionality of the `loop` instruction can be replaced by

   ```
   dec     CX
   jnz     target
   ```

   Surprisingly, the `loop` instruction is slower than the corresponding `dec/jnz` instruction pair. The `loop` instruction takes five or six clocks depending on whether the jump is taken. The `dec/jnz` instruction pair takes only two clocks. Of course, the `loop` instruction is better for program readability.
2. Similarly, the `jcxz` instruction takes five or six clocks, whereas the equivalent

   ```
   cmp     CX,0
   jz      target
   ```

   takes only two clocks. Thus, for code optimization, these complex instructions should be avoided.

The Pentium also has two conditional loop instructions: `loope/loopz` and `loone/loopnz`. These instructions are similar to the unconditional `loop` instruction, except these instructions loop back based on the value of the zero flag. The format of the three loop instructions along with the action taken is shown below.

Mnemonic		Meaning	Action
`loop`	`target`	Loop	CX = CX − 1 if CX ≠ 0 jump to target
`loope` `loopz`	`target` `target`	Loop while equal Loop while zero	CX = CX − 1 if (CX ≠ 0 and ZF = 1) jump to target
`loopne` `loopnz`	`target` `target`	Loop while not equal Loop while zero	CX = CX − 1 if (CX ≠ 0 and ZF = 0) jump to target

9.6.4 Logical Instructions

The Pentium instruction set provides five logical instructions: `and`, `or`, `xor`, `test`, and `not`. The syntax of these instructions is

```
and    destination,source
or     destination,source
xor    destination,source
test   destination,source
not    destination
```

The `and`, `or`, and `xor` are binary operators and perform bitwise `and`, `or`, and `xor` logical operations. The `not` is a unary operator that performs bitwise complement operations. We discuss the need for the `test` instruction later.

The binary logical operations set the destination by performing the specified bitwise logical operation on the source and destination operands. The logical `not` operation simply flips the bits: a 1 in input becomes a 0 in the output, and vice versa.

Here are some examples explaining their operation (all numbers are expressed in binary):

		`and AL,BL`	`or AL,BL`	`xor AL,BL`	`not AL`
AL	BL	AL	AL	AL	AL
1010 1110	1111 0000	1010 0000	1111 1110	0101 1110	0101 0001
0110 0011	1001 1100	0000 0000	1111 1111	1111 1111	1001 1100
1100 0110	0000 0011	0000 0010	1100 0111	1100 0101	0011 1001
1111 0000	0000 1111	0000 0000	1111 1111	1111 1111	0000 1111

Logical instructions also set some of the flags and therefore can be used in conjunction with conditional jump instructions to implement decision making in assembly language programs. We discuss this topic in Chapter 12.

Use of Logical Instructions

Logical instructions are typically used to implement compound logical expressions and bitwise logical operations of high-level languages. In addition, logical operators are useful in bit manipulations. Here we give a brief review of their bit manipulation capabilities. We discuss their use in high-level languages in Chapter 12.

The `and` instruction is useful to clear one or more bits. For example,

```
and     AX,3FFFH
```

clears the two most significant bits of the AX register. The `and` operation can also be used to isolate one or more bits of a byte, word, or doubleword. We present an example of this use later.

The `or` instruction is useful to set one or more bits. For example,

```
or     AX,8000H
```

sets the most significant bit of the AX register while leaving the remaining 15 bits unaltered. Another example use of the `or` instruction is given on page 509.

We can use the `xor` instruction to toggle one or more bits. The statement

```
xor     AX,5555H
```

toggles the even bits of the AX register. Another use of the `xor` instruction is to initialize registers to 0. We can, of course, do this by

```
mov     AX,0
```

but the same result can be achieved by

```
xor     AX,AX
```

This works no matter what is in the AX register. These two instructions, however, are not exactly equivalent. The `xor` instruction affects flags, whereas the `mov` instruction does not. Note that we can also use the `sub` instruction to do the same. All three instructions take one clock cycle to execute, even though the `mov` instruction requires more bytes to encode the instruction.

In the following example, we test the least significant bit of the data in the AL register, and the program control is transferred to the appropriate code depending on the value of this bit.

```
        . . .
        . . .
and     AL,01H
je      bit_is_zero
<code to be executed
```

```
        when the bit is one>
      jmp    skip1
bit_is_zero:
      <code to be executed
       when the bit is zero>
skip1:
      <rest of the code>
```

To understand how the jump is effective in this example, let us assume that AL = 10101110B. The instruction

```
and    AL,01H
```

makes the result 0 and stores it in the AL register. At the same time, the logical operation also sets the zero flag (i.e., ZF = 1) because the result is zero. Recall that `je` tests the ZF and jumps to the target location if ZF = 1. In this example, it is more appropriate to use `jz` (jump if zero). Thus,

```
jz    bit_is_zero
```

can replace the

```
je    bit_is_zero
```

instruction. The conditional jump `je` is an alias for `jz`.

Need for the Test Instruction: A problem with using the `and` instruction for testing, as in the last example, is that it modifies the destination operand. For instance, in the last example,

```
and    AL,01H
```

changes the contents of AL to either 0 or 1 depending on whether the least significant bit is 0 or 1, respectively.

To avoid this problem, we can use the `test` instruction. Similar to the `and` instruction, this instruction also performs the logical bitwise **and** operation except that the source and destination operands are not modified in any way. However, `test` sets the flags just as the `and` instruction would. Therefore, we can use

```
test    AL,01H
```

instead of

```
and    AL,01H
```

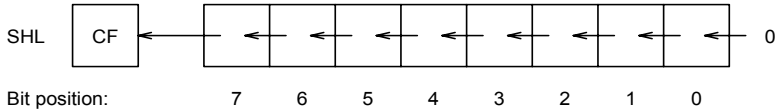
in the last example. Like the `cmp` instruction, `test` takes one clock less to execute than `and` if the destination operand is in memory.

9.6.5 Shift Instructions

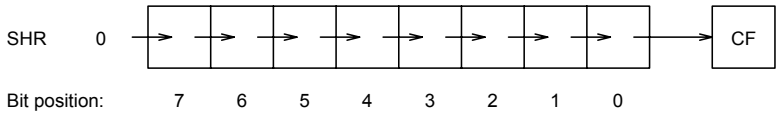
The Pentium provides two types of shift instructions: logical shifts and arithmetic shifts. Within each type, left and right shifts are supported.

Logical Shift Instructions

The `shl` (shift left) instruction can be used to left-shift a destination operand. Each left-shift causes the leftmost bit to move to the carry flag (CF), and the vacated rightmost bit is filled with a zero as shown below:



The `shr` (shift right) instruction works similarly but shifts bits to the right, as shown below:



The general format of these instructions is

```
shl  destination,count      shr  destination,count
shl  destination,CL         shr  destination,CL
```

The destination can be an 8-, 16-, or 32-bit operand stored either in a register or in memory. The second operand specifies the number of bit positions to be shifted. The first format can be used to directly specify the number of bit positions to be shifted. The `count` value can range from 0 to 31. The second format can be used to indirectly specify the shift count, which is assumed to be in the CL register. The CL register contents are not changed by these instructions. In general, the first format is faster!

Even though the shift count can be between 0 and 31, it does not make sense to use count values of zero or greater than 7 (for an 8-bit operand), 15 (for a 16-bit operand), or 31 (for a 32-bit operand). As indicated, the Pentium does not allow the specification of the shift count to be greater than 31. If a greater value is specified, the Pentium takes only the least significant 5 bits of the number as the shift count. Some examples are shown in Table 9.5.

The following code tests the least significant bit of the AL register:

```
            . . .
        shr    AL,1
        jnc    bit_is_zero
        <code to be executed
         when the bit is one>
        jmp    skip1
bit_is_zero:
```

Table 9.5 Some examples of shift instructions (all numbers are expressed in binary)

Instruction	Before shift	After shift	
	AL or AX	AL or AX	CF
`shl AL,1`	1010 1110	0101 1100	1
`shr AL,1`	1010 1110	0101 0111	0
`mov CL,3` `shl AL,CL`	0110 1101	0110 1000	1
`mov CL,5` `shr AX,CL`	1011 1101 0101 1001	0000 0101 1110 1010	1

```
      <code to be executed
       when the bit is zero>
skip1:
      <rest of the code>
            . . .
```

If the AL register contains a 1 in the least significant bit position, this bit will be in the carry flag (CF) after the `shr` instruction has been executed. We can then use a conditional jump instruction that tests the carry flag.

Usage: The shift instructions are useful mainly in these situations:

1. To manipulate bits,
2. To multiply and divide unsigned numbers by a power of two.

Arithmetic Shift Instructions

This set of shift instructions

`sal` (Shift Arithmetic Left)
`sar` (Shift Arithmetic Right)

can be used to shift signed numbers left or right, as shown below:

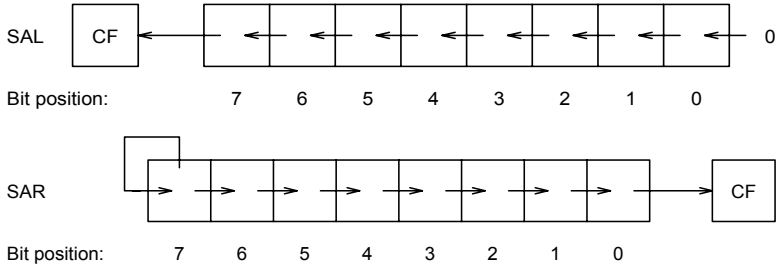


Table 9.6 Doubling of signed numbers

Signed binary number	Decimal value
00001011	+11
00010110	+22
00101100	+44
01011000	+88
11110101	–11
11101010	–22
11010100	–44
10101000	–88

Shifting left by one bit position corresponds to doubling the number, and shifting right by one bit position corresponds to halving it. As with the logical shift instructions, the CL register can be used to specify the count value. The general format is

```
sal     destination,count       sar     destination,count
sal     destination,CL          sar     destination,CL
```

Doubling Signed Numbers

Doubling a signed number by shifting it left by one bit position may appear to cause problems because the leftmost bit is used to represent the sign of the number. It turns out that this is not a problem at all! See the examples presented in Table 9.6 to develop your intuition. The first group presents the doubling effect on positive numbers and the second group shows the doubling effect on negative numbers. In both cases, a 0 replaces the vacated bit. Why isn't shifting out the sign bit causing problems? The reason is that signed numbers are sign-extended to fit a larger than required number of bits. For example, if we want to represent numbers in the range of $+3$ and -4, 3 bits are sufficient to represent this range. If we use a byte to represent the same range, the number is sign-extended by copying the sign bit into the higher-order 5 bits, as shown below:

$$+3 = \overbrace{00000}^{\text{sign bit copied}}011\text{B}$$

$$-3 = \overbrace{11111}^{\text{sign bit copied}}101\text{B}\,.$$

Clearly, doubling a signed number is no different than doubling an unsigned number. Thus, no special left-shift instruction is needed for signed numbers. In fact, `sal` and `shl` are one and the same instruction: `sal` is an alias for `shl`.

Table 9.7 Division of signed numbers by 2

Signed binary number	Decimal value
01011000	+88
00101100	+44
00010110	+22
00001011	+11
10101000	−88
11010100	−44
11101010	−22
11110101	−11

Halving Signed Numbers

Can we also forget about treating the signed numbers differently in halving a number? Unfortunately, we cannot. When we are right-shifting a signed number, the vacated left bit should be replaced by a copy of the sign bit. This rules out the use of `shr` for signed numbers. See the examples presented in Table 9.7. The `sar` instruction does precisely this; the sign bit is copied into the vacated bit on the left.

Remember that the shift right operation performs *integer division*. For example, right-shifting 00001011B (+11D) by a bit results in 00000101B (+5D).

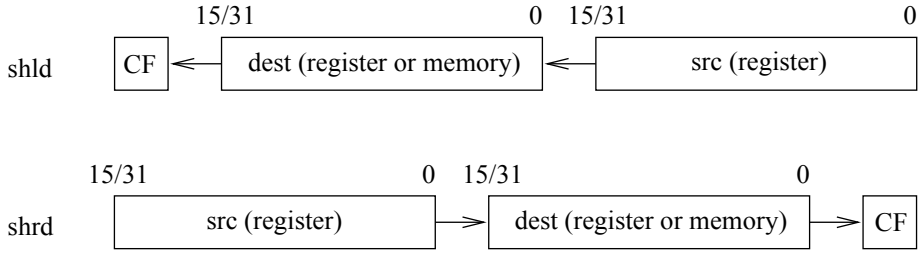
Double-Shift Instructions

The Pentium provides two double-shift instructions for 32-bit and 64-bit shifts. These two instructions operate on either word or doubleword operands and produce a single word or doubleword result, respectively. The double-shift instructions require three operands, as shown below:

```
shld    dest,src,count  ; left shift
shrd    dest,src,count  ; right shift
```

The operands `dest` and `src` can be either a word or doubleword. The `dest` operand can be in a register or memory, but the `src` operand must be in a register. The shift `count` can be specified as in the other shift instructions, either as an immediate value or in the CL register.

A significant difference between shift and double-shift instructions is that the `src` operand supplies the bits in double-shift instructions, as shown below:



Note that the bits shifted out of the `src` operand go into the `dest` operand. However, the `src` operand itself is not modified by the double-shift instructions. Only the `dest` operand is updated appropriately. As in the shift instructions, the last bit shifted out is stored in the carry flag. Later we present an example that demonstrates the use of the double-shift instructions (see Example 9.3 on page 363).

9.6.6 Rotate Instructions

A drawback with the shift instructions is that the bits shifted out are lost. There may be situations where we want to keep these bits. The rotate family of instructions provides this facility. These instructions can be divided into two types: rotate without involving the carry flag (CF), or through the carry flag. We briefly discuss these two types of instructions next.

Rotate Without Carry

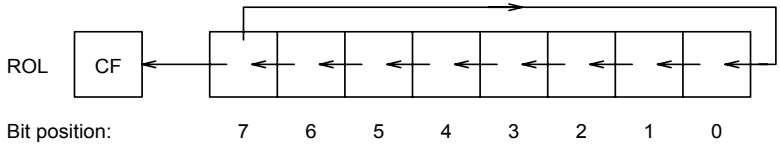
There are two instructions in this group:

`rol` (rotate left)
`ror` (rotate right)

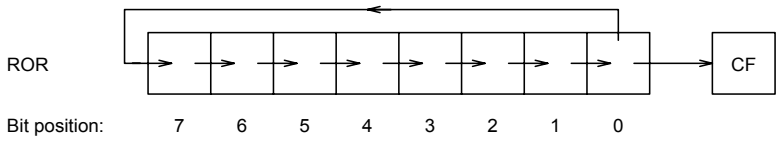
The format of these instructions is similar to the shift instructions and is given below:

```
rol    destination,count      ror    destination,count
rol    destination,CL         ror    destination,CL
```

The `rol` instruction performs left rotation with the bits falling off on the left placed on the right side, as shown below:



The `ror` instruction performs right rotation, as shown below:



In both instructions, the CF will catch the last bit rotated out of the destination. The examples in the following table illustrate the rotate operation:

Instruction	Before execution	After execution	
	AL or AX	AL or AX	CF
`rol AL,1`	1010 1110	0101 1101	1
`ror AL,1`	1010 1110	0101 0111	0
`mov CL,3` `rol AL,CL`	0110 1101	0110 1011	1
`mov CL,5` `ror AX,CL`	1011 1101 0101 1001	1100 1101 1110 1010	1

As a further example, consider encryption of a byte by interchanging the upper and lower nibbles (i.e., 4 bits). This can be done either by

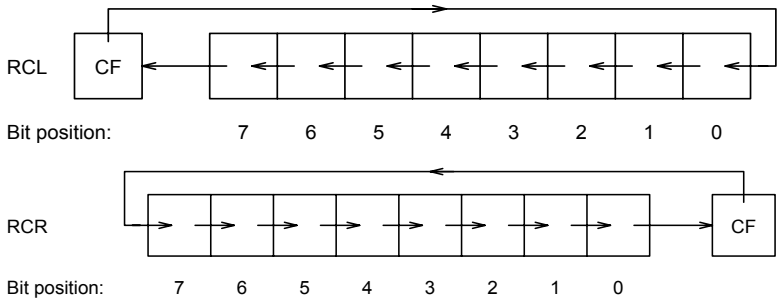
`ror AL,4` or by `rol AL,4`

Rotate Through Carry

The instructions

`rcl` (rotate through carry left)
`rcr` (rotate through carry right)

include the carry flag in the rotation process. That is, the bit that is rotated out at one end goes into the carry flag and the bit that was in the carry flag is moved into the vacated bit, as shown below:



Some examples of `rcl` and `rcr` are given next.

Instruction	Before execution		After execution	
	AL or AX	CF	AL or AX	CF
`rcl AL,1`	1010 1110	0	0101 1100	1
`rcr AL,1`	1010 1110	1	1101 0111	0
`mov CL,3` `rcl AL,CL`	0110 1101	1	0110 1101	1
`mov CL,5` `rcr AX,CL`	1011 1101 0101 1001	0	1001 0101 1110 1010	1

The `rcl` and `rcr` instructions provide flexibility in bit rearranging. Furthermore, these are the only two instructions that take the carry flag bit as an input. This feature is useful in multiword shifts. For example, assume that we want to right-shift the 64-bit number stored in EDX:EAX (the lower 32 bits are in EAX) by one bit position. This can be done by

```
shr     EDX,1
rcr     EAX,1
```

The `shr` instruction moves the least significant bit of EDX to the carry flag. The `rcr` instruction copies this carry flag value into the most significant bit of EAX during the rotation process.

Example 9.3 *Shifting 64-bit numbers.*
As we show in Chapter 12, multiplication and division by a power of two can be performed faster if we use shift operations rather than multiplication or division instructions. Shift instructions operate on operands of size up to 32 bits. What if the operand to be manipulated is bigger?

Since the shift instructions do not involve the carry flag as input, we have two alternatives: either use `rcl` or `rcr` instructions, or use the double-shift instructions for such multiword shifts. As an example, assume that we want to multiply a 64-bit unsigned number by 16. The 64-bit number is assumed to be in the EDX:EAX register pair with EAX holding the least significant 32 bits.

Rotate Version:

```
    mov    CX,4       ; shift by 4 bits
shift_left:
    shl    EAX,1      ; moves leftmost bit of EAX to CF
    rcl    EDX,1      ; CF goes to rightmost bit of EDX
    loop   shift_left
```

Double-Shift Version:

```
shld   EDX,EAX,4 ; EAX is unaffected by shld
shl    EAX,4
```

Similarly, if we want to divide the same number by 16, we can use the following code:

Rotate Version:

```
    mov    CX,4       ; shift by 4 bits
shift_right:
    shr    EDX,1      ; moves rightmost bit of EDX to CF
    rcr    EAX,1      ; CF goes to leftmost bit of EAX
    loop   shift_right
```

Double-Shift Version:

```
shrd   EAX,EDX,4 ; EDX is unaffected by shld
shr    EDX,4
```

□

9.7 Defining Constants

Assemblers provide two directives—EQU and =—to define constants, numeric as well as literal constants. The EQU directive can be used to define numeric constants and strings, whereas the = directive can be used to define numeric constants only.

9.7.1 The EQU Directive

The syntax of the EQU directive is

```
name    EQU    expression
```

which assigns the result of the expression to name. This directive serves the same purpose as #define in C. For example, we can use

```
NUM_OF_STUDENTS    EQU    90
```

to assign 90 to NUM_OF_STUDENTS. It is customary to use capital letters for these names in order to distinguish them from variable names. Once NUM_OF_STUDENTS is defined, we can write

```
                . . .
mov    CX,NUM_OF_STUDENTS
                . . .
cmp    AX,NUM_OF_STUDENTS
                . . .
```

to move 90 into the CX register and to compare AX with 90. Defining constants this way has certain advantages:

1. Such definitions increase program readability. This can be seen by comparing the statement

   ```
   mov     CX,NUM_OF_STUDENTS
   ```

   with

   ```
   mov     CX,90
   ```

   The first statement clearly indicates that we are moving the class size into the CX register.
2. Multiple occurrences of a constant can be changed from a single place. For example, if the class size changes from 90 to 100, we need to change the value in the `EQU` statement only. If we didn't use the `EQU` statement, we would have to scan the source code and make appropriate changes. A risky and error-prone process!

The operand of an `EQU` statement can be an expression that evaluates at assembly time. We can, for example, write

```
NUM_OF_ROWS     EQU     50
NUM_OF_COLS     EQU     10
ARRAY_SIZE      EQU     NUM_OF_ROWS * NUM_OF_COLS
```

to define `ARRAY_SIZE` to be 500.

Strings can be defined in a similar fashion as shown in the following example:

```
JUMP     EQU     jmp
```

Here `JUMP` is an alias for `jmp`. Thus, a statement like

```
JUMP     read_char
```

will be assembled as

```
jmp      read_char
```

The angle brackets (< and >) can be used to define strings that could potentially be interpreted as an expression. For example,

```
ARRAY_SIZE      EQU     <NUM_OF_ROWS * NUM_OF_COLS>
```

forces the assembler to treat

```
NUM_OF_ROWS * NUM_OF_COLS
```

as a string and not evaluate it.

A Restriction: The symbols that have been assigned a value or a string cannot be reassigned another value or string in a given source module. If such redefinitions are required, you should use the = directive, which is discussed next.

9.7.2 The = Directive

The = directive is similar to the EQU directive. The syntax, which is similar to that of the EQU directive, is

```
name = expression
```

There are two key differences:

1. A symbol that is defined by the = directive can be redefined. Therefore, the following code is valid.

```
COUNT = 0
    . . .
    . . .
COUNT = 99
```

2. The = directive cannot be used to assign strings or to redefine keywords or instruction mnemonics. For example,

```
JUMP = jmp
```

is not valid. For these purposes, you should use the EQU directive.

9.8 Macros

Macros provide a means by which a block of text (code, data, etc.) can be represented by a name (called the *macro name*). When the assembler encounters that name later in the program, the block of text associated with the macro name is substituted. This process is referred to as *macro expansion*. In simple terms, macros provide a sophisticated text substitution mechanism.

In assembly language, macros can be defined with MACRO and ENDM directives. The macro text begins with the MACRO directive and ends with the ENDM directive. The macro definition syntax is

```
macro_name    MACRO [parameter1, parameter2, ...]
              macro body
              ENDM
```

In the MACRO directive, the parameters are optional, which is indicated by the square brackets [ ]. macro_name is the name of the macro that, when used later in the program, causes *macro expansion*. To invoke or call a macro, use the macro_name and supply the necessary parameter values. The format is

```
macro_name [argument1, argument2, ...]
```

Example 9.4 *A parameterless macro.*

Here is our first macro example that does not require any parameters. Since using left-shift to multiply by a power of two is more efficient than using multiplication, let us write a macro to do this.

```
multAX_by_16  MACRO
              sal     AX,4
              ENDM
```

The macro code consists of a single `sal` instruction, which will be substituted whenever the macro is called. Now we can invoke this macro by using the macro name `multAX_by_16`, as in the following example:

```
      . . .
mov    AX,27
multAX_by_16
      . . .
```

When the assembler encounters the macro name `multAX_by_16`, it is replaced (i.e., text substituted) by the macro body. Thus, after the macro expansion, the assembler finds the code

```
      . . .
mov    AX,27
sal    AX,4
      . . .
```

□

Macros with Parameters

Just as with procedures, using parameters with macros aids in writing more flexible and useful macros. The number of parameters is limited by how many can fit in a single line. The previous macro always multiplies AX by 16. By using parameters, we can generalize this macro to operate on a byte, word, or doubleword located either in a general-purpose register or memory. The modified macro is

```
mult_by_16      MACRO  operand
                sal    operand,4
                ENDM
```

The parameter `operand` can be any operand that is valid in the `sal` instruction. To multiply a byte in the DL register

```
mult_by_16    DL
```

can be used. This causes the following macro expansion:

```
sal    DL,4
```

Similarly, a memory variable `count` (whether it is a byte, word, or doubleword) can be multiplied by 16 using

```
mult_by_16    count
```

Such a macro call will be expanded as

```
sal     count,4
```

Now, at least superficially, `mult_by_16` looks like any other assembly language instruction, except that we have defined it. These are referred to as *macro-instructions*.

The 8086 processor does not allow specification of the shift count greater than 1 as an immediate value. For this processor, we have to redefine the macro as

```
mult_by_16_8086 MACRO  operand
                sal    operand,1
                sal    operand,1
                sal    operand,1
                sal    operand,1
                ENDM
```

TASM, however, allows you to write immediate shift count values greater than 1 and replaces them by the equivalent set of shift instructions.

Example 9.5 *Memory-to-memory data transfer macro.*
We know that the Pentium does not allow memory-to-memory data transfer. We have to use an intermediate register to facilitate such a data transfer. We can write a macro to perform memory-to-memory data transfers using the basic instructions of the processor. Let us call this macro, which exchanges the values of two memory variables, `Wmxchg` to exchange words of data in memory.

```
Wmxchg   MACRO  operand1, operand2
         xchg   AX,operand1
         xchg   AX,operand2
         xchg   AX,operand1
         ENDM
```

This three-instruction sequence exchanges the memory words `operand1` and `operand2` while leaving AX unaltered. □

9.9 Illustrative Examples

This section presents five examples to illustrate the use of the assembly language instructions discussed in this chapter. In order to follow these examples, you should be able to understand the difference between binary values and character representations. For example, when using a byte to store a number, number 5 is stored as

```
00000101B
```

On the other hand, character 5 is stored as

```
00110101B
```

Character manipulation is easier if you understand this difference and the key characteristics of ASCII, as discussed in Appendix B. Appendix C provides necessary information to assemble these programs with MASM and TASM. The corresponding information for NASM is given in Appendix E.

Example 9.6 *Displays the ASCII value of the input key in binary.*
This program reads a key from the keyboard and displays its ASCII value in binary. It then queries the user as to whether he wants to quit. Depending on the response, the program either requests another character or terminates.

To display the binary value of the ASCII code of the input key, we test each bit starting with the most significant bit (i.e., leftmost bit). We use the `mask` variable, initialized to 80H (i.e., `1000 0000B`), to test only the most significant bit. Let's assume that the character is in the AL register. If its most significant bit is 0, the code

```
test    AL,mask
```

sets ZF. In this case, a 0 is displayed by directing program flow using the `jz` instruction. Otherwise, a 1 is displayed. The `mask` is then divided by two, which is equivalent to right-shifting `mask` by one bit position. Thus, we are ready for testing the second most significant bit. The process is repeated for each bit of the ASCII value. The pseudocode of the program is as follows:

```
main()
read_char:
    display prompt message
    read input character into char
    display output message text
    mask := 80H {AH is used to store mask}
    count := 8 {CX is used to store count}
    repeat
        if ((char AND mask) = 0)
        then
            write 0
        else
            write 1
        end if
        mask := mask/2 {can be done by shr}
        count := count − 1
    until (count = 0)
    display query message
    read response
```

```
        if (response = 'Y')
        then
            goto done
        else
            goto read_char
        end if
    done:
        return
    end main
```

The assembly language program shown in Program 9.1 follows the pseudocode in a straightforward way. Note that the Pentium provides an instruction to perform integer division. However, `shr` is about 17 times faster than the divide instruction to divide a number by 2! More details about the division instructions are given in Chapter 12.

Program 9.1 Conversion of ASCII to binary representation

```
 1:  TITLE   Binary equivalent of characters   BINCHAR.ASM
 2:  COMMENT |
 3:          Objective: To print the binary equivalent of
 4:                     ASCII character code.
 5:              Input: Requests a character from keyboard.
 6:             Output: Prints the ASCII code of the
 7:  |                  input character in binary.
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  char_prompt    DB  'Please input a character: ',0
12:  out_msg1       DB  'The ASCII code of ''',0
13:  out_msg2       DB  ''' in binary is ',0
14:  query_msg      DB  'Do you want to quit (Y/N): ',0
15:
16:  .CODE
17:  INCLUDE io.mac
18:  main    PROC
19:          .STARTUP
20:  read_char:
21:          PutStr  char_prompt  ; request a char. input
22:          GetCh   AL           ; read input character
23:          nwln
24:          PutStr  out_msg1
25:          PutCh   AL
26:          PutStr  out_msg2
27:          mov     AH,80H       ; mask byte = 80H
28:          mov     CX,8         ; loop count to print 8 bits
```

```
print_bit:
        test    AL,AH           ; test does not modify AL
        jz      print_0         ; if tested bit is 0, print it
        PutCh   '1'             ; otherwise, print 1
        jmp     skip1
print_0:
        PutCh   '0'             ; print 0
skip1:
        shr     AH,1            ; right shift mask bit to test
                                ;  next bit of the ASCII code
        loop    print_bit
        nwln
        PutStr  query_msg       ; query user whether to terminate
        GetCh   AL              ; read response
        nwln
        cmp     AL,'Y'          ; if response is not 'Y'
        jne     read_char       ; read another character
done:                           ; otherwise, terminate program
        .EXIT
main    ENDP
        END   main
```

Example 9.7 *Displays the ASCII value of the input key in hexadecimal.*
The objective of this example is to show how numbers can be converted to characters by using character manipulation. This and the next example are similar to the previous one except that the ASCII value is printed in hex. In order to get the least significant hex digit, we have to mask off the upper half of the byte and then perform integer to hex digit conversion. The example shown below assumes that the input character is L, whose ASCII value is 4CH:

$$\text{L} \xrightarrow{\text{ASCII}} 01001100\text{B} \xrightarrow[\text{upper half}]{\text{mask off}} 00001100\text{B} \xrightarrow[\text{to hex}]{\text{convert}} \text{C}.$$

Similarly, to get the most significant hex digit we have to isolate the upper half of the byte and move these four bits to the lower half, as shown below:

$$\text{L} \xrightarrow{\text{ASCII}} 01001100\text{B} \xrightarrow[\text{lower half}]{\text{mask off}} 01000000\text{B} \xrightarrow[\text{4 positions}]{\text{shift right}} 00000100\text{B} \xrightarrow[\text{to hex}]{\text{convert}} 4.$$

Notice that shifting right by four bit positions is equivalent to performing integer division by 16. The pseudocode of the program shown in Program 9.2 is as follows:

```
main()
read_char:
    display prompt message
    read input character into char
    display output message text
    temp := char
    char := char AND F0H {mask off lower half }
    char := char/16 { shift right by 4 positions }
        {The last two steps can be done by shr }
    convert char to hex equivalent and display
    char := temp {restore char }
    char := char AND 0FH {mask off upper half }
    convert char to hex equivalent and display
    display query message
    read response
    if (response = 'Y')
    then
        goto done
    else
        goto read_char
    end if
done:
    return
end main
```

To convert a number between 0 and 15 to its equivalent in hex, we have to divide the process into two parts depending on whether the number is below 10. The conversion using character manipulation can be summarized as follows:

```
if (number ≤ 9)
then
    write (number + '0')
else
    write (number + 'A' − 10)
end if
```

If the number is between 0 and 9, we have to add the ASCII value of character 0 to convert it to its equivalent character. For instance, if the number is 5 (00000101B), it should be converted to character 5, whose ASCII value is 35H (00110101B). Therefore, we have to add 30H, which is the ASCII value of 0. This is done in Program 9.2 by

```
add     AL,'0'
```

on line 34. If the number is between 10 and 15, we have to convert it to a hex digit between A and F. You can verify that the required translation is achieved by

number − 10 + ASCII value of character A

In Program 9.2, this is done by

```
add    AL,'A'-10
```

on line 37.

Program 9.2 Conversion to hexadecimal by character manipulation

```
 1:  TITLE   Hex equivalent of characters   HEX1CHAR.ASM
 2:  COMMENT |
 3:          Objective: To print the hex equivalent of
 4:                     ASCII character code.
 5:              Input: Requests a character from keyboard.
 6:             Output: Prints the ASCII code of the
 7:  |                  input character in hex.
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  char_prompt    DB  'Please input a character: ',0
12:  out_msg1       DB  'The ASCII code of ''',0
13:  out_msg2       DB  ''' in hex is ',0
14:  query_msg      DB  'Do you want to quit (Y/N): ',0
15:
16:  .CODE
17:  .486
18:  INCLUDE io.mac
19:  main    PROC
20:          .STARTUP
21:  read_char:
22:          PutStr  char_prompt  ; request a char. input
23:          GetCh   AL           ; read input character
24:          nwln
25:          PutStr  out_msg1
26:          PutCh   AL
27:          PutStr  out_msg2
28:          mov     AH,AL        ; save input character in AH
29:          shr     AL,4         ; move upper 4 bits to lower half
30:          mov     CX,2         ; loop count - 2 hex digits to print
31:  print_digit:
32:          cmp     AL,9         ; if greater than 9
33:          jg      A_to_F       ; convert to A through F digits
34:          add     AL,'0'       ; otherwise, convert to 0 through 9
35:          jmp     skip
36:  A_to_F:
```

```
37:          add     AL,'A'-10     ; subtract 10 and add 'A'
38:                                ;  to convert to A through F
39:  skip:
40:          PutCh   AL            ; write the first hex digit
41:          mov     AL,AH         ; restore input character in AL
42:          and     AL,0FH        ; mask off the upper half byte
43:          loop    print_digit
44:          nwln
45:          PutStr  query_msg     ; query user whether to terminate
46:          GetCh   AL            ; read response
47:          nwln
48:          cmp     AL,'Y'        ; if response is not 'Y'
49:          jne     read_char     ; read another character
50:  done:                         ; otherwise, terminate program
51:          .EXIT
52:  main    ENDP
53:          END   main
```

Example 9.8 *Displays ASCII value of the input key in hexadecimal using the xlat instruction.* The objective of this example is to show how the use of `xlat` simplifies the solution of the last example. In this example, we use the `xlat` instruction to convert an integer value in the range between 0 and 15 to its equivalent hex digit. The code is shown in Program 9.3. To use `xlat` we have to construct a translation table, which is done by the following statement (line 17):

```
hex_table    DB    '0123456789ABCDEF'
```

We can then use the integer value as an index into the table. For example, an integer value of 10 points to `A`, which is the equivalent hex digit. In order to use the `xlat` instruction, BX should point to the base of the `hex_table` (line 32) and AL should have an integer value between 0 and 15. The rest of the program is straightforward to follow.

Program 9.3 Conversion to hexadecimal by using the `xlat` instruction

```
 1:  TITLE   Hex equivalent of characters   HEX2CHAR.ASM
 2:  COMMENT |
 3:          Objective: To print the hex equivalent of
 4:                     ASCII character code. Demonstrates
 5:                     the use of xlat instruction.
 6:              Input: Requests a character from keyboard.
 7:             Output: Prints the ASCII code of the
 8:  |                  input character in hex.
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
```

```
12:  char_prompt    DB  'Please input a character: ',0
13:  out_msg1       DB  'The ASCII code of ''',0
14:  out_msg2       DB  ''' in hex is ',0
15:  query_msg      DB  'Do you want to quit (Y/N): ',0
16:  ; translation table: 4-bit binary to hex
17:  hex_table      DB  '0123456789ABCDEF'
18:
19:  .CODE
20:  .486
21:  INCLUDE io.mac
22:  main    PROC
23:          .STARTUP
24:  read_char:
25:          PutStr  char_prompt  ; request a char. input
26:          GetCh   AL           ; read input character
27:          nwln
28:          PutStr  out_msg1
29:          PutCh   AL
30:          PutStr  out_msg2
31:          mov     AH,AL        ; save input character in AH
32:          mov     BX,OFFSET hex_table  ; BX := translation table
33:          shr     AL,4         ; move upper 4 bits to lower half
34:          xlatb                ; replace AL with hex digit
35:          PutCh   AL           ; write the first hex digit
36:          mov     AL,AH        ; restore input character to AL
37:          and     AL,0FH       ; mask off upper 4 bits
38:          xlatb
39:          PutCh   AL           ; write the second hex digit
40:          nwln
41:          PutStr  query_msg    ; query user whether to terminate
42:          GetCh   AL           ; read response
43:          nwln
44:          cmp     AL,'Y'       ; if response is not 'Y'
45:          jne     read_char    ; read another character
46:  done:                        ; otherwise, terminate program
47:          .EXIT
48:  main    ENDP
49:          END   main
```

Example 9.9 *Conversion of lowercase letters to uppercase.*

This example demonstrates how indirect addressing can be used to access elements of an array. It also illustrates how character manipulation can be used to convert lowercase letters to uppercase. The program receives a character string from the keyboard and converts all lowercase

letters to uppercase and displays the string. Characters other than the lowercase letters are not changed in any way. The pseudocode of Program 9.4 is as follows:

```
main()
    display prompt message
    read input string
    index := 0
    char := string[index]
    while (char ≠ NULL)
        if ((char ≥ 'a') AND (char ≤ 'z'))
        then
            char := char + 'A' – 'a'
        end if
        display char
        index := index + 1
        char := string[index]
    end while
end main
```

You can see from Program 9.4 that the compound condition **if** requires two `cmp` instructions (lines 27 and 29). Also the program uses the BX register in indirect addressing mode and always holds the pointer value of the character to be processed. In Chapter 11, we show a better way of accessing elements of an array. The end of the string is detected by

```
cmp     AL,0        ; check if AL is NULL
je      done
```

and is used to terminate the **while** loop (lines 25 and 26).

Program 9.4 Conversion to uppercase by character manipulation

```
 1:  TITLE   uppercase conversion of characters   TOUPPER.ASM
 2:  COMMENT |
 3:          Objective: To convert lowercase letters to
 4:                     corresponding uppercase letters.
 5:              Input: Requests a character string from keyboard.
 6:  |          Output: Prints the input string in uppercase.
 7:  .MODEL SMALL
 8:  .STACK 100H
 9:  .DATA
10:  name_prompt    DB  'Please type your name: ',0
11:  out_msg        DB  'Your name in capitals is: ',0
12:  in_name        DB  31 DUP (?)
13:
14:  .CODE
```

```
15:  INCLUDE io.mac
16:  main    PROC
17:          .STARTUP
18:          PutStr  name_prompt  ; request character string
19:          GetStr  in_name,31   ; read input character string
20:          nwln
21:          PutStr  out_msg
22:          mov     BX,OFFSET in_name  ; BX := address of in_name
23:  process_char:
24:          mov     AL,[BX]      ; move the char. to AL
25:          cmp     AL,0         ; if it is the NULL character
26:          je      done         ;  conversion done
27:          cmp     AL,'a'         ; if (char < 'a')
28:          jl      not_lower_case ; not a lowercase letter
29:          cmp     AL,'z'         ; if (char > 'z')
30:          jg      not_lower_case ; not a lowercase letter
31:  lower_case:
32:          add     AL,'A'-'a'   ; convert to uppercase
33:  not_lower_case:
34:          PutCh   AL           ; write the character
35:          inc     BX           ; BX points to next char.
36:          jmp     process_char ; go back to process next char.
37:          nwln
38:  done:
39:          .EXIT
40:  main    ENDP
41:          END   main
```

Example 9.10 *Sum of the individual digits of a number.*

This example shows how decimal digits can be converted from their character representations to equivalent binary. The program receives a number (maximum 10 digits) and displays the sum of the individual digits of the input number. For example, if the input number is 45213, the program displays $4 + 5 + 2 + 1 + 3 = 15$. Since ASCII assigns a special set of contiguous values to the 10-digit characters, it is straightforward to get their numerical value (as discussed in Appendix B). All we have to do is to mask off the upper half of the byte. In Program 9.5 this is done by the `and` instruction

```
and    AL,0FH
```

on line 28.

Alternatively, we can also subtract the character code for 0

```
sub    AL,'0'
```

For the sake of brevity, we leave writing the pseudocode of Program 9.5 as an exercise.

Program 9.5 Sum of individual digits of a number

```
 1:  TITLE   Add individual digits of a number   ADDIGITS.ASM
 2:  COMMENT |
 3:          Objective: To find the sum of individual digits of
 4:                     a given number. Shows character to binary
 5:                     conversion of digits.
 6:              Input: Requests a number from keyboard.
 7:  |          Output: Prints the sum of the individual digits.
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  number_prompt  DB  'Please type a number (<11 digits): ',0
12:  out_msg        DB  'The sum of individual digits is: ',0
13:  number         DB  11 DUP (?)
14:
15:  .CODE
16:  INCLUDE io.mac
17:  main    PROC
18:          .STARTUP
19:          PutStr  number_prompt  ; request an input number
20:          GetStr  number,11    ; read input number as a string
21:          nwln
22:          mov     BX,OFFSET number  ; BX := address of number
23:          sub     DX,DX        ; DX := 0 -- DL keeps the sum
24:  repeat_add:
25:          mov     AL,[BX]      ; move the digit to AL
26:          cmp     AL,0         ; if it is the NULL character
27:          je      done         ;  sum is done
28:          and     AL,0FH       ; mask off the upper 4 bits
29:          add     DL,AL        ; add the digit to sum
30:          inc     BX           ; increment BX to point to next digit
31:          jmp     repeat_add   ;  and jump back
32:  done:
33:          PutStr  out_msg
34:          PutInt  DX           ; write sum
35:          nwln
36:          .EXIT
37:  main    ENDP
38:          END   main
```

9.10 Summary

In this chapter, we presented the basics of Pentium assembly language programming. We discussed three types of assembly language statements:

1. Executable statements instruct the CPU as to what to do;
2. Assembler directives facilitate the assembly process by giving information to the assembler;
3. Macros provide a sophisticated text substitution facility.

We have discussed assembler directives to allocate storage space for data variables and to define numeric and string constants. Most assembly language instructions require one or more operands. Specification of operands is referred to as the addressing mode. We have described four addressing modes to specify the operands. In a later chapter, we discuss the remaining addressing modes supported by the Pentium.

We have presented an overview of the Pentium instruction set. Although we have discussed in detail the data transfer instructions, we have provided only an overview of the remaining instructions of the Pentium instruction set. The next three chapters present a detailed discussion of these instructions.

In the last section, we have given several examples to illustrate the basics of the assembly language programs. These examples follow the structure of the standalone assembly language program described in Appendix C. The information included in this chapter gives you enough knowledge to write reasonable assembly programs. The exercises of this chapter are designed so that you can work on them using only the material presented in this chapter.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- = directive
- Addressing modes
- Assembler directives
- Backward jump
- Conditional jumps
- Data allocation
- Default segments
- Direct addressing mode
- Direct jumps
- Effective address
- EQU directive
- Executable instructions
- Far jump
- Forward jump
- Immediate addressing mode
- Intersegment jump
- Intrasegment jump
- MACRO directive
- Macro expansion
- Macro instructions
- Macro parameters
- Macros
- Near jump
- Override prefix

- Overriding default segments
- PTR directive
- Register addressing mode
- Relative address
- SHORT directive
- Short jump
- Symbol table
- Type specifier
- Unconditional jump

9.11 Exercises

9–1 Why doesn't the CPU execute assembler directives?

9–2 What is the difference between the following two data definition statements?

```
int1    DB    2 DUP (?)
int2    DW    ?
```

9–3 On page 338, we have stated that

```
mov    BX,OFFSET array[SI]
```

does not make sense. Explain why?

9–4 For each of the following statements, what is the amount of storage space reserved (in bytes)? Also indicate the initialized data. Verify your answers using your assembler.

```
(a) table  DW  100 DUP (-1)
(b) nest1  DB  5 DUP (2 DUP ('%'), 3 DUP ('$'))
(c) nest2  DB  4 DUP (5 DUP (2 DUP (1),3 DUP (2)))
(d) value  DW  -2300
(e) count  DW  40000
(f) msg1   DB  'Finder''s fee is:',0
(g) msg2   DB  'Sorry! Invalid input.',0DH,0AH,0
```

9–5 What are the three assembler directives that allow macro definition? Discuss the differences and similarities among them.

9–6 What are the advantages of allowing parameters in a macro definition?

9–7 What is an addressing mode? Why does Pentium provide several addressing modes?

9–8 We discussed four addressing modes in this chapter. Which addressing mode is the most efficient one? Explain why.

9–9 Can we use the immediate addressing mode in the `inc` instruction? Justify your answer.

9–10 What are the differences between direct and indirect addressing modes?

9–11 In the 16-bit addressing mode, which registers can be used in the indirect addressing mode? Also specify the default segments associated with each register used in this mode.

9–12 Discuss the pros and cons of using the `lea` instruction as opposed to using the `mov` instruction along with the `OFFSET` assembler directive.

9–13 On page 368, we have stated that the `Wmxchg` macro does not alter the contents of the AX. Verify that this is true.

9–14 Use the following data definitions to answer this question:

```
.DATA
num1    DW    100
num2    DB    225
char1   DB    'Y'
num3    DD    0
```

Identify whether the following instructions are legal or illegal. Explain the reason for each illegal instruction:

```
(a) mov    EAX,EBX          (b) mov    EAX,num2
(c) mov    BL,num1          (d) mov    DH,char1
(e) mov    char1,num2       (f) mov    IP,num1
(g) add    75,EAX           (h) cmp    75,EAX
(i) sub    char1,'A'        (j) xchg   AL,num2
(k) xchg   AL,23            (l) inc    num3
```

9–15 Assume that the registers are initialized to

EAX = 12345D, EBX = 9528D
ECX = −1275D, EDX = −3001D

What is the destination operand value (in hex) after executing the following instructions. (*Note:* Assume that the four registers are initialized as shown above for each question.)

```
(a) add    EAX,EBX          (b) sub    AX,CX
(c) and    EAX,EDX          (d) or     BX,AX
(e) not    EDX              (f) shl    BX,2
(g) shl    EAX,CL           (h) shr    BX,2
(i) shr    EAX,CL           (j) sub    CX,BX
(k) add    CX,DX            (l) sub    DX,CX
```

9–16 In each of the following code fragments, state whether `mov AX,10` or `mov BX,1` is executed:

```
(a)                          (b)
    mov    CX,5                  mov    CX,5
    sub    DX,DX                 mov    DX,10
    cmp    DX,CX                 shr    DX,1
    jge    jump1                 cmp    CX,DX
    mov    BX,1                  je     jump1
    jmp    skip1                 mov    BX,1
jump1:                           jmp    skip1
    mov    AX,10           jump1:
skip1:                           mov    AX,10
       . . .                 skip1:
                                      . . .
```

```
(c)
    mov    CX,15BAH
    mov    DX,8244H
    and    DX,CX
    jz     jump1
    mov    BX,1
    jmp    skip1
jump1:
    mov    AX,10
skip1:
       . . .
```

```
(d)
    mov    CX,5
    not    CX
    mov    DX,10
    cmp    CX,DX
    jg     jump1
    mov    BX,1
    jmp    skip1
jump1:
    mov    AX,10
skip1:
       . . .
```

9–17 Describe in one sentence what the following code is accomplishing in terms of number manipulation:

```
(a)
    not    AX
    add    AX,1
```

```
(b)
    not    AX
    inc    AX
```

```
(c)
    sub    AH,AH
    sub    DH,DH
    mov    DL,AL
    add    DX,DX
    add    DX,DX
    add    DX,AX
    add    DX,DX
```

```
(d)
    sub    AH,AH
    sub    DH,DH
    mov    DL,AL
    mov    CL,3
    shl    DX,CL
    shl    AX,1
    add    DX,AX
```

9–18 Do you need to know the initial contents of the AX register in order to determine the contents of the AX register after executing the following code? If so, explain why. Otherwise, find the AX contents.

```
(a)
    mov    DX,AX
    not    AX
    or     AX,DX
```

```
(b)
    mov    DX,AX
    not    AX
    and    AX,DX
```

9–19 The `inc` and `dec` instructions do not affect the carry flag. Explain why it is really not required.

9–20 Suppose the `add` instruction is not available. Show how we can use the `adc` instruction to implement the `add` instruction. Of course, you can use other instructions as well.

9–21 Suppose the `adc` instruction is not available. Show how we can use the `add` instruction to implement the `adc` instruction. Of course, you can use other instructions as well.

9–22 Show how you can implement multiplication by 12 by using four additions. You can use registers for temporary storage.

9–23 What is the use of the `neg` instruction?

9–24 Show how you can implement the `neg` instruction with an `add` instruction.

9–25 The logical `and` operation can be implemented by using only `or` and `not` operations. Show how this can be done. You can use as many `or` and `not` operations as you want. But try to implement by using only three `not` and one `or` operation. Is this question related to the digital logic material covered in Chapter 2?

9–26 The logical `or` operation can be implemented by using only `and` and `not` operations. Show how this can be done. You can use as many `and` and `not` operations as you want. But try to implement by using only three `not` and one `and` operation. As in the last question, cast this question in the digital logic context.

9–27 Explain how `and` and `or` logical operations can be used to "cut and paste" a specific set of bits.

9–28 Suppose the instruction set did not support the `not` instruction. How do you implement it using only `and` and `or` instructions?

9–29 Can we use the logical shift instructions `shl` and `shr` on signed data?

9–30 Can we use the arithmetic shift instructions `sal` and `sar` on unsigned data?

9–31 Give an assembly language program fragment to copy low-order 4 bits from the AL register and higher-order 4 bits from the AH register into the DL register. You should accomplish this using only the logical operations of Pentium.

9–32 Repeat the above exercise using only the shift/rotate operations of the Pentium instruction set.

9–33 Show the assembly language program fragment to complement only the odd bits of the AL register using only the logical operations.

9–34 Repeat the above exercise using only the shift/rotate operations of the Pentium instruction set.

9.12 Programming Exercises

9–P1 The program on page 373, Program 9.2, uses uppercase letters A through F for the hex digits. Modify the program to use the lowercase letters instead.

9–P2 The program on page 374, Program 9.3, uses uppercase letters A through F for the hex digits. Modify the program to use the lowercase letters instead. Since you did the same thing in the last exercise, which do you think is a better program to do changes like this? Explain why.

9–P3 Modify the program of Example 9.6 so that, in response to the query

```
Do you want to quit (Y/N):
```

the program terminates only if the response is Y or y, continues with a request for another character only if the response to the query is N or n, and otherwise repeats the query.

9–P4 Modify the program of Example 9.6 to accept a string and display the binary equivalent of the input string. As in the example, the user should be queried about program termination.

9–P5 Modify the `addigits.asm` program such that it accepts a string from the keyboard consisting of digit and nondigit characters. The program should display the sum of the digits present in the input string. All nondigit characters should be ignored. For example, if the input string is

```
ABC1?5wy76:~2
```

the output of the program should be

```
sum of individual digits is: 21
```

9–P6 Write an assembly language program to encrypt digits as shown below:

input digit:	0 1 2 3 4 5 6 7 8 9;
encrypted digit:	4 6 9 5 0 3 1 8 7 2.

Briefly discuss whether you would use the `xlat` instruction. Your program should accept a string consisting of digit and nondigit characters. The encrypted string should be displayed in which only the digits are affected. Then the user should be queried whether she wants to terminate the program. If the response is either 'y' or 'Y', you should terminate the program; otherwise, you should request another input string from the keyboard.

The encryption scheme given here has the property that when you encrypt an already encrypted string, you get back the original string. Use this property to verify your program.

9–P7 Using only the assembly language instructions discussed so far, write a program to accept a number in hexadecimal form and display the decimal equivalent of the number. The following is a typical interaction of your program (user input is shown in bold):

Please input a positive number in hex (4 digits max.): **A10F**
The decimal equivalent of A10FH is 41231
Do you want to terminate the program (Y/N): **Y**

You should refer to Appendix A for an algorithm to convert from base b to decimal.
Hints:

1. Required multiplication can be done by the `shl` instruction.
2. Once you have converted the hex number into the equivalent in binary using the algorithm of Appendix A, you can use the `PutInt` routine to display the decimal equivalent.

9–P8 Repeat the previous exercise with the following modifications: the input number is given in decimal and the program displays the result of (integer) dividing the input by 4. You should not use the `GetInt` routine to read the input number. Instead, you should read the input as a string using `GetStr`. A typical interaction of the program is (user input is shown in bold):

Please input a positive number (<65,535): **41231**
41231/4 = 10307
Do you want to terminate the program (Y/N): **Y**

Remember that the decimal number is read as a string of digit characters. Therefore, you will have to convert it to binary form to store internally. This conversion requires multiplication by 10 (see Appendix A). We haven't discussed multiplication instruction yet (and you should not use it even if you are familiar with it). But there is a way of doing multiplication by 10 using only the instructions discussed in this chapter. (If you have done the exercises of this chapter, you already know how!)

9–P9 Write a program that reads an input number (given in decimal) between 0 and 65,535 and displays the hexadecimal equivalent. You can read the input using the `GetInt` routine. As with the other programming exercises, you should query the user for program termination.

9–P10 Modify the last program to display the octal equivalent of the input number.

Chapter 10

Procedures and the Stack

Objectives

- To introduce the stack and its implementation in the Pentium;
- To describe stack operations and the use of the stack;
- To present procedures and parameter passing mechanisms;
- To discuss separate assembly of source program modules.

The last chapter gave an introduction to the assembly language programs. Here we discuss how procedures are written in the assembly language. Procedures are important programming constructs that facilitate modular programming. The stack plays an important role in procedure invocation and execution. Section 10.1 introduces the stack concept and the next section discusses how the stack is implemented in the Pentium. Stack operations—push and pop—are discussed in Section 10.3. Section 10.4 discusses stack uses.

After a brief introduction to procedures (Section 10.5), assembly language directives for writing procedures are discussed in Section 10.6. Section 10.7 presents the Pentium instructions for procedure invocation and return. Parameter passing mechanisms are discussed in detail in Section 10.8. The stack plays an important role in parameter passing. Using the stack it is relatively straightforward to pass a variable number of parameters to a procedure (discussed in Section 10.9). The issue of local variable storage in procedures is discussed in Section 10.10.

Although short assembly language programs can be stored in a single file, real application programs are likely to be broken into several files, called modules. The issues involved in writing and assembling multiple source program modules are discussed in Section 10.11. The last section provides a summary of the chapter.

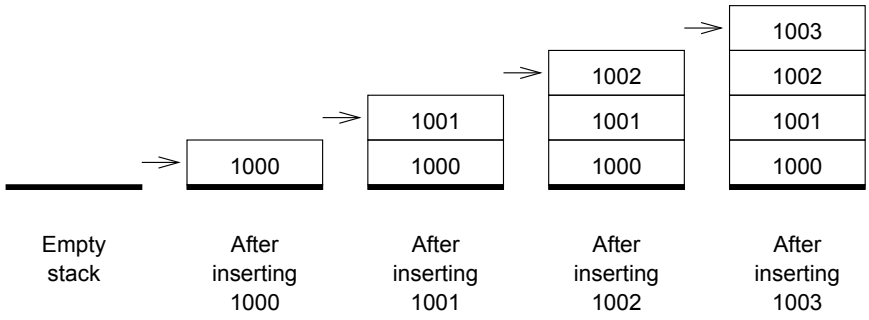


Figure 10.1 An example showing stack growth: Numbers 1000 through 1003 are inserted in ascending order. The arrow points to the top-of-stack.

10.1 What Is a Stack?

Conceptually, a stack is a last-in-first-out (LIFO) data structure. The operation of a stack is analogous to the stack of trays you find in cafeterias. The first tray removed from the stack of trays would be the last tray that had been placed on the stack. There are two operations associated with a stack: insertion and deletion. If we view the stack as a linear array of elements, stack insertion and deletion operations are restricted to one end of the array. Thus, the only element that is directly accessible is the element at the top-of-stack (TOS). In stack terminology, insert and delete operations are referred to as *push* and *pop* operations, respectively.

There is another related data structure, the *queue*. A queue can be considered as a linear array with insertions done at one end of the array and deletions at the other end. Thus, a queue is a first-in-first-out (FIFO) data structure.

As an example of a stack, let us assume that we are inserting numbers 1000 through 1003 into a stack in ascending order. The state of the stack can be visualized as shown in Figure 10.1. The arrow points to the top-of-stack. When the numbers are deleted from the stack, the numbers will come out in the reverse order of insertion. That is, 1003 is removed first, then 1002, and so on. After the deletion of the last number, the stack is said to be in the empty state (see Figure 10.2).

In contrast, a queue maintains the order. Suppose that the numbers 1000 through 1003 are inserted into a queue as in the stack example. When removing the numbers from the queue, the first number to enter the queue would be the one to come out first. Thus, the numbers deleted from the queue would maintain their insertion order.

10.2 Pentium Implementation of the Stack

The memory space reserved in the stack segment is used to implement the stack. The registers SS and (E)SP are used to implement the Pentium stack. If 16-bit address size segments are used as we do in this book, SP is used as the stack pointer, and ESP is used for 32-bit address size segments. In the rest of the chapter, our focus is on 16-bit segments.

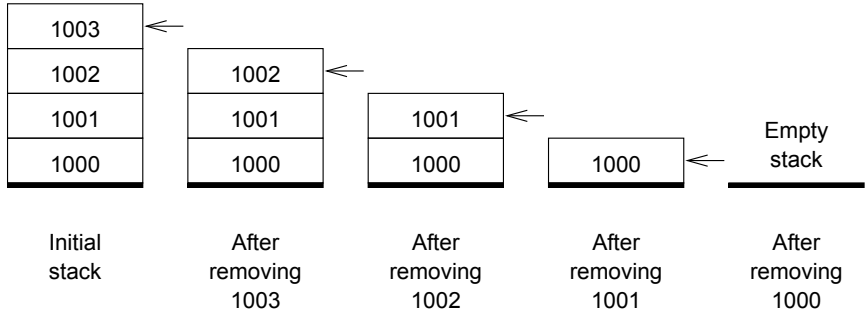


Figure 10.2 Deletion of data items from the stack: The arrow points to the top-of-stack.

The top-of-stack, which points to the last item inserted into the stack, is indicated by SS:SP, with the SS register pointing to the beginning of the stack segment, and the SP register giving the offset value of the last item inserted.

The key Pentium stack implementation characteristics are as follows:

- Only words (i.e., 16-bit data) or doublewords (i.e., 32-bit data) are saved on the stack, never a single byte.
- The stack grows toward lower memory addresses. Since we graphically represent memory with addresses increasing from the bottom of a page to the top, we say that the stack grows "downward."
- Top-of-stack always points to the last data item placed on the stack. TOS, which is represented by SS:SP, always points to the lower byte of the last word inserted into the stack.

The statement

```
.STACK 100H
```

creates an empty stack as shown in Figure 10.3*a*, and allocates 256 (i.e., 100H) bytes of memory for stack operations. When the stack is initialized, TOS points to a byte just outside the reserved stack area. It is an error to read from an empty stack as this causes a *stack underflow*.

When a data item is pushed onto the stack, SP is first decremented by two, and then the word is stored at SS:SP. Since the Pentium uses little-endian byte order, the higher-order byte is stored in the higher memory address. For instance, when we push 21ABH, the stack expands by two bytes, and SP is decremented by two to point to the last data item, as shown in Figure 10.3*b*. The stack shown in Figure 10.3*c* results when we expand the stack further by four more bytes by pushing the doubleword 7FBD329AH onto the stack.

The stack full condition is indicated by the zero offset value (i.e., the SP register is 0000H). If we try to insert a data item into a full stack, *stack overflow* occurs. Both stack underflow and overflow are programming errors and should be handled with care.

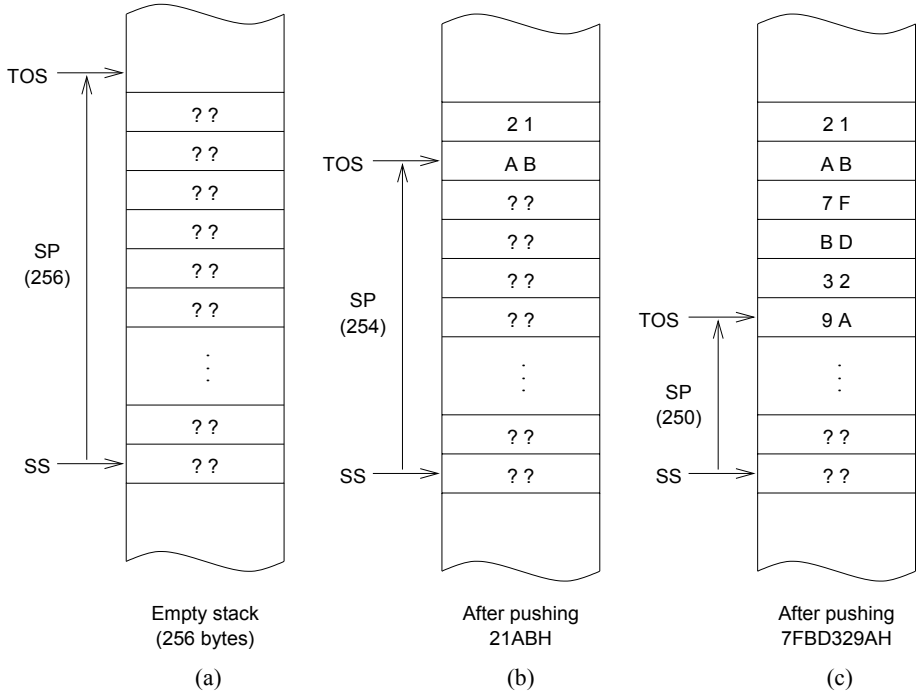


Figure 10.3 Stack implementation in the Pentium: SS:SP points to the top-of-stack.

Retrieving a 32-bit data item from the stack causes the offset value to increase by four to point to the next data item on the stack. For example, if we retrieve a doubleword from the stack shown in Figure 10.4*a*, we get 7FBD329AH from the stack and SP is updated, as shown in Figure 10.4*b*. Notice that the four memory locations retain their values. However, since TOS is updated, these four locations will be used to store the next data value pushed onto the stack, as shown in Figure 10.4*c*.

10.3 Stack Operations

10.3.1 Basic Instructions

The stack data structure allows two basic operations: insertion of a data item into the stack (called the *push* operation) and deletion of a data item from the stack (called the *pop* operation). The Pentium allows these two operations on word or doubleword data items. The syntax is

```
push    source
pop     destination
```

The operand of these two instructions can be a 16- or 32-bit general-purpose register, segment register, or a word or doubleword in memory. In addition, `source` for the push instruction

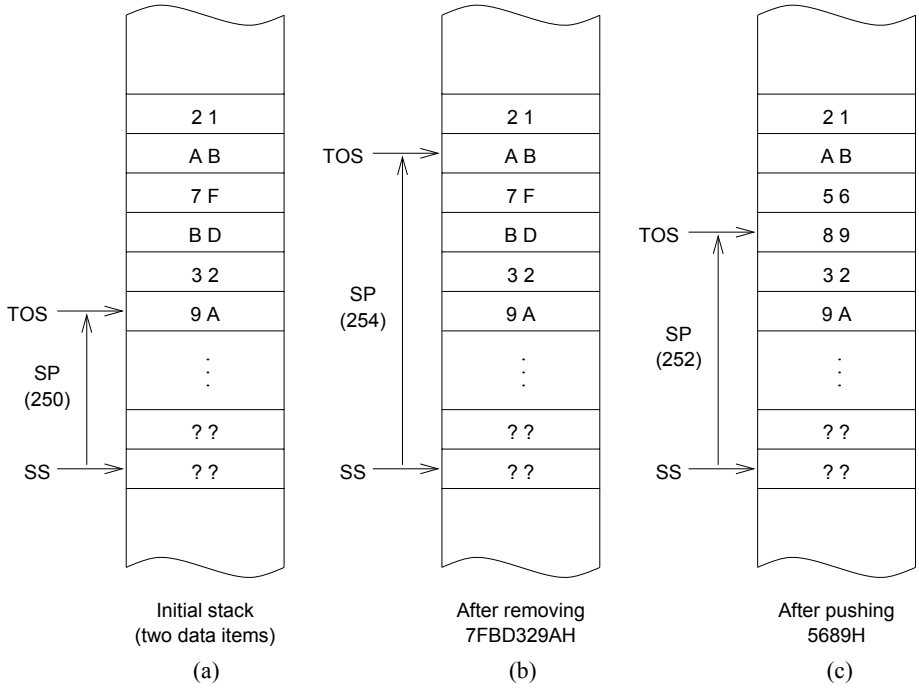


Figure 10.4 An example showing stack insert and delete operations.

can be an immediate operand of size 8, 16, or 32 bits. Table 10.1 summarizes the two stack operations.

On an empty stack created by

```
.STACK 100H
```

the statements

```
push    21ABH
push    7FBD329AH
```

would result in the stack shown in Figure 10.4*a*. Executing the statement

```
pop     EBX
```

on this stack would result in the stack shown in Figure 10.4*b* with the register EBX receiving 7FBD329AH.

10.3.2 Additional Instructions

The Pentium supports two special instructions for stack manipulation. These instructions can be used to save or restore the flags and general-purpose registers.

Table 10.1 Stack operations on 16- and 32-bit data

`push source16`	SP = SP − 2 (SS:SP) = `source16`	SP is first decremented by 2 to modify TOS. Then the 16-bit data from `source16` is copied onto the stack at the new TOS. The stack expands by 2 bytes.
`push source32`	SP = SP − 4 (SS:SP) = `source32`	SP is first decremented by 4 to modify TOS. Then the 32-bit data from `source32` is copied onto the stack at the new TOS. The stack expands by 4 bytes.
`pop dest16`	`dest16` = (SS:SP) SP = SP + 2	The data item located at TOS is copied to `dest16`. Then SP is incremented by 2 to update TOS. The stack shrinks by 2 bytes.
`pop dest32`	`dest32` = (SS:SP) SP = SP + 4	The data item located at TOS is copied to `dest32`. Then SP is incremented by 4 to update TOS. The stack shrinks by 4 bytes.

Stack Operations on Flags

The `push` and `pop` operations cannot be used to save or restore the flags register. For this, the Pentium provides two special versions of these instructions:

```
pushf    (push 16-bit flags)
popf     (pop 16-bit flags)
```

These instructions do not need any operands. For operating on the 32-bit flags register (EFLAGS), we can use `pushfd` and `popfd` instructions.

Stack Operations on All General-Purpose Registers

The Pentium also provides special `pusha` and `popa` instructions to save and restore the eight general-purpose registers. The `pusha` saves the 16-bit general-purpose registers AX, CX, DX, BX, SP, BP, SI, and DI. These registers are pushed in the order specified. The last register pushed is the DI register. The `popa` restores these registers except that it will not copy the SP value (i.e., the SP value is not loaded into the SP register as part of the `popa` instruction). The corresponding instructions for the 32-bit registers are `pushad` and `popad`. These instructions are useful in procedure calls, as we show in Section 10.8.4.

10.4 Uses of the Stack

The stack is used for three main purposes: as a scratchpad to temporarily store data, for transfer of program control, and for passing parameters during a procedure call.

10.4.1 Temporary Storage of Data

The stack can be used as a scratchpad to store data on a temporary basis. For example, consider exchanging the contents of two 32-bit variables that are in the memory: `value1` and `value2`. We cannot use

```
xchg    value1,value2       ; illegal
```

because both operands of `xchg` are in the memory. The code

```
mov    EAX,value1
mov    EBX,value2
mov    value1,EBX
mov    value2,EAX
```

works, but it uses two 32-bit registers. This code requires four memory operations. However, due to the limited number of general-purpose registers, finding spare registers that can be used for temporary storage is nearly impossible in almost all programs.

What if we need to preserve the contents of the EAX and EBX registers? In this case, we need to save these registers before using them and restore them as shown below:

```
        . . .
;save EAX and EBX registers on the stack
     push   EAX
     push   EBX
     ;EBX and ECX registers can now be used
     mov    EAX,value1
     mov    EBX,value2
     mov    value1,EBX
     mov    value2,EAX
;restore EBX and ECX registers from the stack
     pop    EBX
     pop    EAX
        . . .
```

This code requires eight memory accesses. Because the stack is a LIFO data structure, the sequence of `pop` instructions is a mirror image of the `push` instruction sequence.

An elegant way of exchanging the two values is

```
push    value1
push    value2
pop     value1
pop     value2
```

Notice that the above code does not use any general-purpose registers and requires eight memory operations as in the other example. Another point to note is that `push` and `pop` instructions allow movement of data from memory to memory (i.e., between data and stack segments). This is a special case because `mov` instructions do not allow memory-to-memory data transfer. Stack operations are an exception. String instructions, discussed in Chapter 12, also allow memory-to-memory data transfer.

Stack is frequently used as a scratchpad to save and restore registers. The necessity often arises when we need to free up a set of registers so they can be used by the current code. This is often the case with procedures as we show in Section 10.8.

It should be clear from these examples that the stack grows and shrinks during the course of a program execution. It is important to allocate enough storage space for the stack, as stack overflow and underflow could cause unpredictable results, often causing system errors.

10.4.2 Transfer of Control

The previous discussion concentrated on how we, as programmers, can use the stack to store data temporarily. The stack is also used by some instructions to store data temporarily. In particular, when a procedure is called, the return address of the instruction is stored on the stack so that the control can be transferred back to the calling program. A detailed discussion of this topic is in Section 10.7.

10.4.3 Parameter Passing

Another important use of the stack is to act as a medium to pass parameters to the called procedure. The stack is extensively used by high-level languages to pass parameters. A discussion on the use of the stack for parameter passing is deferred until Section 10.8.

10.5 Procedures

A procedure is a logically self-contained unit of code designed to perform a particular task. These are sometimes referred to as *subprograms* and play an important role in modular program development. In high-level languages such as Pascal, there are two types of subprograms: *procedures* and *functions*. There is a strong similarity between a Pascal function and a mathematical function. Each function receives a list of arguments and performs a computation based on the arguments passed onto it and returns a single value. Procedures also receive a list of arguments just as the functions do. However, procedures, after performing their computation, may return zero or more results back to the calling procedure. In C language, both these subprogram types are combined into a single function construct.

In the C function

```
int sum (int x, int y)
{
    return (x + y);
}
```

the parameters `x` and `y` are called formal parameters and the function body is defined based on these parameters. When this function is called (or invoked) by a statement like

```
total = sum (number1, number2);
```

the actual parameters—`number1` and `number2`—are used in the computation of the function `sum`.

There are two types of parameter passing mechanisms: *call-by-value* and *call-by-reference*. In the call-by-value mechanism, the called function (`sum` in our example) is provided only the current value of the arguments for its use. Thus, in this case, the values of these actual parameters are not changed in the called function; these values can only be used as in a mathematical function. In our example, the `sum` function is invoked by using the call-by-value mechanism, as we simply pass the values of `number1` and `number2` to the called function `sum`.

In the call-by-reference mechanism, the called function actually receives the addresses (i.e., pointers) of the parameters from the calling function. The function can change the contents of these parameters—and these changes will be seen by the calling function—by directly manipulating the actual parameter storage space. For instance, the following `swap` function

```
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

assumes that `swap` receives the addresses of the two parameters from the calling function. Thus, we are using the call-by-reference mechanism for parameter passing. Such a function can be invoked by

```
swap (&data1, &data2);
```

Often both types of parameter passing mechanisms are used in the same function. As an example, consider finding the roots of the quadratic equation

$$ax^2 + bx + c = 0\,.$$

The two roots are defined as

$$\text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\,,$$

$$\text{root2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,.$$

The roots are real if $b^2 \geq 4ac$, and imaginary otherwise.

Suppose that we want to write a function that receives a, b, and c and returns the values of the two roots (if real) and indicates whether the roots are real or imaginary.

```
int roots (double a, double b, double c,
           double *root1, double *root2)
{
    int root_type = 1;
    if (4 * a * c <= b * b){  /* roots are real */
       *root1 = (-b + sqrt(b*b - 4*a*c))/(2*a);
       *root2 = (-b - sqrt(b*b - 4*a*c))/(2*a);
    }
    else    /* roots are imaginary */
       root_type = 0;
    return (root_type);
 }
```

The function `roots` receives parameters `a`, `b`, and `c` by the call-by-value mechanism, and `root1` and `root2` parameters are passed using the call-by-reference mechanism. A typical invocation of `roots` is

```
root_type = roots (a, b, c, &root1, &root2);
```

In summary, procedures receive a list of parameters, which may be passed either by the call-by-value or by the call-by-reference mechanism. If more than one result is to be returned by a called procedure, the call-by-reference parameter passing mechanism should be used.

10.6 Assembler Directives for Procedures

Assemblers provide two directives to define procedures in the assembly language: `PROC` and `ENDP`. The `PROC` directive (stands for procedure) signals the beginning of a procedure, and `ENDP` (end procedure) indicates the end of a procedure. Both these directives take a label that is the name of the procedure. In addition, the `PROC` directive may optionally include `NEAR` or `FAR` to indicate whether the procedure is a `NEAR` procedure or a `FAR` procedure. The general format is

```
proc-name    PROC    NEAR
```

to define a near procedure, and

```
proc-name    PROC    FAR
```

to define a far procedure.

A procedure is said to be a near procedure if the calling and called procedures are both located in the same code segment. On the other hand, if the calling and called procedures are located in two different code segments, they are called far procedures. Near procedures involve intrasegment calls, and far procedures involve intersegment calls. Here we restrict our attention to near procedure calls.

When `NEAR` or `FAR` is not included in the `PROC` directive, the procedure definition defaults to the `NEAR` procedure. A typical procedure definition is

```
proc-name PROC
            . . .
      <procedure body>
            . . .
proc-name ENDP
```

where `proc-name` in both `PROC` and `ENDP` statements must match.

10.7 Pentium Instructions for Procedures

The Pentium provides `call` and `ret` (return) instructions to write procedures in assembly language. The `call` instruction can be used to invoke a procedure, and has the format

```
call    proc-name
```

where `proc-name` is the name of the procedure to be called. The assembler replaces `proc-name` by the offset value of the first instruction of the called procedure.

10.7.1 How Is Program Control Transferred?

The offset value provided in the `call` instruction is not the absolute value (i.e., offset is not relative to the start of the code segment pointed to by the CS register), but a relative displacement in bytes from the instruction following the `call` instruction. Let us look at the following example:

```
 offset   machine code
(in hex)    (in hex)
                      main  PROC
                              . . .
cs:000A      E8000C           call   sum
cs:000D      8BD8             mov    BX,AX
                              . . .
                      main  ENDP

                      sum   PROC
cs:0019      55               push   BP
                              . . .
                      sum   ENDP

                      avg   PROC
                              . . .
cs:0028      E8FFEE           call   sum
cs:002B      8BD0             mov    DX,AX
                              . . .
                      avg   ENDP
```

The `call` instruction of `main` is located at CS:000AH and the next instruction at CS:000DH. The first instruction of the procedure `sum` is at CS:0019H in the code segment. After the `call` instruction has been fetched, the IP register points to the next instruction to be executed (i.e., IP = 000DH). This is the instruction that should be executed after completing the execution of `sum`. The processor makes a note of this by pushing the contents of the IP register onto the stack.

Now, to transfer control to the first instruction of the `sum` procedure, the IP register would have to be loaded with the offset value of the

```
push BP
```

instruction in `sum`. To do this, the processor adds the 16-bit relative displacement found in the `call` instruction to the contents of the IP register. Proceeding with our example, the machine language encoding of the `call` instruction, which requires three bytes, is E8000CH. The first byte E8H is the opcode for the `call` and the next two bytes give the (signed) relative displacement in bytes. In this example, it is the difference between 0019H (offset of the `push BP` instruction in `sum`) and 000DH (offset of the instruction `mov BX,AX` in `main`). Therefore, 0019H − 000DH = 000CH. Adding this difference to the contents of the IP register (after fetching the `call` instruction) leaves the IP register pointing to the first instruction of `sum`.

Note that the procedure call in `main` is a forward call, and therefore the relative displacement is a positive number. As an example of a backward procedure call, let us look at the `sum` procedure call in the `avg` procedure. In this case, the program control has to be transferred back. That is, the displacement is a negative value. Following the explanation given in the last paragraph, we can calculate the displacement as 0019H − 002BH = FFEEH. Since negative numbers are expressed in 2's complement notation, FFEEH corresponds to −12H (i.e., −18D), which is the displacement value in bytes.

The following is a summary of the actions taken during a near procedure call:

```
SP = SP – 2                        ; push return address onto the stack
(SS:SP) = IP
IP = IP + relative displacement    ; update IP to point to the procedure
```

The relative displacement is a signed 16-bit number to accommodate both forward and backward procedure calls.

10.7.2 The ret Instruction

The `ret` (return) instruction is used to transfer control from the called procedure to the calling procedure. Return transfers control to the instruction following the `call` (instruction `mov BX,AX` in our example). How will the processor know where this instruction is located? Remember that the processor made a note of this when the `call` instruction was executed. When the `ret` instruction is executed, the return address from the stack is recovered. The actions taken during the execution of the `ret` instruction are

```
IP = SS:SP     ; pop return address at TOS into IP
SP = SP + 2    ; update TOS by adding 2 to SP
```

An optional integer may be included in the `ret` instruction, as in

```
ret 6
```

The details on this optional number are covered in Section 10.8.2, which discusses the stack-based parameter passing mechanism.

10.8 Parameter Passing

Parameter passing in assembly language is different and more complicated than that used in high-level languages. In assembly language, the calling procedure first places all the parameters needed by the called procedure in a mutually accessible storage area (usually registers or memory). Only then can the procedure be invoked. There are two common methods depending on the type of storage area used to pass parameters: *register method* or *stack method*. As their names imply, the register method uses general-purpose registers to pass parameters, and the stack is used in the other method.

10.8.1 Register Method

In the register method, the calling procedure places the necessary parameters in the general-purpose registers before invoking the procedure. Next, let us look at a couple of examples before considering the advantages and disadvantages of passing parameters using the register method.

Example 10.1 *Parameter passing by call-by-value using registers.*
In this example, two parameter values are passed to the called procedure via the general-purpose registers. The procedure `sum` receives two integers in the CX and DX registers and returns the sum of these two integers via AX. No check is done to detect the overflow condition. The program, shown in Program 10.1, requests two integers from the user and displays the sum on the screen.

Program 10.1 Parameter passing by call-by-value using registers

```
 1:  TITLE   Parameter passing via registers      PROCEX1.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via registers
 4:              Input: Requests two integers from the user.
 5:  |          Output: Outputs the sum of the input integers.
 6:  .MODEL SMALL
 7:  .STACK 100H
 8:  .DATA
 9:  prompt_msg1  DB   'Please input the first number: ',0
10:  prompt_msg2  DB   'Please input the second number: ',0
11:  sum_msg      DB   'The sum is ',0
12:
```

```
13:  .CODE
14:  INCLUDE io.mac
15:
16:  main  PROC
17:        .STARTUP
18:        PutStr  prompt_msg1    ; request first number
19:        GetInt  CX             ; CX := first number
20:        nwln
21:        PutStr  prompt_msg2    ; request second number
22:        GetInt  DX             ; DX := second number
23:        nwln
24:        call    sum            ; returns sum in AX
25:        PutStr  sum_msg        ; display sum
26:        PutInt  AX
27:        nwln
28:  done:
29:        .EXIT
30:  main  ENDP
31:
32:  ;-----------------------------------------------------------
33:  ;Procedure sum receives two integers in CX and DX.
34:  ; The sum of the two integers is returned in AX.
35:  ;-----------------------------------------------------------
36:  sum   PROC
37:        mov     AX,CX          ; sum := first number
38:        add     AX,DX          ; sum := sum + second number
39:        ret
40:  sum   ENDP
41:        END     main
```

Example 10.2 *Parameter passing by call-by-reference using registers.*
This example shows how parameters can be passed by call-by-reference using the register method. The program requests a character string from the user and displays the number of characters in the string (i.e., string length). The string length is computed by the function `str_len`. This function scans the input string for the NULL character while keeping track of the number of characters in the string. The pseudocode is shown below:

```
str_len (string)
    index := 0
    length := 0
    while (string[index] ≠ NULL)
        index := index + 1
        length := length + 1      { AX is used for string length}
```

```
        end while
        return (length)
    end str_len
```

The `str_len` function receives a pointer to the string in BX and returns the string length in the AX register. The program listing is given in Program 10.2. The `main` procedure executes

```
mov    BX,OFFSET string
```

to place the address of `string` in BX (line 23) before invoking the procedure on line 24. Note that even though the procedure modifies the BX register during its execution, it restores the original value of BX, pointing to the string, by saving its value initially on the stack (line 37) and restoring it (line 46) before returning to the `main` procedure.

Program 10.2 Parameter passing by call-by-reference using registers

```
 1:  TITLE   Parameter passing via registers      PROCEX2.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via registers
 4:              Input: Requests a character string from the user.
 5:  |          Output: Outputs the length of the input string.
 6:
 7:  BUF_LEN     EQU  41          ; string buffer length
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  string      DB  BUF_LEN DUP (?)  ;input string < BUF_LEN chars.
12:  prompt_msg  DB  'Please input a string: ',0
13:  length_msg  DB  'The string length is ',0
14:
15:  .CODE
16:  INCLUDE io.mac
17:
18:  main  PROC
19:        .STARTUP
20:        PutStr  prompt_msg     ; request string input
21:        GetStr  string,BUF_LEN ; read string from keyboard
22:        nwln
23:        mov     BX,OFFSET string  ; BX := string address
24:        call    str_len        ; returns string length in AX
25:        PutStr  length_msg     ; display string length
26:        PutInt  AX
27:        nwln
28:  done:
29:        .EXIT
30:  main  ENDP
```

```
31:
32:  ;-------------------------------------------------------------
33:  ;Procedure str_len receives a pointer to a string in BX.
34:  ; String length is returned in AX.
35:  ;-------------------------------------------------------------
36:  str_len PROC
37:        push    BX
38:        sub     AX,AX          ; string length := 0
39:  repeat:
40:        cmp     BYTE PTR [BX],0 ; compare with NULL char.
41:        je      str_len_done   ; if NULL we are done
42:        inc     AX             ; else, increment string length
43:        inc     BX             ; point BX to the next char.
44:        jmp     repeat         ;  and repeat the process
45:  str_len_done:
46:        pop     BX
47:        ret
48:  str_len ENDP
49:        END     main
```

Pros and Cons of the Register Method

The register method has its advantages and disadvantages. These are summarized here.

Advantages:

1. The register method is convenient and easier for passing a small number of parameters.
2. This method is also faster because all the parameters are available in registers.

Disadvantages:

1. The main disadvantage is that only a few parameters can be passed by using registers, as there are a limited number of general-purpose registers available in the CPU.
2. Another problem is that the general-purpose registers are often used by the calling procedure for some other purpose. Thus, it is necessary to temporarily save the contents of these registers on the stack to free them for use in parameter passing before calling a procedure, and restore them after returning from the called procedure. In this case, it is difficult to realize the second advantage listed above, as the stack operations involve memory access.

10.8.2 Stack Method

In this method of parameter passing, all parameters required by a procedure are pushed onto the stack before the procedure is called. As an example, let us consider passing the two parameters required by the `sum` procedure shown in Program 10.1. This can be done by

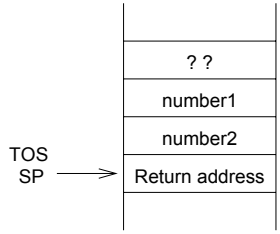


Figure 10.5 Stack state after the `sum` procedure call: Return address is the IP value pushed onto the stack as part of executing the call instruction.

```
push    number1
push    number2
call    sum
```

After executing the call instruction, which automatically pushes the IP contents onto the stack, the stack state is shown in Figure 10.5.

Reading the two arguments—`number1` and `number2`—is tricky. Since the parameter values are buried inside the stack, first we have to pop the IP value to read the required two parameters. This, for example, can be done by

```
pop     AX
pop     BX
pop     CX
```

in the `sum` procedure. Since we have removed the return address (IP) from the stack, we will have to restore it by

```
push    AX
```

so that TOS is pointing to the return address.

The main problem with this code is that we need to set aside general-purpose registers to copy parameter values. This means that the calling procedure cannot use these registers for any other purpose. Worse still, what if you want to pass 10 parameters? One way to free up registers is to copy the parameters from the stack to local data variables, but this is impractical and inefficient.

The best way to get parameter values is to leave them on the stack and read them from the stack as needed. Since the stack is a sequence of memory locations, SP + 2 points to `number2`, and SP + 4 to `number1`. Unfortunately, in 16-bit addressing mode

```
mov     BX,[SP+2]
```

is not allowed. However, we can increment SP by two and use SP in indirect addressing mode, as shown below:

```
add     SP,2
mov     BX,[SP]
```

A problem with this solution is that it is very cumbersome, as we have to remember to update SP to point to the return address on the stack before the end of the procedure.

In the 32-bit addressing mode, we can use ESP with a displacement to point to a parameter on the stack. For instance,

```
mov     BX,[ESP+2]
```

can be used, but this causes another problem. The stack pointer register is updated by the push and pop instructions. As a result, the relative offset changes with the stack operations performed in the called procedure. This is not a desirable situation.

There is a better alternative: we can use the BP register instead of SP to specify an offset into the stack segment. For example, we can copy the value of `number2` into the AX register by

```
mov     BP,SP
mov     AX,[BP+2]
```

This is the usual way of pointing to the parameters on the stack. Since every procedure uses the BP register to access parameters, the BP register should be preserved. Therefore, we should save the contents of the BP register before executing the

```
mov     BP,SP
```

statement. We, of course, use the stack for this. Note that

```
push    BP
mov     BP,SP
```

causes the parameter displacement to increase by two bytes, as shown in Figure 10.6*a*.

The information stored in the stack—parameters, return address, and the old BP value—is collectively called the *stack frame*. As we show on page 421, the stack frame also consists of local variables if the procedure uses them. The BP value is referred to as the *frame pointer* (FP). Once the BP value is known, we can access all items in the stack frame.

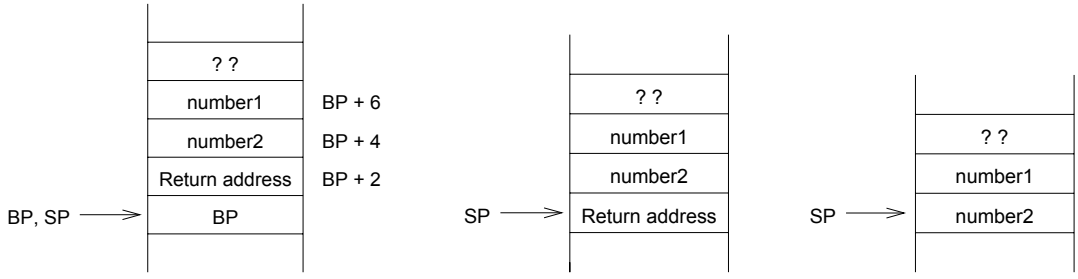
Before returning from the procedure, we should use

```
pop    BP
```

to restore the original value of BP. The resulting stack state is shown in Figure 10.6*b*.

The `ret` statement discussed in Section 10.7.2 causes the return address to be placed in the IP register, and the stack state after `ret` is shown in Figure 10.6*c*.

Now the problem is that the four bytes of the stack occupied by the two parameters are no longer useful. One way to free these four bytes is to increment SP by four after the call statement, as shown below:



(a) Stack after saving BP (b) Stack after `pop BP` (c) Stack after `ret`

Figure 10.6 Changes in stack state during a procedure execution.

```
push    number1
push    number2
call    sum
add     SP,4
```

For example, the Turbo C compiler uses this method to clear parameters from the stack. The above assembly language code segment corresponds to the

```
sum(number2, number1);
```

function call in C.

Rather than adjusting the stack by the calling procedure, the called procedure can also clear the stack. Note that we cannot write

```
sum     PROC
          . . .
        add     SP,4
        ret
sum     ENDP
```

because when `ret` is executed, SP should point to the return address on the stack.

The solution lies in the optional operand that can be specified in the `ret` statement. The format is

```
ret     optional-value
```

which results in the following sequence of actions:

IP = (SS:SP)
SP = SP + 2 + `optional-value`

The `optional-value` should be a number (i.e., immediate). Since the purpose of the optional value is to discard the parameters pushed onto the stack, this operand takes a positive value.

Who Should Clean Up the Stack?

We have discussed the following ways of discarding the unwanted parameters on the stack:

1. clean-up is done by the calling procedure, or
2. clean-up is done by the called procedure.

If procedures require a fixed number of parameters, the second method is preferred. In this case, we write the clean-up code only once in the called procedure independent of the number of times this procedure is called. We follow this convention in our assembly language programs. However, if a procedure receives a variable number of parameters, we have to use the first method. We discuss this topic in detail in a later section.

10.8.3 Preserving Calling Procedure State

It is important to preserve the contents of the registers across a procedure call. The necessity for this is illustrated by the following code:

```
            . . .
            mov     CX,count
repeat:
            call    compute
            . . .
            . . .
            loop    repeat
            . . .
```

The code invokes the `compute` procedure `count` times. The CX register maintains the number of remaining iterations. Recall that, as a part of the `loop` instruction execution, the CX register is decremented by 1 and, if not 0, starts another iteration.

Suppose, now, that the `compute` procedure uses the CX register during its computation. Then, when `compute` returns control to the calling program, CX would have changed, and the program logic would be incorrect.

Since there are a limited number of registers and registers should be used for writing efficient code, registers should be preserved. The stack is used to save registers temporarily.

10.8.4 Which Registers Should Be Saved?

The answer to this question is simple: Save those registers that are used by the calling procedure but changed by the called procedure. This leads to the following question: Which procedure, the calling or the called, should save the registers?

Usually, one or two registers are used to return a value by the called procedure. Therefore, such register(s) do not have to be saved. For example, in Turbo C, an integer C function returns the result in the AX register; if the function returns a `long int` data type result, which requires 32 bits, both the AX and DX registers are used.

In order to avoid the selection of the registers to be saved, we could save, blindly, all registers each time a procedure is invoked. For instance, we could use the `pusha` instruction (see page 392). But such an action may result in unnecessary overhead, as `pusha` takes five clocks to push all eight registers, whereas an individual register `push` instruction takes only one clock. Recall that producing efficient code is an important motivation for using the assembly language.

If the calling procedure were to save the necessary registers, it needs to know the registers used by the called procedure. This causes two serious difficulties:

1. Program maintenance would be difficult because, if the called procedure were modified later on and a different set of registers used, every procedure that calls this procedure would have to be modified.
2. Programs tend to be longer because if a procedure is called several times, we have to include the instructions to save and restore the registers each time the procedure is called.

For these reasons, we assume that the called procedure saves the registers that it uses and restores them before returning to the calling procedure. This also conforms to modular program design principles.

When to Use pusha

The `pusha` instruction is useful in certain instances, but not all. We identify some instances where `pusha` is not useful. First, what if some of the registers saved by `pusha` are used for returning results? For instance, Turbo C uses the AX register for returning a 16-bit result and the DX:AX register pair for a 32-bit result. In this case `pusha` is not really useful, as `popa` destroys the result to be returned to the calling procedure. Second, since `pusha` takes five clocks whereas a single `push` takes only a single clock, `pusha` is efficient only if you want to save more than four registers. In some instances where you want to save only one or two registers, it may be worthwhile to use the `push` instruction. Of course, the other side of the coin is that `pusha` improves readability of code and reduces memory required for the instructions.

When `pusha` is used to save registers, it modifies the offset of the parameters. Note that

```
pusha
mov     BP,SP
```

causes the stack state, shown in Figure 10.7, to be different from that shown in Figure 10.6*a* on page 405. You can see that the offset of `number1` and `number2` increases.

Pentium's ENTER and LEAVE Instructions

The Pentium has two instructions to facilitate stack frame allocation and release on procedure entry and exit. The `enter` instruction can be used to allocate a stack frame on entering a procedure. The format is

```
enter    bytes,level
```

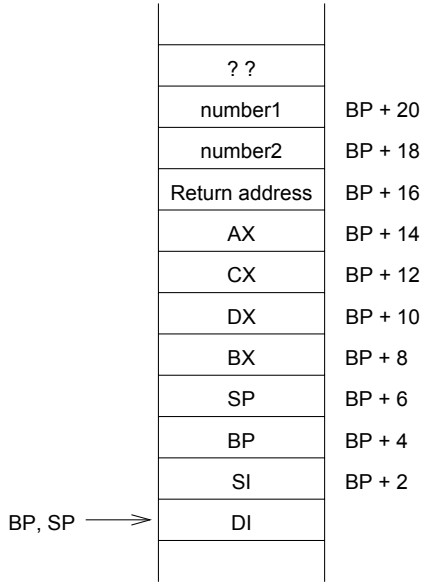


Figure 10.7 Stack state after `pusha`.

The first operand `bytes` specifies the number of bytes of local variable storage we want on the new stack frame. We do not need local variable space until Example 10.8 on page 424. Until then, we set the first operand to zero. The second operand `level` gives the nesting level of the procedure. If we specify a nonzero level, it copies `level` stack frame pointers into the new frame from the preceding stack frame. In all our examples, we set the second operand to zero. Thus, when the operand-size is 16 bits, the statement

```
enter    XX,0
```

is equivalent to

```
push     BP
mov      BP,SP
sub      SP,XX
```

As usual, if the operand size is 32 bits, EBP and ESP are used instead of BP and SP registers, respectively. Even when the operand-size is 16 bits, `enter` uses EBP and ESP if the stack size attribute is 32 bits.

The `leave` instruction releases the stack frame allocated by the `enter` instruction. It does not take any operands. When the operand size is 16 bits, the `leave` instruction effectively performs the following:

```
mov      SP,BP
pop      BP
```

We use the `leave` instruction before the `ret` instruction as shown in the following template for procedures:

```
proc-name  PROC
    enter   XX,0
        . . .
    procedure body
        . . .
    leave
    ret    YY
proc-name  ENDP
```

As we show on page 424, the `XX` value is nonzero only if our procedure needs some local variable space on the stack frame. The value `YY` is used to clear the arguments passed on to the procedure.

10.8.5 Illustrative Examples

In this section, we use three examples to illustrate the use of the stack for parameter passing.

Example 10.3 *Parameter passing by call-by-value using the stack.*
This is the stack counterpart of Example 10.1, which passes two integers to the procedure `sum`. The procedure returns the sum of these two integers in the AX register, as in Example 10.1. The program listing is given in Program 10.3.

The program requests two integers from the user. It reads the two numbers into the CX and DX registers using `GetInt` (lines 20 and 23). Since the stack is used to pass the two numbers, we have to place them on the stack before calling the `sum` procedure (see lines 25 and 26). The state of the stack after the control is transferred to `sum` is shown in Figure 10.5 on page 403.

As discussed in Section 10.8.2, the BP register is used to access the two parameters from the stack. Therefore, we have to save BP itself on the stack. We do this by using the `enter` instruction (line 40), which changes the stack state to that in Figure 10.6*a* on page 405.

The original value of BP is restored at the end of the procedure using the `leave` instruction (line 43). Accessing the two numbers follows the explanation given in Section 10.8.2. Note that the first number is at BP + 6, and the second one at BP + 4. As in Example 10.1, no overflow check is done by `sum`. Control is returned to `main` by

```
ret    4
```

because `sum` has received two parameters requiring a total space of four bytes on the stack. This `ret` statement will clear `number1` and `number2` from the stack.

Program 10.3 Parameter passing by call-by-value using the stack

```
 1:  TITLE   Parameter passing via the stack      PROCEX3.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via the stack.
 4:              Input: Requests two integers from the user.
 5:  |          Output: Outputs the sum of the input integers.
 6:  .MODEL SMALL
 7:  .STACK 100H
 8:  .DATA
 9:  prompt_msg1  DB   'Please input the first number: ',0
10:  prompt_msg2  DB   'Please input the second number: ',0
11:  sum_msg      DB   'The sum is ',0
12:
13:  .CODE
14:  .486
15:  INCLUDE io.mac
16:
17:  main  PROC
18:        .STARTUP
19:        PutStr  prompt_msg1    ; request first number
20:        GetInt  CX             ; CX = first number
21:        nwln
22:        PutStr  prompt_msg2    ; request second number
23:        GetInt  DX             ; DX = second number
24:        nwln
25:        push    CX             ; place first number on stack
26:        push    DX             ; place second number on stack
27:        call    sum            ; returns sum in AX
28:        PutStr  sum_msg        ; display sum
29:        PutInt  AX
30:        nwln
31:  done:
32:        .EXIT
33:  main  ENDP
34:
35:  ;-----------------------------------------------------------
36:  ;Procedure sum receives two integers via the stack.
37:  ; The sum of the two integers is returned in AX.
38:  ;-----------------------------------------------------------
39:  sum   PROC
40:        enter   0,0            ; save BP
41:        mov     AX,[BP+6]      ; sum = first number
42:        add     AX,[BP+4]      ; sum = sum + second number
43:        leave                  ; restore BP
```

```
44:         ret     4               ; return and clear parameters
45:  sum    ENDP
46:         END     main
```

Example 10.4 *Parameter passing by call-by-reference using the stack.*
This example shows how the stack can be used for parameter passing using the call-by-reference mechanism. The procedure `swap` receives two pointers to two characters and interchanges them. The program, shown in Program 10.4, requests a string from the user and displays the input string with the first two characters interchanged.

In preparation for calling `swap`, the `main` procedure places the addresses of the first two characters of the input string on the stack (lines 25 to 28). The `swap` procedure, after saving the BP register as in the last example, can access the pointers of the two characters at BP + 4 and BP + 6. Since the procedure uses the BX register, we save it on the stack as well. Note that, once the BP is pushed onto the stack and the SP value is copied to BP, the two parameters (i.e., the two character pointers in this example) are available at BP + 4 and BP + 6, irrespective of the other stack push operations in the procedure. This is important from the program maintenance point of view.

Program 10.4 Parameter passing by call-by-reference using the stack

```
 1:  TITLE   Parameter passing via the stack      PROCSWAP.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via the stack.
 4:              Input: Requests a character string from the user.
 5:             Output: Outputs the input string with the first
 6:  |                  two characters swapped.
 7:
 8:  BUF_LEN     EQU  41          ; string buffer length
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
12:  string      DB  BUF_LEN DUP (?)  ;input string < BUF_LEN chars.
13:  prompt_msg  DB  'Please input a string: ',0
14:  output_msg  DB  'The swapped string is: ',0
15:
16:  .CODE
17:  .486
18:  INCLUDE io.mac
19:
20:  main  PROC
21:        .STARTUP
22:        PutStr  prompt_msg     ; request string input
```

```
23:         GetStr  string,BUF_LEN ; read string from the user
24:         nwln
25:         mov     AX,OFFSET string  ; AX = string[0] pointer
26:         push    AX              ; push string[0] pointer on stack
27:         inc     AX              ; AX = string[1] pointer
28:         push    AX              ; push string[1] pointer on stack
29:         call    swap            ; swaps the first two characters
30:         PutStr  output_msg      ; display the swapped string
31:         PutStr  string
32:         nwln
33:   done:
34:         .EXIT
35:   main  ENDP
36:
37:   ;-----------------------------------------------------------
38:   ;Procedure swap receives two pointers (via the stack) to
39:   ; characters of a string. It exchanges these two characters.
40:   ;-----------------------------------------------------------
41:   swap  PROC
42:         enter   0,0
43:         push    BX              ; save BX - procedure uses BX
44:         ; swap begins here. Because of xchg, AL is preserved.
45:         mov     BX,[BP+6]       ; BX = first character pointer
46:         xchg    AL,[BX]
47:         mov     BX,[BP+4]       ; BX = second character pointer
48:         xchg    AL,[BX]
49:         mov     BX,[BP+6]       ; BX = first character pointer
50:         xchg    AL,[BX]
51:         ; swap ends here
52:         pop     BX              ; restore registers
53:         leave
54:         ret     4               ; return and clear parameters
55:   swap  ENDP
56:         END     main
```

Example 10.5 *Bubble sort procedure.*

There are several algorithms to sort an array of numbers. The particular algorithm that we are using here is called the *bubble sort* algorithm. Next we describe the algorithm to sort numbers in ascending order.

Bubble Sort: The bubble sort algorithm consists of several passes through the array of numbers to be sorted. Each pass scans the array, performing the following actions:

- Compare adjacent pairs of data elements.
- If they are out of order, swap them.

```
        Initial state:  4 3 5 1 2
After 1st comparison:  3 4 5 1 2 (4 and 3 swapped)
After 2nd comparison:  3 4 5 1 2 (no swap)
After 3rd comparison:  3 4 1 5 2 (5 and 1 swapped)
   End of first pass:  3 4 1 2 5 (5 and 2 swapped)
```

Figure 10.8 Actions taken during the first pass of the bubble sort algorithm.

```
  Initial state:  4 3 5 1 2
 After 1st pass:  3 4 1 2 5 (5 in its final position)
 After 2nd pass:  3 1 2 4 5 (4 in its final position)
 After 3rd pass:  1 2 3 4 5 (array in sorted order)
After the final pass:  1 2 3 4 5 (final pass to check)
```

Figure 10.9 Behavior of the bubble sort algorithm.

The algorithm terminates if, during a pass, no data elements are swapped. If at least a single swap is done during a pass, it will initiate another pass to scan the array.

Figure 10.8 shows the behavior of the algorithm during the first pass. The algorithm starts by comparing the first and second data elements (`4` and `3`). Since they are out of order, `4` and `3` are interchanged. Next, the second data element `4` is compared with the third data element `5`, and no swapping takes place as they are in order. During the next step, `5` and `1` are compared and swapped and finally `5` and `2` are swapped. This terminates the first pass. The algorithm has performed $N-1$ comparisons, where N is the number of data elements in the array. At the end of the first pass, the largest data element `5` is moved to its final position in the array.

Figure 10.9 shows the state of the array after each pass. Notice that after the first pass, the largest number (`5`) is in its final position. Similarly, after the second pass, the second largest number (`4`) moves to its final position, and so on. This is why this algorithm is called the bubble sort: during the first pass, the largest element bubbles to the top, the second largest bubbles to the top during the second pass, and so on. Even though the array is in sorted order after the third pass, one more pass is required by the algorithm to detect that the array is sorted.

The number of passes required to sort an array depends on how unsorted the initial array is. If the array elements are already in sorted order, only a single pass is required. At the other extreme, if the array is completely unsorted (i.e., elements are initially in the descending order), the algorithm requires a number of passes equal to one less than the number of elements in the array. The pseudocode for the bubble sort algorithm is shown in Figure 10.10.

Bubble Sort Program: This program requests a set of up to 20 nonzero integers from the user and displays them in sorted order. The input can be terminated earlier by typing a zero.

The logic of the main program is straightforward. The `read_loop` (lines 26 to 34) reads the input integers. Since the CX is initialized to `MAX_SIZE`, which is set to 20 in this program,

```
bubble_sort (arrayPointer, arraySize)
    status := UNSORTED
    #comparisons := arraySize
    while (status = UNSORTED)
        #comparisons := #comparisons − 1
        status := SORTED
        for (i = 0 to #comparisons)
            if (array[i] > array[i+1])
                swap ith and (i + 1)th elements of the array
                status := UNSORTED
            end if
        end for
    end while
end bubble_sort
```

Figure 10.10 Pseudocode for the bubble sort algorithm.

the `read_loop` iterates a maximum of 20 times. Typing a zero can also terminate the reading of input integers. The zero input condition is detected and the loop is terminated by the statements on lines 29 and 30.

The `bubble_sort` procedure receives the size of the array to be sorted and a pointer to the array. These two parameters are pushed onto the stack (lines 36 and 37) before calling the `bubble_sort` procedure. The `print_loop` (lines 41 to 47) displays the sorted array.

In the `bubble-sort` procedure, the CX register is used to keep track of the number of comparisons while DX maintains the status information. The SI register points to the ith element of the input array.

The `while` loop condition is tested by lines 92 to 94. The `for` loop body corresponds to lines 79 to 90 and 96 to 101. The rest of the code follows the pseudocode. Note that the array pointer is available in the stack at BP + 18 and its size at BP + 20, as we use `pusha` to save all registers. Also notice that this program uses only the 16-bit addressing modes.

Program 10.5 Bubble sort program to sort integers in ascending order

```
1:  COMMENT |         Bubble sort procedure     BBLSORT.ASM
2:   Objective: To implement the bubble sort algorithm.
3:       Input: A set of nonzero integers to be sorted.
4:      Input is terminated by entering zero.
5:  |          Output: Outputs the numbers in ascending order.
6:  CRLF        EQU   0DH,0AH
7:  MAX_SIZE    EQU   20
8:  .MODEL SMALL
9:  .STACK 100H
```

```
10:  .DATA
11:  array       DW  MAX_SIZE DUP (?)  ; input array for integers
12:  prompt_msg  DB  'Enter nonzero integers to be sorted.',CRLF
13:       DB  'Enter zero to terminate the input.',0
14:  output_msg  DB  'Input numbers in ascending order:',0
15:
16:  .CODE
17:  .486
18:  INCLUDE   io.mac
19:  main  PROC
20:        .STARTUP
21:        PutStr  prompt_msg     ; request input numbers
22:        nwln
23:        mov     BX,OFFSET array  ; BX := array pointer
24:        mov     CX,MAX_SIZE    ; CX := array size
25:        sub     DX,DX          ; number count := 0
26:  read_loop:
27:        GetInt  AX             ; read input number
28:        nwln
29:        cmp     AX,0           ; if the number is zero
30:        je      stop_reading   ; no more numbers to read
31:        mov     [BX],AX        ; copy the number into array
32:        add     BX,2           ; BX points to the next element
33:        inc     DX             ; increment number count
34:        loop    read_loop      ; reads a max. of MAX_SIZE numbers
35:  stop_reading:
36:        push    DX             ; push array size onto stack
37:        push    OFFSET array   ; place array pointer on stack
38:        call    bubble_sort
39:        PutStr  output_msg     ; display sorted input numbers
40:        nwln
41:        mov     BX,OFFSET array
42:        mov     CX,DX          ; CX := number count
43:  print_loop:
44:        PutInt  [BX]
45:        nwln
46:        add     BX,2
47:        loop    print_loop
48:  done:
49:        .EXIT
50:  main  ENDP
51:  ;-----------------------------------------------------------
52:  ;This procedure receives a pointer to an array of integers
53:  ; and the size of the array via the stack. It sorts the
54:  ; array in ascending order using the bubble sort algorithm.
```

```
55:  ;-------------------------------------------------------------
56:  SORTED    EQU   0
57:  UNSORTED  EQU   1
58:  bubble_sort     PROC
59:        pusha
60:        mov     BP,SP
61:
62:        ;CX serves the same purpose as the end_index variable
63:        ; in the C procedure. CX keeps the number of comparisons
64:        ; to be done in each pass. Note that CX is decremented
65:        ; by 1 after each pass.
66:        mov     CX, [BP+20]  ; load array size into CX
67:        mov     BX, [BP+18]  ; load array address into BX
68:
69:  next_pass:
70:        dec     CX           ; if # of comparisons is zero
71:        jz      sort_done    ; then we are done
72:        mov     DI,CX        ; else start another pass
73:
74:        ;DX is used to keep SORTED/UNSORTED status
75:        mov     DX,SORTED    ; set status to SORTED
76:
77:        ;SI points to element X and SI+2 to the next element
78:        mov     SI,BX        ; load array address into SI
79:  pass:
80:        ;This loop represents one pass of the algorithm.
81:        ;Each iteration compares elements at [SI] and [SI+2]
82:        ; and swaps them if ([SI]) < ([SI+2]).
83:        mov     AX,[SI]
84:        cmp     AX,[SI+2]
85:        jg      swap
86:  increment:
87:        ;Increment SI by 2 to point to the next element
88:        add     SI,2
89:        dec     DI
90:        jnz     pass
91:
92:        cmp     DX,SORTED       ; if status remains SORTED
93:        je      sort_done       ; then sorting is done
94:        jmp     next_pass       ; else initiate another pass
95:
96:  swap:
97:        ; swap elements at [SI] and [SI+2]
98:        xchg    AX,[SI+2]
99:        mov     [SI],AX
```

```
100:        mov     DX,UNSORTED     ; set status to UNSORTED
101:        jmp     increment
102:
103:  sort_done:
104:        popa
105:        ret     4               ; return and clear parameters
106:  bubble_sort    ENDP
107:        END    main
```

10.9 Handling a Variable Number of Parameters

Procedures in C can be defined to accept a variable number of parameters. The input and output functions, `scanf` and `printf`, are the two common procedures that take a variable number of parameters. In this case, the called procedure does not know the number of parameters passed onto it. Usually, the first parameter in the parameter list specifies the number of parameters passed. This parameter should be pushed onto the stack last so that it is just below the return address independent of the number of parameters passed.

In assembly language procedures, a variable number of parameters can be easily handled by the stack method of parameter passing. Only the stack size imposes a limit on the number of parameters that can be passed. The next example illustrates the use of the stack to pass variable numbers of parameters in assembly language programs:

Example 10.6 *Passing a variable number of parameters via the stack.*
In this example, the procedure `variable_sum` receives a variable number of integers via the stack. The actual number of integers passed is the last parameter pushed onto the stack before calling the procedure. The procedure finds the sum of the integers and returns this value in the AX register.

The `main` procedure in Program 10.6 requests input from the user. Only nonzero values are accepted as valid input (entering a zero terminates the input). The `read_number` loop (lines 26 to 33) reads input numbers using `GetInt` and pushes them onto the stack. The CX register keeps a count of the number of input values, which is passed as the last parameter (line 35) before calling the `variable_sum` procedure. The state of the stack at line 56, after executing the `enter` instruction, is shown in Figure 10.11.

The `variable_sum` procedure first reads the number of parameters passed onto it from the stack at BP + 4 into the CX register. The `add_loop` (lines 63 to 66) successively reads each integer from the stack and computes their sum in the AX. Note that on line 64 we use a segment override prefix. If we write

```
    add     AX,[BX]
```

the contents of the BX are treated as the offset value into the data segment. However, our parameters are located in the stack segment. Therefore, it is necessary to indicate that the offset

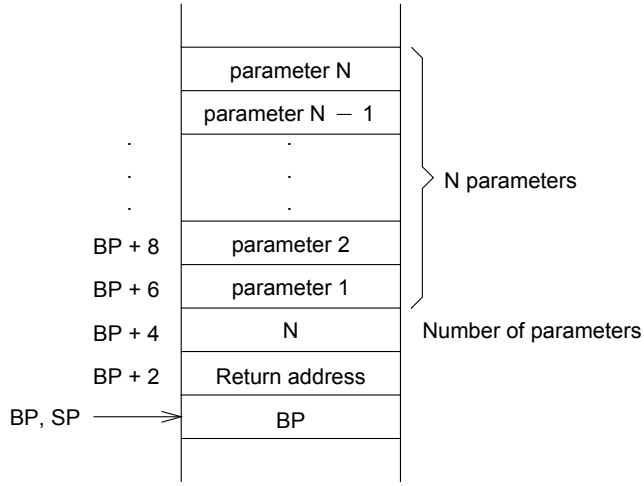


Figure 10.11 State of the stack after executing the `enter` statement.

in BX is relative to SS (and not DS). The segment override prefixes—CS:, DS:, ES:, FS:, GS:, and SS:—can be placed in front of a memory operand to indicate a segment other than the default segment.

In this example, we have deliberately used the BX to illustrate the use of segment override prefixes. We could have used the BP itself to access the parameters. For example, the code

```
        add     BP,6
        sub     AX,AX
add_loop:
        add     AX,[BP]
        add     BP,2
        loop    add_loop
```

can replace the code at lines 61 to 66. A disadvantage of this modified code is that, since we have modified the BP, we no longer can access, for example, the parameter count value in the stack. For this example, however, this method works fine. A better way is to use an index register to represent the offset relative to the BP. We defer this discussion until Chapter 11, which discusses the addressing modes of the Pentium.

Another interesting feature is that the parameter space on the stack is cleared by `main`. Since we pass a variable number of parameters, we cannot use `ret` to clear the parameter space. This is done in `main` by lines 38 to 40. The CX is first incremented to include the count parameter (line 38). The byte count of the parameter space is computed on line 39. This value is added to the SP register to clear the parameter space (line 40).

Program 10.6 Program to illustrate passing a variable number of parameters

```
 1:  TITLE   Variable # of parameters passed via stack   VARPARA.ASM
 2:  COMMENT |
 3:          Objective: To show how variable number of parameters
 4:                     can be passed via the stack.
 5:              Input: Requests variable number of nonzero integers.
 6:                     A zero terminates the input.
 7:  |           Output: Outputs the sum of input numbers.
 8:  CRLF   EQU   0DH,0AH    ; carriage return and line feed
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
12:  prompt_msg  DB  'Please input a set of nonzero integers.',CRLF
13:              DB  'You must enter at least one integer.',CRLF
14:              DB  'Enter zero to terminate the input.',0
15:  sum_msg     DB  'The sum of the input numbers is: ',0
16:
17:  .CODE
18:  .486
19:  INCLUDE io.mac
20:
21:  main  PROC
22:        .STARTUP
23:        PutStr  prompt_msg     ; request input numbers
24:        nwln
25:        sub     CX,CX          ; CX keeps number count
26:  read_number:
27:        GetInt  AX             ; read input number
28:        nwln
29:        cmp     AX,0           ; if the number is zero
30:        je      stop_reading   ; no more nuumbers to read
31:        push    AX             ; place the number on stack
32:        inc     CX             ; increment number count
33:        jmp     read_number
34:  stop_reading:
35:        push    CX             ; place number count on stack
36:        call    variable_sum   ; returns sum in AX
37:        ; clear parameter space on the stack
38:        inc     CX             ; increment CX to include count
39:        add     CX,CX          ; CX = CX * 2 (space in bytes)
40:        add     SP,CX          ; update SP to clear parameter
41:                               ; space on the stack
```

```
42:         PutStr  sum_msg         ; display the sum
43:         PutInt  AX
44:         nwln
45:   done:
46:         .EXIT
47:   main  ENDP
48:
49:   ;-----------------------------------------------------------
50:   ;This procedure receives variable number of integers via the
51:   ; stack. The last parameter pushed on the stack should be
52:   ; the number of integers to be added. Sum is returned in AX.
53:   ;-----------------------------------------------------------
54:   variable_sum  PROC
55:         enter   0,0
56:         push    BX              ; save BX and CX
57:         push    CX
58:
59:         mov     CX,[BP+4]       ; CX = # of integers to be added
60:         mov     BX,BP
61:         add     BX,6            ; BX = pointer to first number
62:         sub     AX,AX           ; sum = 0
63:   add_loop:
64:         add     AX,SS:[BX]      ; sum = sum + next number
65:         add     BX,2            ; BX points to the next integer
66:         loop    add_loop        ; repeat count in CX
67:
68:         pop     CX              ; restore registers
69:         pop     BX
70:         leave
71:         ret                     ; parameter space cleared by main
72:   variable_sum  ENDP
73:         END     main
```

10.10 Local Variables

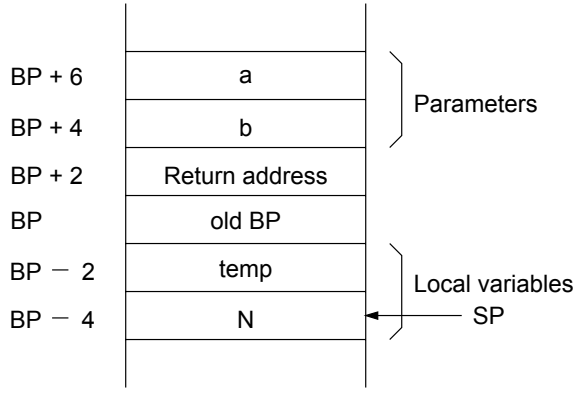
So far in our discussion, we have not considered how local variables can be used in a procedure. To focus our discussion, consider the following C code:

```
int compute(int a, int b)
{
   int   temp, N;
         . . .
         . . .
}
```

The variables `temp` and `N` are local variables that come into existence when the procedure `compute` is invoked and disappear when the procedure terminates. Thus, these local variables are dynamic. We could reserve space for the local variables in a data segment. However, such space allocation is not desirable for two reasons:

1. Space allocation done in the data segment is static and remains active even when the procedure is not.
2. More important, it does not work with recursive procedures (i.e., procedures that call themselves).

For these reasons, space for local variables is reserved on the stack. For the C function, the stack may look like



The above figure shows the contents of the stack frame. In high-level languages, it is also referred to as the *activation record* because each procedure activation requires all this information. The BP value, also called the *frame pointer*, allows us to access the contents of the stack frame. For example, parameters `a` and `b` can be accessed at BP + 6 and BP + 4, respectively. Local variables `temp` and `N` can be accessed at BP − 2 and BP − 4, respectively.

To aid program readability, we can use the `EQU` directive to name the stack locations. Thus, we can write

```
mov     BX,a
mov     temp,AX
```

instead of

```
mov     BX,[BP+6]
mov     [BP-2],AX
```

after establishing `temp` and `a` labels by using the EQU directive, as shown below.

```
a       EQU   WORD PTR [BP+6]
temp    EQU   WORD PTR [BP-2]
```

We now look at two examples, both of which compute Fibonacci numbers. However, one example uses registers for local variables, and the other uses the stack.

Example 10.7 *Fibonacci number computation using registers for local variables.*
The Fibonacci sequence of numbers is defined as

$$\begin{aligned} \text{fib}(1) &= 1, \\ \text{fib}(2) &= 1, \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \text{ for } n > 2. \end{aligned}$$

In other words, the first two numbers in the Fibonacci sequence are 1. The subsequent numbers are obtained by adding the previous two numbers in the sequence. Thus,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

is the Fibonacci sequence of numbers.

In this and the next example, we write a procedure to compute the largest Fibonacci number that is less than or equal to a given input number. The `main` procedure requests this number and passes it on to the `fibonacci` procedure.

The `fibonacci` procedure keeps the last two Fibonacci numbers in local variables. These are mapped to registers AX and BX. The higher of the two Fibonacci numbers is kept in the BX. The `fib_loop` successively computes the Fibonacci number until it is greater than or equal to the input number. Then the Fibonacci number in AX is returned to the `main` procedure.

Program 10.7 Fibonacci number computation with local variables mapped to registers

```
 1:  TITLE   Fibonacci numbers (register version)    PROCFIB1.ASM
 2:  COMMENT |
 3:          Objective: To compute Fibonacci number using registers
 4:                     for local variables.
 5:              Input: Requests a positive integer from the user.
 6:             Output: Outputs the largest Fibonacci number that
 7:  |                  is less than or equal to the input number.
 8:
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
12:  prompt_msg   DB  'Please input a positive number (>1): ',0
13:  output_msg1  DB  'The largest Fibonacci number less than '
14:               DB  'or equal to ',0
15:  output_msg2  DB  ' is ',0
16:
17:  .CODE
```

```
18:  INCLUDE io.mac
19:
20:  main  PROC
21:        .STARTUP
22:        PutStr  prompt_msg     ; request input number
23:        GetInt  DX             ; DX = input number
24:        nwln
25:        call    fibonacci
26:        PutStr  output_msg1    ; display Fibonacci number
27:        PutInt  DX
28:        PutStr  output_msg2
29:        PutInt  AX
30:        nwln
31:  done:
32:        .EXIT
33:  main  ENDP
34:
35:  ;-----------------------------------------------------------
36:  ;Procedure fibonacci receives an integer in DX and computes
37:  ; the largest Fibonacci number that is less than or equal to
38:  ; the input number. The Fibonacci number is returned in AX.
39:  ;-----------------------------------------------------------
40:  fibonacci  PROC
41:        push    BX
42:        ; AX maintains the smaller of the last two Fibonacci
43:        ;  numbers computed; BX maintains the larger one.
44:        mov     AX,1           ; initialize AX and BX to
45:        mov     BX,AX          ;  first two Fibonacci numbers
46:  fib_loop:
47:        add     AX,BX          ; compute next Fibonacci number
48:        xchg    AX,BX          ; maintain the required order
49:        cmp     BX,DX          ; compare with input number in DX
50:        jle     fib_loop       ; if not greater, find next number
51:        ; AX contains the required Fibonacci number
52:        pop     BX
53:        ret
54:  fibonacci  ENDP
55:        END     main
```

Example 10.8 *Fibonacci number computation using the stack for local variables.*
In this example, we use the stack for storing the two Fibonacci numbers. The variable `fib_lo` corresponds to fib($n-1$) and `fib_hi` to fib(n).

The code

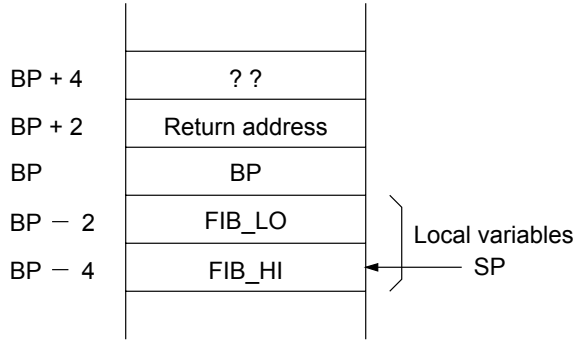
```
push    BP
mov     BP,SP
sub     SP,4
```

saves the BP value and copies the SP value into the BP as usual. It also decrements the SP by 4, thus creating four bytes of storage space for the two local variables `fib_lo` and `fib_hi`. This three-instruction sequence can be replaced by the

```
enter   4,0
```

instruction. As mentioned before, the first operand specifies the number of bytes reserved for local variables.

At this point, the stack allocation is



The two local variables can be accessed at BP $-$ 2 and BP $-$ 4. The two EQU statements, on lines 40 and 41, conveniently establish labels for these two locations. We can clear the local variable space and restore the BP value by

```
mov     SP,BP
pop     BP
```

instructions. The `leave` instruction performs exactly this. Thus, the `leave` instruction on line 59 automatically clears the local variable space. The rest of the code follows the logic of Example 10.7.

Program 10.8 Fibonacci number computation with local variables mapped to the stack

```
 1:  TITLE   Fibonacci numbers (stack version)    PROCFIB2.ASM
 2:  COMMENT |
 3:          Objective: To compute Fibonacci number using the stack
 4:                     for local variables.
 5:              Input: Requests a positive integer from the user.
 6:             Output: Outputs the largest Fibonacci number that
 7:  |                  is less than or equal to the input number.
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  prompt_msg   DB  'Please input a positive number (>1): ',0
12:  output_msg1  DB  'The largest Fibonacci number less than '
13:               DB  'or equal to ',0
14:  output_msg2  DB  ' is ',0
15:
16:  .CODE
17:  .486
18:  INCLUDE io.mac
19:
20:  main  PROC
21:        .STARTUP
22:        PutStr  prompt_msg     ; request input number
23:        GetInt  DX             ; DX := input number
24:        nwln
25:        call    fibonacci
26:        PutStr  output_msg1    ; print Fibonacci number
27:        PutInt  DX
28:        PutStr  output_msg2
29:        PutInt  AX
30:        nwln
31:  done:
32:        .EXIT
33:  main  ENDP
34:
35:  ;-----------------------------------------------------------
36:  ;Procedure fibonacci receives an integer in DX and computes
37:  ; the largest Fibonacci number that is less than the input
38:  ; number. The Fibonacci number is returned in AX.
39:  ;-----------------------------------------------------------
40:  FIB_LO   EQU  WORD PTR [BP-2]
41:  FIB_HI   EQU  WORD PTR [BP-4]
42:  fibonacci  PROC
43:        enter   4,0            ; space for two local variables
44:        push    BX
```

```
45:         ; FIB_LO maintains the smaller of the last two Fibonacci
46:         ;  numbers computed; FIB_HI maintains the larger one.
47:         mov     FIB_LO,1        ; initialize FIB_LO and FIB_HI to
48:         mov     FIB_HI,1        ;  first two Fibonacci numbers
49:   fib_loop:
50:         mov     AX,FIB_HI       ; compute next Fibonacci number
51:         mov     BX,FIB_LO
52:         add     BX,AX
53:         mov     FIB_LO,AX
54:         mov     FIB_HI,BX
55:         cmp     BX,DX           ; compare with input number in DX
56:         jle     fib_loop        ; if not greater, find next number
57:         ; AX contains the required Fibonacci number
58:         pop     BX
59:         leave                   ; clears local variable space
60:         ret
61:   fibonacci     ENDP
62:         END     main
```

10.11 Multiple Source Program Modules

In the program examples we have seen so far, the entire assembly language program is in a single file. This is fine for short example programs. Real application programs, however, tend to be large, consisting of hundreds of procedures. Rather than keeping such a massive source program in a single file, it is advantageous to break it into several small pieces, where each piece of source code is stored in a separate file or *module*. There are three advantages associated with multimodule programs:

- The chief advantage is that, after modifying a source module, it is only necessary to reassemble that module. On the other hand, if you keep only a single file, the whole file has to be reassembled.
- Making modifications to the source code is easier with several small files.
- It is safer to edit a short file; any unintended modifications to the source file are limited to a single small file.

If we want to separately assemble modules, we have to precisely specify the intermodule interface. For example, if a procedure is called in the current module but is defined in another module, we have to state that fact so that the assembler will not flag such procedure calls as errors. Assemblers provide two directives—`PUBLIC` and `EXTRN`—to facilitate separate assembly of source modules. These two directives are discussed in the following sections. A simple example follows this discussion.

10.11.1 PUBLIC Directive

The `PUBLIC` directive makes the associated label(s) public and therefore available to other modules of the program. The format is

```
PUBLIC    label1, label2, ...
```

Almost any label can be made public. These include procedure names, memory variables, and equated labels, as shown in the following example:

```
PUBLIC  error_msg, total, sample
                . . .
.DATA
error_msg       DB    ''Out of range!'',0
total           DW    0
                . . .
.CODE
                . . .
sample    PROC
                . . .
sample    ENDP
```

Note that when you make a label public, it is not necessary to specify the type of label.

10.11.2 EXTRN Directive

The `EXTRN` directive can be used to tell the assembler that certain labels are not defined in the current source file (i.e., module), but can be found in other modules. Thus, the assembler leaves "holes" in the corresponding `.obj` file that the linker will fill in later. The format is

```
EXTRN     label:type
```

where `label` is a label that is made public by a `PUBLIC` directive in some other module. The `type` specifies the type of label, some of which are listed in Table 10.2.

The `PROC` type should be used for procedure names if simplified segment directives such as `.MODEL` and `.STACK` are used. In this case, the appropriate procedure type is automatically included. For example, when the `.MODEL` is `SMALL`, the `PROC` type defaults to `NEAR` type. Assuming the labels `error_msg`, `total`, and `sample` are made public, as in the last example, the following example code makes them available in the current module:

```
.MODEL SMALL
         . . .
EXTRN    error_msg:BYTE, total:WORD
EXTRN    sample:PROC
         . . .
```

Note that the directive is spelled `EXTRN` (not `EXTERN`).

Table 10.2 Some example type specifiers

Type	Description
UNKNOWN	Undetermined or unknown type
BYTE	Data variable (size is 8 bits)
WORD	Data variable (size is 16 bits)
DWORD	Data variable (size is 32 bits)
QWORD	Data variable (size is 64 bits)
FWORD	Data variable (size is 6 bytes)
TBYTE	Data variable (size is 10 bytes)
PROC	A procedure name (Near or Far according to .MODEL)
NEAR	A near procedure name
FAR	A far procedure name

Example 10.9 *A two-module example to find string length.*
We now present a simple example that reads a string from the user and displays the string length (i.e., number of characters in the string). The source code consists of two procedures: `main` and `string_length`. The `main` procedure is responsible for requesting and displaying the string length information. It uses `GetStr`, `PutStr`, and `PutInt` I/O routines. The `string_length` procedure computes the string length. The entire source program is split between two modules: the `main` procedure is in the `module1.asm` file, and the procedure `string_length` is in the `module2.asm` file. A listing of `module1.asm` is given in Program 10.9. Notice that on line 16, we declare `string_length` as an externally defined procedure by using the `EXTRN` directive.

Program 10.9 The `main` procedure defined in `module1.asm` calls the `sum` procedure defined in `module2.asm`

```
 1:  TITLE   Multimodule program for string length   MODULE1.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via registers.
 4:              Input: Requests two integers from keyboard.
 5:  |          Output: Outputs the sum of the input integers.
 6:  BUF_SIZE  EQU  41   ; string buffer size
 7:  .MODEL SMALL
 8:  .STACK 100H
 9:  .DATA
10:  prompt_msg   DB   'Please input a string: ',0
11:  length_msg   DB   'String length is: ',0
```

```
12:  string1      DB   BUF_SIZE DUP (?)
13:
14:  .CODE
15:  INCLUDE io.mac
16:  EXTRN   string_length:PROC
17:  main  PROC
18:        .STARTUP
19:        PutStr  prompt_msg     ; request a string
20:        GetStr  string1,BUF_SIZE  ; read string input
21:        nwln
22:        mov     BX,OFFSET string1 ; BX := string pointer
23:        call    string_length  ; returns string length in AX
24:        PutStr  length_msg     ; display string length
25:        PutInt  AX
26:        nwln
27:  done:
28:        .EXIT
29:  main  ENDP
30:        END     main
```

Program 10.10 This module defines the `sum` procedure called by `main`

```
 1:  TITLE         String length procedure         MODULE2.ASM
 2:  COMMENT |
 3:          Objective: To write a procedure to compute string
 4:                     length of a NULL terminated string.
 5:              Input: String pointer in BX register.
 6:  |          Output: Returns string length in AX.
 7:  .MODEL SMALL
 8:  .CODE
 9:  PUBLIC string_length
10:  string_length PROC
11:        ; all registers except AX are preserved
12:        push    SI             ; save SI
13:        mov     SI,BX          ; SI := string pointer
14:  repeat:
15:        cmp     BYTE PTR [SI],0  ; is it NULL?
16:        je      done             ; if so, done
17:        inc     SI               ; else, move to next character
18:        jmp     repeat           ;       and repeat
19:  done:
20:        sub     SI,BX          ; compute string length
21:        mov     AX,SI          ; return string length in AX
```

```
22:         pop     SI              ; restore SI
23:         ret
24:  string_length ENDP
25:         END
```

Program 10.10 gives the `module2.asm` program listing. This module consists of a single procedure. By using the PUBLIC directive, we make this procedure public (line 9) so that other modules can use this procedure. The `string_length` procedure receives a pointer to a NULL-terminated string in BX and returns the length of the string in AX. The procedure preserves all registers except for AX. Note that the END statement (last statement) of this module does not have a label, as this is not the procedure that begins the program execution.

We can assemble each source code module separately producing the corresponding `.obj` files. We can then link the `.obj` files together to produce a single `.exe` file. For example, using the Turbo Assembler, the following sequence of commands will produce the executable file:

```
TASM    module1   ← Produces module1.obj
TASM    module2   ← Produces module2.obj
TLINK   module1+module2+io   ← Produces module1.exe
```

TASM, by default, assumes the `.asm` extension and TLINK assumes the `.obj` extension. The above sequence assumes that you have `io.obj` in your current directory. If you are using Microsoft Assembler, replace TASM with MASM and TLINK with LINK.

10.12 Summary

The stack is a last-in-first-out data structure that plays an important role in procedure invocation and execution. It supports two operations: push and pop. Only the element at the top-of-stack is directly accessible through these operations. The Pentium uses the stack segment to implement the stack. The top-of-stack is represented by SS:SP. In the Pentium implementation, the stack grows toward lower memory addresses (i.e., grows downward).

The stack serves three main purposes: temporary storage of data, transfer of control during a procedure call and return, and parameter passing.

When writing procedures in assembly language, parameter passing has to be explicitly handled. Parameter passing can be done via registers or the stack. Although the register method is efficient, the stack-based method is more general. Also, when the stack is used for parameter passing, handling a variable number of parameters is straightforward. We have demonstrated this by means of an example.

As with parameter passing, local variables of a procedure can be stored either in registers or on the stack. Due to the limited number of registers available, only a few local variables can be mapped to registers. The stack avoids this limitation, but it is slow.

Real application programs are unlikely to be short enough to keep in a single file. It is advantageous to break large source programs into more manageable chunks. Then we can keep each chunk in a separate file (i.e., modules). We have discussed how such multimodule programs are written and assembled into a single executable file.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Activation record
- Bubble sort
- Call-by-reference
- Call-by-value
- ENDP directive
- EXTRN directive
- FAR procedures
- Frame pointer
- Local variables
- NEAR procedures
- Parameter passing
- Parameter passing—register method
- Parameter passing—stack method
- PROC directive
- PUBLIC directive
- Segment override
- Stack frame
- Stack operations
- Stack overflow
- Stack underflow
- Top-of-stack
- Variable number of parameters

10.13 Exercises

10–1 What are the defining characteristics of a stack?

10–2 Discuss the differences between a queue and a stack.

10–3 What is top-of-stack? How is it represented in the Pentium?

10–4 What is stack underflow? Which stack operation can potentially cause stack underflow?

10–5 What is stack overflow? Which stack operation can potentially cause stack overflow?

10–6 What are the main uses of the stack?

10–7 In Section 10.4.1 on page 393, we have discussed two ways of exchanging `value1` and `value2`. Both methods require eight memory accesses. Can you write a code fragment that does this exchange using only six memory accesses? Make sure that your code does not alter the contents of any registers. *Hint:* Use the `xchg` instruction.

10–8 In the Pentium, can we invoke a procedure through the call instruction without the presence of a stack segment? Explain.

10–9 What is the main difference between a near procedure and a far procedure?

10–10 What are the two most common methods of parameter passing? Identify the circumstances under which each method is preferred.

10–11 What are the disadvantages of passing parameters via the stack?

10–12 Can we pass a variable number of parameters using the register parameter passing method? Explain the limitations and the problems associated with such a method.

10–13 We have stated on page 404 that placing the code

```
push    BP
mov     BP,SP
```

at the beginning of a procedure is good for program maintenance. Explain why.

10–14 In passing a variable number of parameters via the stack, why is it necessary to push the parameter count last?

10–15 Why are local variables of a procedure not mapped to the data segment?

10–16 How is storage space for local variables created in the stack?

10–17 A swap procedure can exchange two elements (pointed to by SI and DI) of an array using

```
xchg   AX,[DI]
xchg   AX,[SI]
xchg   AX,[DI]
```

The above code preserves the contents of the AX register. This code requires six memory accesses. Can we do better than this in terms of the number of memory accesses if we save and restore the AX using `push` and `pop` stack operations?

10–18 The bubble sort example discussed in this chapter used a single source file. In this exercise you are asked to split the source code of this program into two modules: the `main` procedure in one module, and the bubble sort procedure in the other. Then assemble and link this code to produce the `.exe` file. Verify the correctness of the program.

10–19 Verify that the following procedure is equivalent to the `string_length` procedure given in Section 10.11. Which procedure is better and why?

```
string_length1  PROC
        push    BX
        sub     AX,AX
repeat:
        cmp     BYTE PTR [BX],0
        je      done
        inc     AX
        inc     BX
        jmp     repeat
done:
        pop     BX
        ret
string_length1  ENDP
```

10.14 Programming Exercises

10–P1 Write an assembly language program that reads a set of integers from the keyboard and displays their sum on the screen. Your program should read up to 20 integers (except zero) from the user. The input can be terminated by entering a zero or by entering 20 integers. The array of input integers is passed along with its size to the `sum` procedure, which returns the sum in the AX register. Your `sum` procedure need not check for overflow.

10–P2 Write a procedure `max` that receives three integers from `main` and returns the maximum of the three in AX. The `main` procedure requests the three integers from the user and displays the maximum number returned by the `max` procedure.

10–P3 Extend the last exercise to return both maximum and minimum of the three integers received by your procedure `minmax`. In order to return the minimum and maximum values, your procedure `minmax` also receives two pointers from `main` to variables `min_int` and `max_int`.

10–P4 Extend the last exercise to handle a variable number of integers passed on to the `minmax` procedure. The `main` procedure should request input integers from the user. Positive or negative values, except zero, are valid. Entering a zero terminates the input integer sequence. The minimum and maximum values returned by the procedure are displayed by `main`.

10–P5 Write a procedure to perform string reversal. The procedure `reverse` receives a pointer to a character string (terminated by a NULL character) and reverses the string. For example, if the original string is

```
slap
```

the reversed string should read

```
pals
```

The `main` procedure should request the string from the user. It should also display the reversed string as output of the program.

10–P6 Write a procedure `locate` to locate a character in a given string. The procedure receives a pointer to a NULL-terminated character string and the character to be located. When the first occurrence of the character is located, its position is returned to `main`. If no match is found, a negative value is returned. The `main` procedure requests a character string and a character to be located and displays the position of the first occurrence of the character returned by the `locate` procedure. If there is no match, a message should be displayed to that effect.

10–P7 Write a procedure that receives a string via the stack (i.e., the string pointer is passed to the procedure) and removes all leading blank characters in the string. For example, if the input string is (␣ indicates a blank character)

␣␣␣␣␣Read␣␣my␣lips.

it will be modified by removing all leading blanks as

Read␣␣my␣lips.

10–P8 Write a procedure that receives a string via the stack (i.e., the string pointer is passed to the procedure) and removes all leading and duplicate blank characters in the string. For example, if the input string is (␣ indicates a blank character)

␣ ␣ ␣ ␣ ␣Read␣ ␣ ␣my␣ ␣ ␣ ␣ ␣lips.

it will be modified by removing all leading and duplicate blanks as

Read␣my␣lips.

10–P9 Write a program to read a number (consisting of up to 28 digits) and display the sum of the individual digits. Do not use `GetInt` to read the input number; read it as a sequence of characters. A sample input and output of the program is

Input: `123456789`
Output: `45`

10–P10 Write a procedure to read a string, representing a person's name, in the format

first-name␣MI␣last-name

and displays the name in the format

last-name,␣first-name␣MI

where ␣ indicates a blank character. As indicated, you can assume that the three names—first name, middle initial, and last name—are separated by single spaces.

10–P11 Modify the last exercise to work on an input that can contain multiple spaces between the names. Also, display the name as in the last exercise but with the last name in all capital letters.

Chapter 11

Addressing Modes

Objectives

- To discuss in detail various addressing modes supported by the Pentium;
- To illustrate the usefulness of these addressing modes in supporting high-level language features;
- To describe how arrays are implemented and manipulated in assembly language;
- To show how recursive procedures are written in assembly language.

In assembly language, specification of data required by instructions can be done in a variety of ways. In Chapter 9, we discussed four different addressing modes: register, immediate, direct, and indirect. The last two addressing modes specify operands in memory. However, operands located in memory can be specified by several other addressing modes. Section 11.2 describes these memory addressing modes in detail, and Section 11.3 gives examples to illustrate their use.

Arrays are important for organizing a collection of related data. Although one-dimensional arrays are straightforward to implement, multidimensional arrays are more involved. These issues are discussed in Section 11.4. Section 11.4.3 gives some examples to illustrate the use of addressing modes in processing one- and two-dimensional arrays.

Recursion is introduced in Section 11.5. We use a couple of examples to illustrate the principles involved in writing recursive procedures in assembly language.

11.1 Introduction

CISC processors support a large number of addressing modes compared to RISC processors. RISC processors use the load/store architecture. In this architecture, assembly language instructions take their operands from the processor registers and store the results in registers. This is

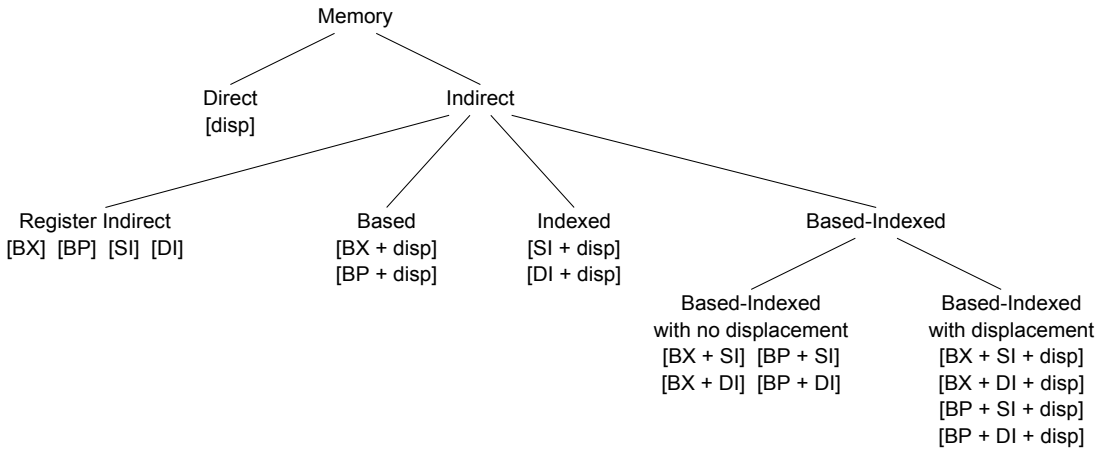


Figure 11.1 Memory addressing modes for 16-bit addresses.

what we called register addressing mode in Chapter 9. These processors use special load and store instructions to move data between registers and memory. As a result, RISC processors support very few (often just two) addressing modes.

The Pentium, being a CISC processor, provides several addressing modes. The three main ones are as follows:

- *Register Addressing Mode:* In this addressing mode, as discussed in Chapter 9, processor registers provide the input operands, and results are stored back in registers. Since the Pentium uses a two-address format, one operand specification acts as both source and destination. This addressing mode is the best way of specifying the operands, as the delay in accessing the operands is minimal.
- *Immediate Addressing Mode:* This addressing mode can be used to specify at most one source operand. The operand value is encoded as part of the instruction. Thus, the operand is available as soon as the instruction is read.
- *Memory Addressing Modes:* When an operand is in memory, the Pentium provides a variety of addressing modes to specify it. Recall that we have to specify the logical address in order to specify the location of a memory operand. The logical address consists of two components: segment base and offset. Note that offset is also referred to as the effective address. Memory addressing modes differ in how they specify the effective address.

We have already discussed the direct and register indirect addressing modes in Chapter 9. The direct addressing mode gives the effective address directly in the instruction. In the indirect addressing mode, the effective address is in one of the general-purpose registers. This chapter discusses the remaining memory addressing modes.

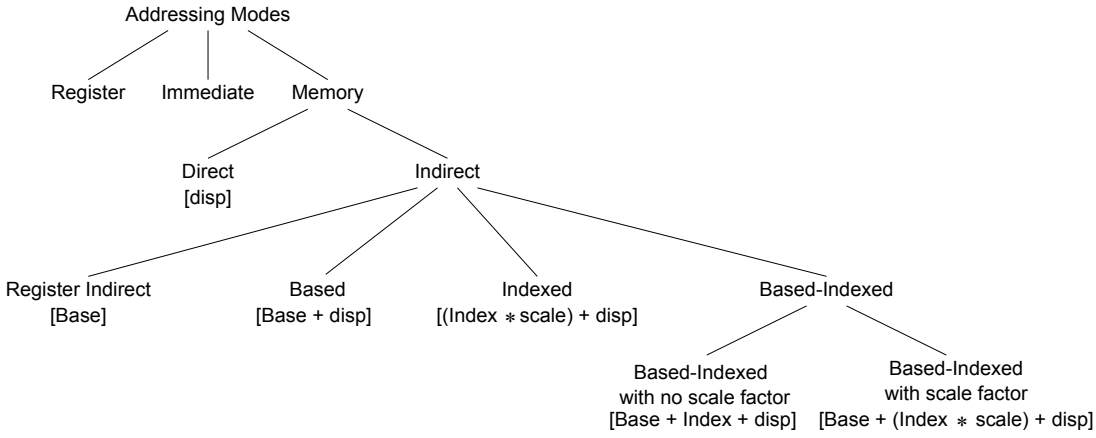


Figure 11.2 Addressing modes of the Pentium for 32-bit addresses.

11.2 Memory Addressing Modes

The primary motivation for providing different addressing modes is to efficiently support high-level language constructs and data structures. The actual memory addressing modes available depend on the address size used (16 bits or 32 bits). The memory addressing modes available for 16-bit addresses are the same as those supported by the 8086. Figure 11.1 shows the default memory addressing modes available for 16-bit addresses. The Pentium supports a more flexible set of addressing modes for 32-bit addresses. These addressing modes are shown in Figure 11.2 and are summarized below:

Segment + Base + (Index * Scale) + displacement

CS	EAX	EAX	1	No displacement
SS	EBX	EBX	2	8-bit displacement
DS	ECX	ECX	4	32-bit displacement
ES	EDX	EDX	8	
FS	ESI	ESI		
GS	EDI	EDI		
	EBP	EBP		
	ESP			

The differences between 16-bit and 32-bit addressing are summarized in Table 11.1. How does the processor know whether to use 16- or 32-bit addressing? As discussed in Chapter 7, it uses the D bit in the CS segment descriptor to determine if the address is 16 or 32 bits long (see page 267). It is, however, possible to override these defaults. The Pentium provides two size override prefixes:

66H	Operand size override prefix
67H	Address size override prefix

Table 11.1 Differences between 16-bit and 32-bit addressing

	16-bit addressing	32-bit addressing
Base register	BX BP	EAX, EBX, ECX, EDX ESI, EDI, EBP, ESP
Index register	SI DI	EAX, EBX, ECX, EDX ESI, EDI, EBP
Scale factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bits	0, 8, 32 bits

By using these prefixes, we can mix 16- and 32-bit data and addresses. Remember that our assembly language programs use 16-bit data and addresses. This, however, does not restrict us from using 32-bit data and addresses. For example, when we write

```
mov     AX,123
```

the assembler generates the following machine language code:

```
B8 007B
```

However, when we use a 32-bit operand, as in

```
mov     EAX,123
```

the following code is generated by the assembler:

```
66 | B8 0000007B
```

The assembler automatically inserts the operand size override prefix (66H).

The greatest benefit of the address size override prefix is that we can use all the addressing modes provided for 32-bit addresses in the 16-bit addressing modes. For instance, we can use a scale factor, as in the following example:

```
mov     AX,[EBX+ESI*2]
```

The assembler automatically inserts the address size override prefix (67H) as shown below:

```
67 | 8B 04 73
```

It is also possible to mix both override prefixes as demonstrated by the following example. The assembly language statement

```
mov     EAX,[EBX+ESI*2]
```

causes the assembler to insert both operand and address size override prefixes:

```
66 | 67 | 8B 04 73
```

Remember that with 16-bit addresses, our segments are limited to 64 KB. Even though we have used 32-bit registers EBX and ESI in the last two examples, offsets into the segment are still limited to 64 KB (i.e., offset should be less than or equal to FFFFH). The processor generates a general protection fault if this value is exceeded. In summary, the address size prefix only allows us to use the additional addressing modes of the Pentium with 16-bit addresses.

11.2.1 Based Addressing

In the based addressing mode, one of the registers acts as the base register in computing the effective address of an operand. The effective address is computed by adding the contents of the specified base register with a signed displacement value given as part of the instruction. For 16-bit addresses, the signed displacement is either an 8- or a 16-bit number. For 32-bit addresses, it is either an 8- or a 32-bit number.

Based addressing provides a convenient way to access individual elements of a structure. Typically, a base register can be set up to point to the base of the structure and the displacement can be used to access an element within the structure. For example, consider the following record of a course schedule:

Course number	Integer	2 bytes
Course title	Character string	38 bytes
Term offered	Single character	1 byte
Room number	Character string	5 bytes
Enrollment limit	Integer	2 bytes
Number registered	Integer	2 bytes
Total storage per record		50 bytes

In this example, suppose we want to find the number of available spaces in a particular course. We can let the BX register point to the base address of the corresponding course record and use displacement to read the number of students registered and the enrollment limit for the course to compute the desired answer. This is illustrated in Figure 11.3.

This addressing mode is also useful in accessing arrays whose element size is not 2, 4, or 8 bytes. In this case, the displacement can be set equal to the offset to the beginning of the array, and the base register holds the offset of a specific element relative to the beginning of the array.

11.2.2 Indexed Addressing

In this addressing mode, the effective address is computed as

$$(\text{Index} * \text{scale factor}) + \text{signed displacement}.$$

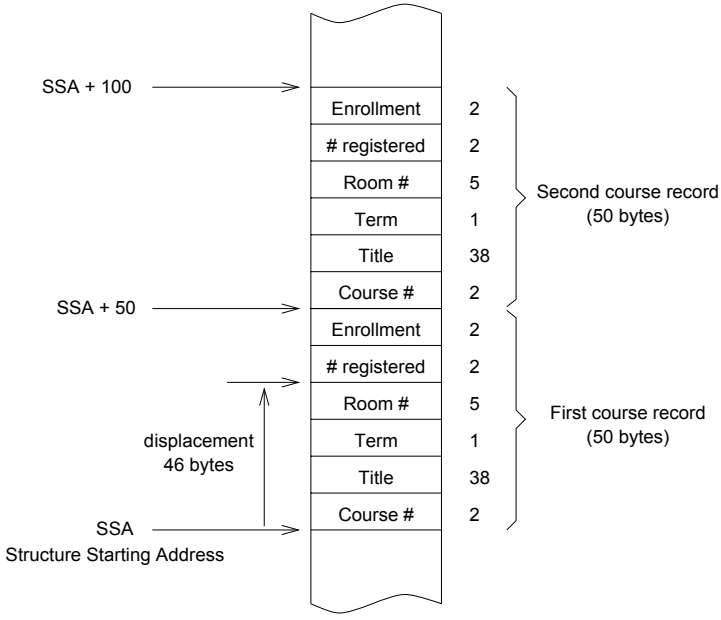


Figure 11.3 Course record layout in memory.

For 16-bit addresses, no scaling factor is allowed (see Table 11.1 on page 438). For 32-bit addresses, a scale factor of 2, 4, or 8 can be specified. Of course, we can use a scale factor in the 16-bit addressing mode by using an address size override prefix.

The indexed addressing mode is often used to access elements of an array. The beginning of the array is given by the displacement, and the value of the index register selects an element within the array. The scale factor is particularly useful to access arrays of elements whose size is 2, 4, or 8 bytes.

The following are valid instructions using the indexed addressing mode to specify one of the operands.

```
add     AX,[DI+20]
mov     AX,marks_table[ESI*4]
add     AX,table1[SI]
```

In the second instruction, the assembler would supply a constant displacement that represents the offset of `marks_table` in the data segment. Assume that each element of `marks_table` takes four bytes. Since we are using a scale factor of four, ESI should have the index value. For example, if we want to access the tenth element, ESI should have nine as the index value starts with zero.

If no scale factor is used as in the last instruction, SI should hold the offset of the element in *bytes* relative to the beginning of the array. For example, if `table1` is an array of four-byte

elements, SI register should have 36 to refer to the tenth element. By using the scale factor, we avoid such byte counting.

11.2.3 Based-Indexed Addressing

Based-Indexed with No Scale Factor

In this addressing mode, the effective address is computed as

Base + Index + signed displacement.

The displacement can be a signed 8- or 16-bit number for 16-bit addresses; it can be a signed 8- or 32-bit number for 32-bit addresses.

This addressing mode is useful in accessing two-dimensional arrays with the displacement representing the offset to the beginning of the array. This mode can also be used to access arrays of records where the displacement represents the offset to a field in a record. In addition, this addressing mode is used to access arrays passed on to a procedure. In this case, the base register could point to the beginning of the array, and an index register can hold the offset to a specific element.

Assuming that BX points to `table1`, which consists of four-byte elements, we can use the code

```
mov     AX,[BX+SI]
cmp     AX,[BX+SI+4]
```

to compare two successive elements of `table1`. This type of code is particularly useful if the `table1` pointer is passed as a parameter.

Based-Indexed with Scale Factor

In this addressing mode, the effective address is computed as

Base + (Index $*$ scale factor) + signed displacement.

This addressing mode provides an efficient indexing mechanism into a two-dimensional array when the element size is 2, 4, or 8 bytes.

11.3 Illustrative Examples

We now present two examples to illustrate the usefulness of the various addressing modes. The first example sorts an array of integers using the insertion sort algorithm, and the other example implements a binary search to locate a value in a sorted array. Example 11.1 uses only the 16-bit addressing modes (see Figure 11.1), whereas Example 11.2 uses both 16-bit and 32-bit addressing modes.

Example 11.1 *Sorting an integer array using the insertion sort.*
This example requests a set of integers from the user and displays these numbers in sorted order. The main procedure reads a maximum of `MAX_SIZE` integers (lines 23 to 30). It accepts only nonnegative numbers. Entering a negative number terminates the input (lines 26 and 27).

The main procedure passes the array pointer and its size (lines 32 to 36) to the insertion sort procedure. The remainder of the main procedure displays the sorted array returned by the sort procedure. Note that the main procedure uses the indirect addressing mode on lines 28 and 43.

There are several sorting algorithms to sort an array of numbers. Here we use the insertion sort algorithm. We discuss another sort algorithm later (see Example 11.6 on page 458). The basic principle behind the insertion sort is simple: insert a new number into the sorted array in its proper place. To apply this algorithm, we start with an empty array. Then insert the first number. Now the array is in sorted order with just one element. Next insert the second number in its proper place. This results in a sorted array of size two. Repeat this process until all the numbers are inserted. The pseudocode for this algorithm, shown below, assumes that the array index starts with 0:

```
insertion_sort (array, size)
    for (i = 1 to size−1)
        temp := array[i]
        j := i − 1
        while ((temp < array[j]) AND (j ≥ 0))
            array[j+1] := array[j]
            j := j − 1
        end while
        array[j+1] := temp
    end for
end insertion_sort
```

Here, index *i* points to the number to be inserted. The array to the left of *i* is in sorted order. The numbers to be inserted are the ones located at or to the right of index *i*. The next number to be inserted is at *i*. The implementation of the insertion sort procedure, shown in Program 11.1, follows the pseudocode.

Program 11.1 Insertion sort

```
1:  TITLE    Sorting an array by insertion sort    INS_SORT.ASM
2:  COMMENT |
3:          Objective: To sort an integer array using insertion sort.
4:              Input: Requests numbers to fill array.
5:  |           Output: Displays sorted array.
6:  .MODEL SMALL
7:  .STACK 100H
8:  .DATA
9:  MAX_SIZE       EQU 100
```

```
10:  array          DW  MAX_SIZE DUP (?)
11:  input_prompt   DB  'Please enter input array: '
12:                 DB  '(negative number terminates input)',0
13:  out_msg        DB  'The sorted array is:',0
14:
15:  .CODE
16:  .486
17:  INCLUDE io.mac
18:  main    PROC
19:          .STARTUP
20:          PutStr  input_prompt ; request input array
21:          mov     BX,OFFSET array
22:          mov     CX,MAX_SIZE
23:  array_loop:
24:          GetInt  AX           ; read an array number
25:          nwln
26:          cmp     AX,0         ; negative number?
27:          jl      exit_loop    ; if so, stop reading numbers
28:          mov     [BX],AX      ; otherwise, copy into array
29:          add     BX,2         ; increment array address
30:          loop    array_loop   ; iterates a maximum of MAX_SIZE
31:  exit_loop:
32:          mov     DX,BX        ; DX keeps the actual array size
33:          sub     DX,OFFSET array  ; DX := array size in bytes
34:          shr     DX,1         ; divide by 2 to get array size
35:          push    DX           ; push array size & array pointer
36:          push    OFFSET array
37:          call    insertion_sort
38:          PutStr  out_msg      ; display sorted array
39:          nwln
40:          mov     CX,DX
41:          mov     BX,OFFSET array
42:  display_loop:
43:          PutInt  [BX]
44:          nwln
45:          add     BX,2
46:          loop    display_loop
47:  done:
48:          .EXIT
49:  main    ENDP
50:
51:  ;---------------------------------------------------------
52:  ; This procedure receives a pointer to an array of integers
53:  ; and the array size via the stack. The array is sorted by
54:  ; using insertion sort. All registers are preserved.
```

```
55:  ;-----------------------------------------------------------
56:  SORT_ARRAY  EQU  [BX]
57:  insertion_sort PROC
58:         pusha                   ; save registers
59:         mov     BP,SP
60:         mov     BX,[BP+18]      ; copy array pointer
61:         mov     CX,[BP+20]      ; copy array size
62:         mov     SI,2            ; array left of SI is sorted
63:  for_loop:
64:         ; variables of the algorithm are mapped as follows.
65:         ; DX = temp, SI = i, and DI = j
66:         mov     DX,SORT_ARRAY[SI] ; temp := array[i]
67:         mov     DI,SI           ; j := i-1
68:         sub     DI,2
69:  while_loop:
70:         cmp     DX,SORT_ARRAY[DI]  ; temp < array[j]
71:         jge     exit_while_loop
72:         ; array[j+1] := array[j]
73:         mov     AX,SORT_ARRAY[DI]
74:         mov     SORT_ARRAY[DI+2],AX
75:         sub     DI,2            ; j := j-1
76:         cmp     DI,0            ; j >= 0
77:         jge     while_loop
78:  exit_while_loop:
79:         ; array[j+1] := temp
80:         mov     SORT_ARRAY[DI+2],DX
81:         add     SI,2            ; i := i+1
82:         dec     CX
83:         cmp     CX,1            ; if CX = 1, we are done
84:         jne     for_loop
85:  sort_done:
86:         popa                    ; restore registers
87:         ret     4
88:  insertion_sort ENDP
89:         END     main
```

Since the sort procedure does not return any value to the main program in registers, we can use `pusha` (line 58) and `popa` (line 86) to save and restore registers. As `pusha` saves all eight 16-bit registers on the stack, the offset is appropriately adjusted to access the array size and array pointer parameters (lines 60 and 61).

The `while` loop is implemented by lines 69 to 78, and the `for` loop is implemented by lines 63 to 84. Note that the array pointer is copied to the BX (line 60), and line 56 assigns a convenient label to this. We have used the based-indexed addressing mode on lines 66, 70, and

73 without any displacement and on lines 74 and 80 with displacement. Based addressing is used on lines 60 and 61 to access parameters from the stack.

Example 11.2 *Binary search procedure.*
Binary search is an efficient algorithm to locate a value in a sorted array. The search process starts with the whole array. The value at the middle of the array is compared with the number we are looking for: if there is a match, its index is returned. Otherwise, the search process is repeated either on the lower half (if the number is less than the value at the middle), or on the upper half (if the number is greater than the value at the middle). The pseudocode of the algorithm is given below:

```
binary_search (array, size, number)
    lower := 0
    upper := size − 1
    while (lower ≤ upper)
        middle := (lower + upper)/2
        if (number = array[middle])
        then
            return (middle)
        else
            if (number < array[middle])
            then
                upper := middle − 1
            else
                lower := middle + 1
            end if
        end if
    end while
    return (0)      {number not found}
    end binary_search
```

The listing of the binary search program is given in Program 11.2. The main procedure is similar to that in the last example. The lower and upper index variables are mapped to the AX and CX registers. The number to be searched is stored in the DX, and the array pointer is in the BX. Register SI keeps the middle index value.

Program 11.2 Binary search

```
1:  TITLE   Binary search of a sorted integer array   BIN_SRCH.ASM
2:  COMMENT |
3:          Objective: To implement binary search of a sorted
4:                     integer array.
5:              Input: Requests numbers to fill array and a
```

```
 6:                      number to be searched for from user.
 7:              Output: Displays the position of the number in
 8:                      the array if found; otherwise, not found
 9:  |                   message.
10:  .MODEL SMALL
11:  .STACK 100H
12:  .DATA
13:  MAX_SIZE       EQU 100
14:  array          DW  MAX_SIZE DUP (?)
15:  input_prompt   DB  'Please enter input array (in sorted order): '
16:                 DB  '(negative number terminates input)',0
17:  query_number   DB  'Enter the number to be searched: ',0
18:  out_msg        DB  'The number is at position ',0
19:  not_found_msg  DB  'Number not in the array!',0
20:  query_msg      DB  'Do you want to quit (Y/N): ',0
21:
22:  .CODE
23:  .486
24:  INCLUDE io.mac
25:  main    PROC
26:          .STARTUP
27:          PutStr  input_prompt ; request input array
28:          nwln
29:          sub     ESI,ESI      ; set index to zero
30:          mov     CX,MAX_SIZE
31:  array_loop:
32:          GetInt  AX           ; read an array number
33:          nwln
34:          cmp     AX,0            ; negative number?
35:          jl      exit_loop       ; if so, stop reading numbers
36:          mov     array[ESI*2],AX ; otherwise, copy into array
37:          inc     SI           ; increment array index
38:          loop    array_loop   ; iterates a maximum of MAX_SIZE
39:  exit_loop:
40:  read_input:
41:          PutStr  query_number ; request number to be searched for
42:          GetInt  AX           ; read the number
43:          nwln
44:          push    AX           ; push number, size & array pointer
45:          push    SI
46:          push    OFFSET array
47:          call    binary_search
48:          ; binary_search returns in AX the position of the number
49:          ; in the array; if not found, it returns 0.
50:          cmp     AX,0         ; number found?
```

```
51:          je      not_found    ; if not, display number not found
52:          PutStr  out_msg      ; else, display number position
53:          PutInt  AX
54:          jmp     user_query
55:  not_found:
56:          PutStr  not_found_msg
57:  user_query:
58:          nwln
59:          PutStr  query_msg    ; query user whether to terminate
60:          GetCh   AL           ; read response
61:          nwln
62:          cmp     AL,'Y'       ; if response is not 'Y'
63:          jne     read_input   ; repeat the loop
64:  done:                        ; otherwise, terminate program
65:          .EXIT
66:  main    ENDP
67:
68:  ;-----------------------------------------------------------
69:  ; This procedure receives a pointer to an array of integers,
70:  ; the array size, and a number to be searched via the stack.
71:  ; It returns in AX the position of the number in the array
72:  ; if found; otherwise, returns 0.
73:  ; All registers, except AX, are preserved.
74:  ;-----------------------------------------------------------
75:  binary_search  PROC
76:         enter   0,0
77:         push    EBX
78:         push    ESI
79:         push    CX
80:         push    DX
81:         xor     EBX,EBX       ; EBX = 0
82:         mov     BX,[BP+4]     ; copy array pointer
83:         mov     CX,[BP+6]     ; copy array size
84:         mov     DX,[BP+8]     ; copy number to be searched
85:         xor     AX,AX         ; lower = 0
86:         dec     CX            ; upper = size-1
87:  while_loop:
88:         cmp     AX,CX         ;lower > upper?
89:         ja      end_while
90:         sub     ESI,ESI
91:         mov     SI,AX         ; middle = (lower + upper)/2
92:         add     SI,CX
93:         shr     SI,1
94:         cmp     DX,[EBX+ESI*2]    ; number = array[middle]?
95:         je      search_done
```

```
 96:         jg      upper_half
 97:  lower_half:
 98:         dec     SI              ; middle = middle-1
 99:         mov     CX,SI           ; upper = middle-1
100:         jmp     while_loop
101:  upper_half:
102:         inc     SI              ; middle = middle+1
103:         mov     AX,SI           ; lower = middle+1
104:         jmp     while_loop
105:  end_while:
106:         sub     AX,AX           ; number not found (clear AX)
107:         jmp     skip1
108:  search_done:
109:         inc     SI              ; position = index+1
110:         mov     AX,SI           ; return position
111:  skip1:
112:         pop     DX              ; restore registers
113:         pop     CX
114:         pop     ESI
115:         pop     EBX
116:         leave
117:         ret     6
118:  binary_search  ENDP
119:         END     main
```

Since the binary search procedure returns a value in the AX register, we cannot use the `pusha` instruction as in the last example. This example also demonstrates how some of the 32-bit addressing modes can be used with 16-bit segments. For example, on line 94, we use a scale factor of two to convert the index value in SI to byte count. Also, a single comparison (line 94) is sufficient to test multiple conditions (i.e., equal to, greater than, or less than). If the number is found in the array, the index value in SI is returned via AX (line 110).

11.4 Arrays

Arrays are useful in organizing a collection of related data items, such as test marks of a class, salaries of employees, and so on. We have used arrays of characters to represent strings. Such arrays are one-dimensional: only a single subscript is necessary to access a character in the array. Next we discuss one-dimensional arrays. High-level languages support multidimensional arrays. Multidimensional arrays are discussed in Section 11.4.2.

11.4.1 One-Dimensional Arrays

A one-dimensional array of test marks can be declared in C as

```
int    test_marks [10];
```

In C, the subscript always starts at zero. Thus, the mark of the first student is given by `test_marks[0]` and that of the last student by `test_marks[9]`.

Array declaration in high-level languages specifies the following five attributes:

- Name of the array (`test_marks`),
- Number of the elements (10),
- Element size (2 bytes),
- Type of element (integer), and
- Index range (0 to 9).

From this information, the amount of storage space required for the array can be easily calculated. Storage space in bytes is given by

$$\text{Storage space} = \text{number of elements} * \text{element size in bytes.}$$

In our example, it is equal to $10 * 2 = 20$ bytes. In assembly language, arrays are implemented by allocating the required amount of storage space. For example, the `test_marks` array can be declared as

```
test_marks    DW    10 DUP (?)
```

An array name can be assigned to this storage space. But that is all the support you get in assembly language! It is up to you as a programmer to "properly" access the array taking into account the element size and the range of subscripts.

You need to know how the array is stored in memory in order to access elements of the array. For one-dimensional arrays, representation of the array in memory is rather direct: array elements are stored linearly in the same order as shown in Figure 11.4. In the remainder of this section, we use the convention used for arrays in C (i.e., subscripts are assumed to begin with 0).

To access an element we need to know its displacement value in bytes relative to the beginning of the array. Since we know the element size in bytes, it is rather straightforward to compute the displacement from the subscript value:

$$\text{displacement} = \text{subscript} * \text{element size in bytes.}$$

For example, to access the sixth student's mark (i.e., subscript is 5), you have to use $5 * 2 = 10$ as the displacement value into the `test_marks` array. Section 11.4.3 presents an example that computes the sum of a one-dimensional integer array. If the array element size is 2, 4, or 8 bytes, we can use the scale factor to avoid computing displacement in bytes.

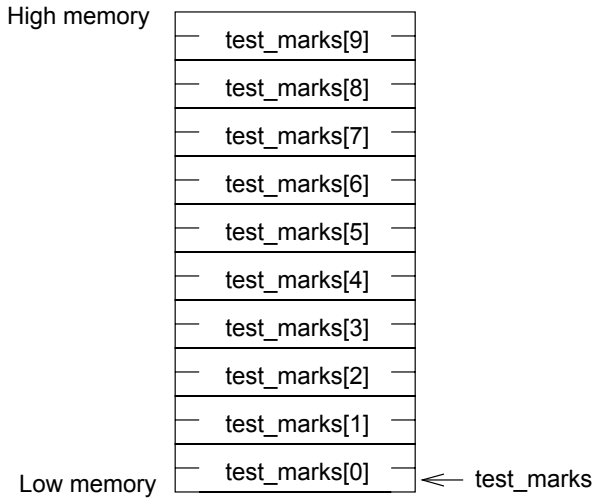


Figure 11.4 One-dimensional array storage representation.

11.4.2 Multidimensional Arrays

Programs often require arrays of more than one dimension. For example, we need a two-dimensional array of size 50 × 3 to store test marks of a class of 50 students taking three tests during a semester. For most programs, arrays of up to three dimensions are adequate. In this section, we discuss how two-dimensional arrays are represented and manipulated in assembly language. Our discussion can be generalized to higher-dimensional arrays.

For example, a 5 × 3 array to store test marks can be declared in C as

```
int     class_marks[5][3];    /* 5 rows and 3 columns */
```

Storage representation of such arrays is not as direct as that for one-dimensional arrays. Since the memory is one-dimensional (i.e., linear array of bytes), we need to transform the two-dimensional structure to a one-dimensional structure. This transformation can be done in one of two common ways:

- Order the array elements row-by-row, starting with the first row,
- Order the array elements column-by-column, starting with the first column.

The first method, called the *row-major ordering*, is shown in Figure 11.5*a*. Row-major ordering is used in most high-level languages including C and Pascal. The other method, called the *column-major ordering*, is shown in Figure 11.5*b*. Column-major ordering is used in FORTRAN. In the remainder of this section, we focus on the row-major ordering scheme.

Why do we need to know the underlying storage representation? When we are using a high-level language, we really do not have to bother about the storage representation. Access

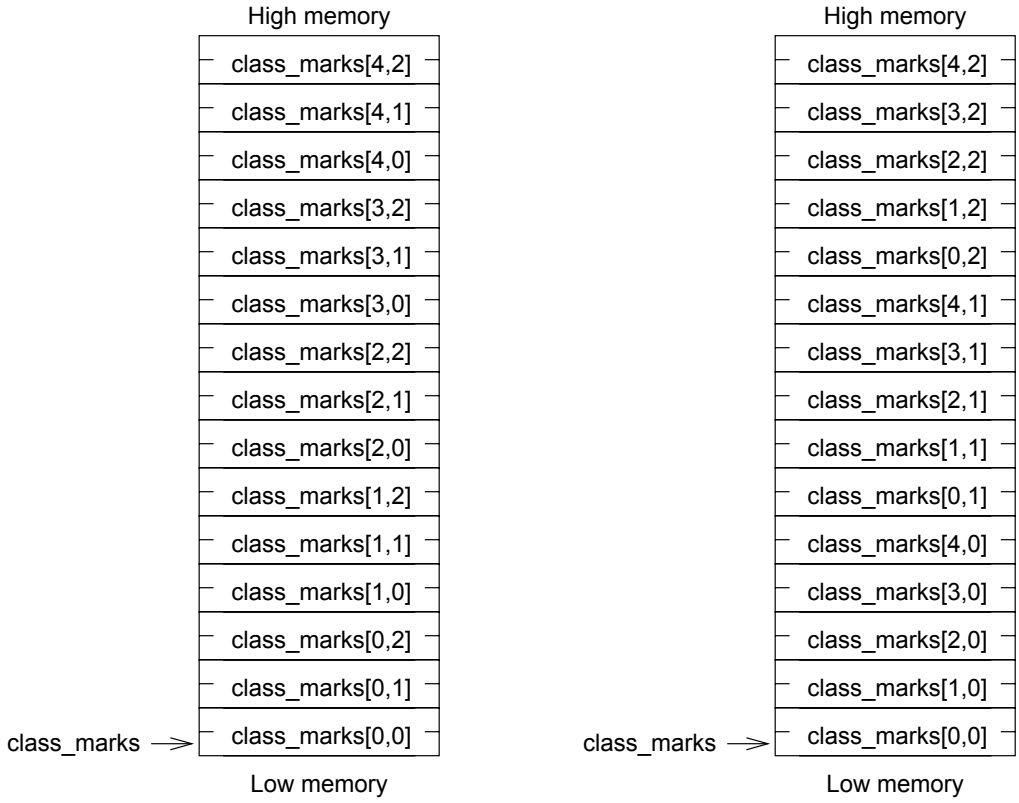


Figure 11.5 Two-dimensional array storage representation.

to arrays is provided by subscripts: one subscript for each dimension of the array. However, when using assembly language, we need to know the storage representation in order to access individual elements of the array for reasons discussed next.

In assembly language, we can allocate storage space for the `class_marks` array as

```
class_marks     DW     5*3 DUP (?)
```

This statement simply allocates the 30 bytes required to store the array. Now we need a formula to translate row and column subscripts to the corresponding displacement. In C language, which uses row-major ordering and subscripts start with zero, we can express displacement of an element at row i and column j as

$$\text{displacement} = (i * \text{COLUMNS} + j) * \text{ELEMENT\_SIZE},$$

where COLUMNS is the number of columns in the array and ELEMENT_SIZE is the number

of bytes required to store an element. For example, displacement of `class_marks[3,1]` is $(3 * 3 + 1) * 2 = 20$. The next section gives an example to illustrate how two-dimensional arrays are manipulated.

11.4.3 Examples of Arrays

This section presents two examples to illustrate manipulation of one- and two-dimensional arrays. These examples also demonstrate the use of advanced addressing modes in accessing multidimensional arrays.

Example 11.3 *Finding the sum of a one-dimensional array.*
This example shows how one-dimensional arrays can be manipulated. Program 11.3 finds the sum of the `test_marks` array and displays the result.

Program 11.3 Computing the sum of a one-dimensional array

```
 1:  TITLE       Sum of a long integer array       ARAY_SUM.ASM
 2:  COMMENT |
 3:          Objective: To find sum of all elements of an array.
 4:              Input: None.
 5:  |          Output: Displays the sum.
 6:  .MODEL SMALL
 7:  .STACK 100H
 8:  .DATA
 9:  test_marks     DD  90,50,70,94,81,40,67,55,60,73
10:  NO_STUDENTS    EQU ($-test_marks)/4     ; number of students
11:  sum_msg        DB  'The sum of test marks is: ',0
12:
13:  .CODE
14:  .486
15:  INCLUDE io.mac
16:  main    PROC
17:          .STARTUP
18:          mov     CX,NO_STUDENTS   ; loop iteration count
19:          sub     EAX,EAX          ; sum := 0
20:          sub     ESI,ESI          ; array index := 0
21:  add_loop:
22:          mov     EBX,test_marks[ESI*4]
23:          PutLInt EBX
24:          nwln
25:          add     EAX,test_marks[ESI*4]
26:          inc     ESI
27:          loop    add_loop
28:
29:          PutStr  sum_msg
```

```
30:          PutLInt EAX
31:          nwln
32:          .EXIT
33:  main    ENDP
34:          END     main
```

Each element of the `test_marks` array, declared on line 9, requires four bytes. The array size `NO_STUDENTS` is computed on line 10 using the predefined location counter symbol $. The predefined symbol $ is always set to the current offset in the segment. Thus, on line 10, $ points to the byte after the array storage space. Therefore, `($-test_marks)` gives the storage space in bytes and dividing this by four gives the number of elements in the array. We are using the indexed addressing mode with a scale factor of four on lines 22 and 25. Remember that the scale factor is only allowed in the 32-bit mode. As a result, we have to use ESI rather than the SI register.

Example 11.4 *Finding the sum of a column in a two-dimensional array.*
Consider the `class_marks` array representing the test scores of a class. For simplicity, assume that there are only five students in the class. Also, assume that the class is given three tests. As we have discussed before, we can use a 5×3 array to store the marks. Each row represents the three test marks of a student in the class. The first column represents the marks of the first test, the second column represents the marks of the second test, and so on. The objective of this example is to find the sum of the last test marks for the class. The program listing is given in Program 11.4.

Program 11.4 Finding the sum of a column in a two-dimensional array

```
 1:  TITLE  Sum of a column in a 2-dimensional array  TEST_SUM.ASM
 2:  COMMENT |
 3:           Objective: To demonstrate array index manipulation
 4:                      in a two-dimensional array of integers.
 5:               Input: None.
 6:  |           Output: Displays the sum.
 7:  .MODEL SMALL
 8:  .STACK 100H
 9:  .DATA
10:  NO_ROWS         EQU  5
11:  NO_COLUMNS      EQU  3
12:  NO_ROW_BYTES    EQU  NO_COLUMNS * 2  ; number of bytes per row
13:  class_marks     DW   90,89,99
14:                  DW   79,66,70
15:                  DW   70,60,77
16:                  DW   60,55,68
17:                  DW   51,59,57
```

```
18:
19:  sum_msg        DB   'The sum of the last test marks is: ',0
20:
21:  .CODE
22:  .486
23:  INCLUDE io.mac
24:  main    PROC
25:          .STARTUP
26:          mov     CX,NO_ROWS   ; loop iteration count
27:          sub     AX,AX        ; sum = 0
28:          ; ESI = index of class_marks[0,2]
29:          sub     EBX,EBX
30:          mov     ESI,NO_COLUMNS-1
31:  sum_loop:
32:          add     AX,class_marks[EBX+ESI*2]
33:          add     EBX,NO_ROW_BYTES
34:          loop    sum_loop
35:
36:          PutStr  sum_msg
37:          PutInt  AX
38:          nwln
39:  done:
40:          .EXIT
41:  main    ENDP
42:          END     main
```

To access individual test marks, we use based-indexed addressing with a displacement on line 32. Note that even though we have used

```
class_marks[EBX+ESI*2]
```

it is translated by the assembler as

```
[EBX+(ESI*2)+constant]
```

where the `constant` is the offset of `class_marks`. For this to work, the EBX should store the offset of the row in which we are interested. For this reason, after initializing the EBX to zero to point to the first row (line 29), `NO_ROW_BYTES` is added in the loop body (line 33). The ESI register is used as the column index. This works for row-major ordering.

11.5 Recursion

We have seen how procedures can be implemented in the Pentium assembly language. We now look at recursive procedures. A recursive procedure calls itself, either directly or indirectly. In direct recursion, procedure P makes another call to itself. In indirect recursion, procedure P makes a call to procedure Q, which in turn calls procedure P. The chain of calls could be longer before a call is made to procedure P.

Recursion is a powerful tool that allows us to express our solution elegantly. Some applications can be naturally expressed using recursion. Computing a factorial is a classic example. Factorial n, denoted $n!$, is the product of positive integers from 1 to n. For example,

$$5! = 1 \times 2 \times 3 \times 4 \times 5.$$

The factorial can be formally defined as

$$\begin{aligned} &\text{factorial}(0) = 1, \\ &\text{factorial}(n) = n * \text{factorial}(n-1) \text{ for } n > 0. \end{aligned}$$

Recursion shows up in this definition as we define factorial(n) in terms of factorial($n-1$). Every recursive function should have a termination condition to end recursion. In this example, when $n = 0$, recursion stops. How do we express such recursive functions in programming languages? Let us first look at how this function is written in C:

```
int fact(int n)
{
    if (n == 0)
        return(1);
    return(n * fact(n-1));
}
```

This is an example of direct recursion. How is this function implemented? At the conceptual level, its implementation is not any different from implementing other procedures. Once you understand that each procedure call instance is distinct from the others, the fact that a recursive procedure calls itself does not make a big difference.

Each active procedure maintains an activation record, which is stored on the stack. The activation record, as explained on page 421, consists of the arguments, return address, and local variables. The activation record comes into existence when a procedure is invoked and disappears after the procedure is terminated. Thus, for each procedure that is not terminated, an activation record that contains the state of that procedure is stored. The number of activation records, and hence the amount of stack space required to run the program, depends on the depth of recursion.

Figure 11.6 shows the stack activation records for factorial(3). As you can see from this figure, each call to the factorial function creates an activation record.

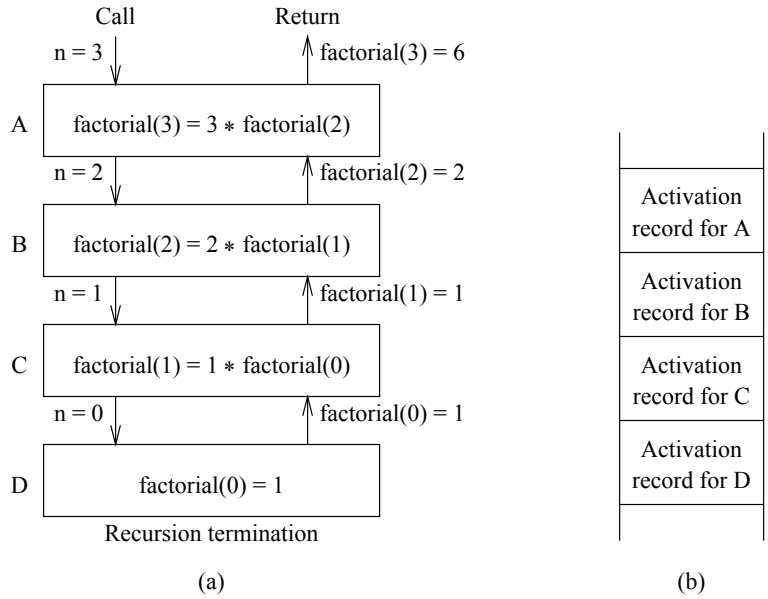


Figure 11.6 Recursive computation of factorial(3).

11.5.1 Illustrative Examples

To illustrate the principles of recursion, we give two examples. The first computes the factorial function. The second example implements the popular quicksort algorithm.

Example 11.5 *Recursive procedure to compute the factorial function.*
An implantation of the factorial function is shown in Program 11.5. The main function provides the user interface. It requests a positive number from the user. If a negative number is given as input, the user is prompted to try again. The positive number, which is read into the BX, is passed on to procedure `fact`.

The `fact` procedure receives the number n in the BL register. It essentially implements the C code given before. One minor difference is that this procedure terminates when $n \leq 1$. This termination would save one recursive call. When the BL is less than or equal to 1, the AX register is set to 1 to terminate recursion. The activation record in this example consists of the return address pushed onto the stack by the `call` instruction. Since we are using the BL register, it is decremented before the call (line 53) and restored after the call (line 55). The multiply instruction

```
mul     BL
```

multiplies the contents of the BL and AL registers and places the 16-bit result in the AX register. We discuss the multiplication instruction in Chapter 12.

Program 11.5 Recursive computation of factorial(N)

```
 1:  TITLE   Factorial - Recursive version            FACT.ASM
 2:  COMMENT |
 3:          Objective: To demonstrate principles of recursion.
 4:              Input: Requests an integer N from the user.
 5:             Output: Outputs N!
 6:  |
 7:
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  prompt_msg  DB  'Please enter a positive integer: ',0
12:  output_msg  DB  'The factorial is: ',0
13:  error_msg   DB  'Sorry! Not a positive number. Try again.',0
14:
15:  .CODE
16:  INCLUDE io.mac
17:
18:  main  PROC
19:        .STARTUP
20:        PutStr  prompt_msg     ; request the number
21:
22:  try_again:
23:        GetInt  BX             ; read number into BX
24:        nwln
25:        cmp     BX,0           ; test for positive number
26:        jge     num_ok
27:        PutStr  error_msg
28:        nwln
29:        jmp     try_again
30:
31:  num_ok:
32:        call    fact
33:
34:        PutStr  output_msg     ; output result
35:        PutInt  AX
36:        nwln
37:
38:  done:
39:        .EXIT
40:  main  ENDP
41:
42:  ;-----------------------------------------------------------
43:  ;Procedure fact receives a positive integer N in BX register.
44:  ;It returns N! in AX register.
```

```
45:  ;-----------------------------------------------------------
46:  fact  PROC
47:        cmp     BL,1            ; if N > 1, recurse
48:        jg      one_up
49:        mov     AX,1            ; return 1 for N < 2
50:        ret                     ; terminate recursion
51:
52:  one_up:
53:        dec     BL              ; recurse with (N-1)
54:        call    fact
55:        inc     BL
56:        mul     BL              ; AX = AL * BL
57:
58:        ret
59:  fact  ENDP
60:        END     main
```

Example 11.6 *Sorting an array of integers using the quicksort algorithm.*
Quicksort is one of the most popular sorting algorithms; it was proposed by C.A.R. Hoare in 1960. Once you understand the basic principle of the quicksort, you will see why recursion naturally expresses it.

At its heart, quicksort uses a divide-and-conquer strategy. The original sort problem is reduced to two smaller sort problems. This is done by selecting a partition element x and partitioning the array to be sorted into two subarrays: all elements less than x are placed in one subarray and all elements greater than x are in the other. Now, we have to sort these two subarrays, which are smaller than the original array. We apply the same procedure to sort these two subarrays. This is where the recursive nature of the algorithm shows up. The quicksort procedure to sort an N-element array is summarized below:

1. Select a partition element x.
2. Assume that we know where this element x should be in the final sorted array. Let it be at `array[i]`. We give details of this step shortly.
3. Move all the other elements that are less than x into positions `array[0]` $\cdots$ `array[i-1]`. Similarly, move those elements that are greater than x into positions `array[i+1]` $\cdots$ `array[N-1]`. Note that these two subarrays are not sorted.
4. Now apply the quicksort procedure recursively to sort these two subarrays until the array is sorted.

How do we know the final position of the partition element x without sorting the array? We don't have to sort the array; we just need to know the number of elements either before or after it. To clarify the working of the quicksort algorithm, let us look at an example. In this example,

and in our quicksort implementation, we pick the last element as the partition value. Obviously, the selection of the partition element influences performance of the quicksort. There are several better ways of selecting the partition value; you can get these details in any textbook on sorting.

Initial state:	`2 9 8 1 3 4 7 6`←—	Partition element;
After 1st pass:	`2 1 3 4` **6** `7 9 8`	Partition element 6 is in its final place.

The second pass works on the following two subarrays:

1st subarray:	`2 1 3 4`;
2nd subarray:	`7 9 8`.

To move the partition element to its final place, we use two pointers i and j. Initially, i points to the first element, and j points to the second to last element. Note that we are using the last element as the partition element. The index i is advanced until it points to an element that is greater than or equal to x. Similarly, j is moved backward until it points to an element that is less than or equal to x. Then we exchange the two values at i and j. We continue this process until i is greater than or equal to j. The quicksort pseudocode is shown below:

```
quick_sort (array, lo, hi)
    if (hi > lo)
        x := array[hi]
        i := lo
        j := hi
        while (i < j)
            while (array[i] < x)
                i := i + 1
            end while
            while (array[j] > x)
                j := j − 1
            end while
            if (i < j)
                array[i] ⟺ array[j]          /* exchange values */
            end if
        end while
        array[i] ⟺ array[hi]            /* exchange values */
        quick_sort (array, lo, i−1)
        quick_sort (array, i+1, hi)
    end if
end quick_sort
```

The quicksort program is shown in Program 11.6. The input values are read by the read loop (lines 28 to 35). This loop terminates if the input is zero. As written, this program can cause problems if the user enters more than 200 integers. You can easily remedy this problem

by initializing the CX with 200 and using the `loop` instruction on line 35. The three arguments are placed in the BX (array pointer), ESI (lo) and EDI (hi) registers (lines 39 to 41). After the quicksort call on line 42, the program outputs the sorted array (lines 45 to 54).

The quicksort procedure follows the pseudocode. Since we are not returning any values, we use `pusha` to preserve all registers (line 66). The two inner **while** loops are implemented by the LO and HI WHILE loops. The exchange of elements is done by using three `xchg` instructions (lines 93 to 95 and 99 to 101). The rest of the program follows the pseudocode in a straightforward manner.

Program 11.6 Sorting integers using the recursive quicksort algorithm

```
 1:  TITLE   Sorting integers using quicksort        QSORT.ASM
 2:  COMMENT |
 3:          Objective: Sorts an array of integers using
 4:                     quick sort. Uses recursion.
 5:              Input: Requests integers from the user.
 6:                     Terminated by entering zero.
 7:  |           Output: Outputs the sorted arrray.
 8:
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
12:  prompt_msg  DB  'Please enter integers. ',0DH,0AH
13:              DB  'Entering zero terminates the input.',0
14:  output_msg  DB  'The sorted array is: ',0
15:
16:  array1      DW  200 DUP (?)
17:
18:  .CODE
19:  .486
20:  INCLUDE io.mac
21:
22:  main  PROC
23:        .STARTUP
24:        PutStr  prompt_msg     ; request the number
25:        nwln
26:        mov     EBX, OFFSET array1
27:        xor     EDI,EDI        ; EDI keeps a count of input numbers
28:  read_more:
29:        GetInt  AX             ; read a number
30:        nwln
31:        mov     [EBX+EDI*2],AX ; store it in array
32:        cmp     AX,0           ; test if it is zero
33:        je      exit_read
34:        inc     EDI
```

```
35:       jmp     read_more
36:
37:  exit_read:
38:       ; prepare arguments for procedure call
39:       mov     EBX,OFFSET array1
40:       xor     ESI,ESI         ; ESI = lo index
41:       dec     EDI             ; EDI = hi index
42:       call    qsort
43:
44:       PutStr  output_msg      ; output sorted array
45:  write_more:
46:       ; since qsort preserves all registers, we will
47:       ; have valid EBX and ESI values.
48:       mov     AX,[EBX+ESI*2]
49:       cmp     AX,0
50:       je      done
51:       PutInt  AX
52:       nwln
53:       inc     ESI
54:       jmp     write_more
55:
56:  done:
57:       .EXIT
58:  main  ENDP
59:
60:  ;-------------------------------------------------------------
61:  ;Procedure qsort receives a pointer to the array in BX.
62:  ;LO and HI are received in ESI and EDI, respectively.
63:  ;It preserves all the registers.
64:  ;-------------------------------------------------------------
65:  qsort PROC
66:       pusha
67:       cmp     EDI,ESI
68:       jle     qsort_done        ; end recursion if hi <= lo
69:
70:       ; save hi and lo for later use
71:       mov     ECX,ESI
72:       mov     EDX,EDI
73:
74:       mov     AX,[EBX+EDI*2] ; AX = xsep
75:
76:  lo_loop:                       ;
77:       cmp     [EBX+ESI*2],AX    ;
78:       jge     lo_loop_done      ; LO while loop
79:       inc     ESI               ;
```

```
 80:       jmp     lo_loop           ;
 81:  lo_loop_done:
 82:
 83:       dec     EDI            ; hi = hi-1
 84:  hi_loop:
 85:       cmp     EDI,ESI           ;
 86:       jle     sep_done          ;
 87:       cmp     [EBX+EDI*2],AX    ; HI while loop
 88:       jle     hi_loop_done      ;
 89:       dec     EDI               ;
 90:       jmp     hi_loop           ;
 91:  hi_loop_done:
 92:
 93:       xchg    AX,[EBX+ESI*2]    ;
 94:       xchg    AX,[EBX+EDI*2]    ; x[i] <=> x[j]
 95:       xchg    AX,[EBX+ESI*2]    ;
 96:       jmp     lo_loop
 97:
 98:  sep_done:
 99:       xchg    AX,[EBX+ESI*2]    ;
100:       xchg    AX,[EBX+EDX*2]    ; x[i] <=> x[hi]
101:       xchg    AX,[EBX+ESI*2]    ;
102:
103:       dec     ESI
104:       mov     EDI,ESI           ; hi = i-1
105:       ; We will modify the ESI value in the next statement.
106:       ; Since the original ESI value is in EDI, we will use
107:       ; DSI value to get i+1 value for the second qsort call.
108:       mov     ESI,ECX
109:       call    qsort
110:
111:       ; EDI has the i value
112:       inc     EDI
113:       inc     EDI
114:       mov     ESI,EDI           ; lo = i+1
115:       mov     EDI,EDX
116:       call    qsort
117:
118:  qsort_done:
119:       popa
120:       ret
121:  qsort ENDP
122:       END     main
```

Recursion Versus Iteration

In theory, every recursive function has an iterative counterpart. To see this, let us write the iterative version to compute the factorial function.

```
int fact_iterative(int n)
{
    int    i, result;

    if (n == 0)
        return (1);

    result = 1;
    for(i = 1; i <= n; i++)
        result = result * i;
    return(result);
}
```

From this example, it is obvious that the recursive version is concise and reflects the mathematical definition of the factorial function. Once you get through the initial learning problems with recursion, recursive code is easier to understand for those functions that are defined recursively. Some such examples we have seen are the factorial function, Fibonacci number computation, binary search, and quicksort.

This leads us to the question of when to use recursion. To answer this question, we need to look at the potential problems recursion can cause. There are two main problems with recursion:

- *Inefficiency:* In most cases, recursive versions tend to be inefficient. You can see this point by comparing the recursive and iterative versions of the factorial function. The recursive version induces more overheads to invoke and return from procedure calls. To compute $N!$, we need to call the factorial function about N times. In the iterative version, the loop iterates about N times.
  Recursion could also introduce duplicate computation. For example, to compute the Fibonacci number

  ```
  fib(5) = fib(4) + fig(3)
  ```

  a recursive procedure computes `fib(3)` two times, `fib(2)` two times, and so on.
- *Demands More Memory:* Recursion tends to require more memory. This can be seen from the simple factorial example. For large N, the demand for stack memory can be excessive. In some cases, the limit on the available memory may make the recursive version unusable.

On the positive side, however, note that recursion leads to better understanding of the code for those naturally recursive problems. In this case, recursion should be used as it aids in program maintenance.

11.6 Summary

The addressing mode refers to the specification of operands required by an assembly language instruction. We discussed several memory addressing modes supported by the Pentium. We showed by means of examples how various 16- and 32-bit addressing modes are useful in supporting features of high-level languages.

Arrays are useful for representing a collection of related data. In high-level languages, programmers do not have to worry about the underlying storage representation used to store arrays in memory. However, when manipulating arrays in assembly language, we need to know this information. This is so because accessing individual elements of an array involves computing the corresponding displacement value. Although there are two common ways of storing a multidimensional array—row-major or column-major order—most high-level languages, including C, use the row-major order. We presented examples to illustrate how one- and two-dimensional arrays are manipulated in assembly language.

We have introduced recursion by means of factorial and quicksort examples. We redo these two examples in the MIPS assembly language in Chapter 15.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Activation record
- Address size override prefix
- Based addressing mode
- Based-indexed addressing mode
- Binary search
- Column-major order
- Fibonacci numbers
- Indexed addressing mode
- Insertion sort
- Location counter
- Multidimensional arrays
- One-dimensional arrays
- Operand size override prefix
- Quicksort
- Recursion
- Row-major order

11.7 Exercises

11–1 Explain why the register addressing mode is the most efficient of all the addressing modes supported by the Pentium.

11–2 Discuss the restrictions imposed by the immediate addressing mode.

11–3 Where (i.e., in which segment) are the data, specified by the immediate addressing mode, stored?

11–4 Describe all the 16-bit addressing modes that you can use to specify an operand that is located in memory.

11–5 Describe all the 32-bit addressing modes that you can use to specify an operand that is located in memory.

11–6 When is it necessary to use the segment override prefix?

11–7 When is it necessary to use the operand size override prefix?

11–8 When is it necessary to use the address size override prefix?

11–9 Is there a fundamental difference between the based and indexed addressing modes?

11–10 What additional flexibility does the based-indexed addressing mode have over based or indexed addressing modes?

11–11 Given the following declaration of `table1`

```
table1    DW    10 DUP (0)
```

fill in the blanks in the following code:

```
mov    SI, _______ ; SI = displacement of 5th element
                   ; (i.e., table1[4] in C)
mov    AX,table1[SI]
cmp    AX, _______ ; compare 5th and 4th elements
```

11–12 What is the difference between row-major and column-major orders for storing multidimensional arrays in memory?

11–13 In manipulating multidimensional arrays in assembly language, why is it necessary to know their underlying storage representation?

11–14 How is the `class_marks` array in Program 11.4 stored in memory: row-major or column-major order?

11–15 How would you change the `class_marks` declaration in order to store it in column-major order?

11–16 Assuming that subscripts begin with 0, derive a formula for the displacement (in bytes) of the element in row i and column j in a two-dimensional array that is stored in column-major order.

11–17 Suppose that the array **A** is a two-dimensional array stored in row-major order. Assume that a low value can be specified for each subscript. Derive a formula to express the displacement (in bytes) of **A**[i,j].

11.8 Programming Exercises

11–P1 What modifications would you make to the insertion sort procedure discussed in Section 11.3 to sort the array in descending order? Make the necessary modifications to the program and test it for correctness.

11–P2 Modify Program 11.3 to read array input data from the user. Your program should be able to accept up to 25 nonzero numbers from the user. A zero terminates the input. Report error if more than 25 numbers are given.

11–P3 Modify Program 11.4 to read marks from the user. The first number of the input indicates the number of students in class (i.e., number of rows), and the next number represents the number of tests given to the class (i.e., number of columns). Your program should be able to handle up to 20 students and five tests. Report error when exceeding these limits.

11–P4 Write a complete assembly language program to read two matrices **A** and **B** and display the result matrix **C**, which is the sum of **A** and **B**. Note that the elements of **C** can be obtained as

$$\mathbf{C}[i, j] = \mathbf{A}[i, j] + \mathbf{B}[i, j] .$$

Your program should consist of a main procedure that calls the `read_matrix` procedure twice to read data for **A** and **B**. It should then call the `matrix_add` procedure, which receives pointers to **A**, **B**, **C**, and the size of the matrices. Note that both **A** and **B** should have the same size. The `main` procedure calls another procedure to display **C**.

11–P5 Write a procedure to perform multiplication of matrices **A** and **B**. The procedure should receive pointers to the two input matrices (**A** of size $l \times m$, **B** of size $m \times n$), the product matrix **C**, and values l, m, and n. Also, the data for the two matrices should be obtained from the user. Devise a suitable user interface to input these numbers.

11–P6 Modify the program of the last exercise to work on matrices stored in the column-major order.

11–P7 Write a program to read a matrix (maximum size 10×10) from the user, and display the transpose of the matrix. To obtain the transpose of matrix **A**, write rows of **A** as columns. Here is an example:

If the input matrix is

$$\begin{bmatrix} 12 & 34 & 56 & 78 \\ 23 & 45 & 67 & 89 \\ 34 & 56 & 78 & 90 \\ 45 & 67 & 89 & 10 \end{bmatrix},$$

the transpose of the matrix is

$$\begin{bmatrix} 12 & 23 & 34 & 45 \\ 34 & 45 & 56 & 67 \\ 56 & 67 & 78 & 89 \\ 78 & 89 & 90 & 10 \end{bmatrix}.$$

11–P8 Write a program to read a matrix (maximum size 10×15) from the user, and display the subscripts of the maximum element in the matrix. Your program should consist of two procedures: `main` is responsible for reading the input matrix and for displaying the position of the maximum element. Another procedure `mat_max` is responsible for finding the position of the maximum element. Parameter passing should be done via the stack. For example, if the input matrix is

$$\begin{bmatrix} 12 & 34 & 56 & 78 \\ 23 & 45 & 67 & 89 \\ 34 & 56 & 78 & 90 \\ 45 & 67 & 89 & 10 \end{bmatrix}$$

the output of the program should be

The maximum element is at (2,3),

which points to the largest value 90.

11–P9 Write a program to read a matrix of integers, perform cyclic permutation of rows, and display the result matrix. Cyclic permutation of a sequence $a_0, a_1, a_2, \ldots, a_{n-1}$ is defined as $a_1, a_2, \ldots, a_{n-1}, a_0$. Apply this process for each row of the matrix. Your program should be able to handle up to 12×15 matrices. If the input matrix is

$$\begin{bmatrix} 12 & 34 & 56 & 78 \\ 23 & 45 & 67 & 89 \\ 34 & 56 & 78 & 90 \\ 45 & 67 & 89 & 10 \end{bmatrix},$$

the permuted matrix is

$$\begin{bmatrix} 34 & 56 & 78 & 12 \\ 45 & 67 & 89 & 23 \\ 56 & 78 & 90 & 34 \\ 67 & 89 & 10 & 45 \end{bmatrix}.$$

11–P10 Generalize the last exercise to cyclically permute by a user-specified number of elements.

11–P11 Write a complete assembly language program to do the following:

- Read the names of students in a class into a one-dimensional array.
- Read test scores of each student into a two-dimensional marks array.
- Output a letter grade for each student in the format:

```
student name      letter grade
```

You can use the following information in writing your program:

- Assume that the maximum class size is 20.
- Assume that the class is given four tests of equal weight (i.e., 25 points each).
- Test marks are rounded to the nearest integer so you can treat them as integers.
- Use the following table to convert percentage marks (i.e, sum of all four tests) to a letter grade:

Marks range	Grade
85–100	A
70–84	B
60–69	C
50–59	D
0–49	F

11–P12 Modify the program for the last exercise to also generate a class summary stating the number of students receiving each letter grade in the following format:

A = number of students receiving A,
B = number of students receiving B,
C = number of students receiving C,
D = number of students receiving D,
F = number of students receiving F.

11–P13 If we are given a square matrix (i.e., a matrix with the number of rows equal to the number of columns), we can classify it as the diagonal matrix if only its diagonal elements are nonzero; as an upper triangular matrix if all the elements below the diagonal are 0; and as a lower triangular matrix if all elements above the diagonal are 0. Some examples are:

Diagonal matrix:

$$\begin{bmatrix} 28 & 0 & 0 & 0 \\ 0 & 87 & 0 & 0 \\ 0 & 0 & 97 & 0 \\ 0 & 0 & 0 & 65 \end{bmatrix}.$$

Upper triangular matrix:

$$\begin{bmatrix} 19 & 26 & 35 & 98 \\ 0 & 78 & 43 & 65 \\ 0 & 0 & 38 & 29 \\ 0 & 0 & 0 & 82 \end{bmatrix}.$$

Lower triangular matrix:

$$\begin{bmatrix} 76 & 0 & 0 & 0 \\ 44 & 38 & 0 & 0 \\ 65 & 28 & 89 & 0 \\ 87 & 56 & 67 & 54 \end{bmatrix}.$$

Write an assembly language program to read a matrix and output the type of matrix.

11–P14 The Fibonacci function is defined as

$$\begin{aligned} \text{fib}(1) &= 1, \\ \text{fib}(2) &= 1, \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{for } n > 2. \end{aligned}$$

We have written iterative versions of this function in Chapter 10 (see page 422). Write a program to recursively compute fib(N). Your main program should request a positive integer N from the user and output fib(N). If the user enters a negative number, prompt her to try again.

11–P15 Ackermann's function $A(m, n)$ is defined for $m \geq 0$ and $n \geq 0$ as

$$\begin{aligned} A(0, n) &= N + 1 && \text{for } n \geq 0, \\ A(m, 0) &= A(m - 1, 1) && \text{for } m \geq 1, \\ A(m, n) &= A(m - 1, A(m, n - 1)) && \text{for } m \geq 1, n \geq 1. \end{aligned}$$

Write a recursive procedure to compute this function. Your main program should handle the user interface to request m and n and display the final result.

11–P16 Write a program to solve the Towers of Hanoi puzzle. The puzzle consists of three pegs and N disks. Disk 1 is smaller than disk 2, which is smaller than disk 3, and so on. Disk N is the largest. Initially, all N disks are on peg 1 such that the largest disk is at the bottom and the smallest at the top (i.e., in the order N, $N - 1$, . . ., 3, 2, 1 from bottom to top). The problem is to move these N disks from peg 1 to peg 2 under two constraints: You can move only one disk at a time and you must not place a larger disk on top of a smaller one. We can express a solution to this problem by using recursion. The function

```
move(N, 1, 2, 3)
```

moves N disks from peg 1 to peg 2 using peg 3 as the extra peg. There is a simple solution if you concentrate on moving the bottom disk on peg 1. The task `move(N, 1, 2, 3)` is equivalent to

```
move(N-1, 1, 3, 2)
```
move the remaining disk from peg 1 to 2
```
move(N-1, 3, 2, 1)
```

Even though the task appears to be complex, we write a very elegant and simple solution to solve this puzzle. Here is a version in C.

```
void move (int n, int x, int y, int z)
{
    if (n == 1)
        printf("Move the top disk from peg %d to %d\n", x, y};
    else
        move(n-1, x, z, y)
        printf("Move the top disk from peg %d to %d\n", x, y};
        move(n-1, z, y, x)
}

int main (void)
{
    int    disks;

    scanf("%d", &disks);
    move(disks, 1, 2, 3);
}
```

Test your program for a very small number of disks (say, less than 6). Even for 64 disks, it takes hundreds of thousands of years on whatever PC you have!

Chapter 12

Selected Pentium Instructions

Objectives

- To discuss how status flags are affected by arithmetic and logic instructions;
- To present the Pentium's multiplication and division instructions;
- To describe conditional execution instructions and how they are useful in implementing high-level language decision structures;
- To show how logical expressions are implemented in high-level languages;
- To give details about the Pentium's string processing instructions.

We discussed several Pentium instructions in the last three chapters. We now look at some of the remaining instructions. We start this chapter with a detailed discussion of the six status flags—zero, carry, overflow, sign, parity, and auxiliary flags. We have already used these flags informally. The discussion here will help us understand how some of the conditional jump instructions are executed by the Pentium. The next section deals with multiplication and division instructions. The Pentium instruction set includes multiplication and division instructions for both signed and unsigned integers.

We have already covered the basic jump instructions in Chapter 9. In Section 12.3, we look at indirect and conditional jump instructions. The following section discusses how selection and iterative constructs of high-level languages are implemented using assembly instructions. Section 12.5 gives details on implementing logical expressions in high-level languages. In addition to the logical instructions discussed in Chapter 9, the Pentium provides several bit test and scan instructions. These instructions are briefly reviewed in Section 12.6. The following section presents several examples to illustrate the use of the instructions discussed in this chapter.

The Pentium instruction set contains several string instructions. These are discussed in Section 12.8.2. In this section, we also look at repeat prefixes that facilitate block movement of data. Some string processing examples are given in Section 12.8.3. The chapter concludes with a summary.

12.1 Status Flags

Six flags in the flags register, described in Chapter 7, are used to monitor the outcome of the arithmetic, logical, and related operations. By now you are familiar with the purpose of some of these flags. The six flags are the zero flag (ZF), carry flag (CF), overflow flag (OF), sign flag (SF), auxiliary flag (AF), and parity flag (PF). For obvious reasons, these six flags are called the *status* flags.

When an arithmetic operation is performed, some of the flags are updated (set or cleared) to indicate certain properties of the result of that operation. For example, if the result of an arithmetic operation is zero, the zero flag is set (i.e., ZF = 1). Once a flag is set or cleared, it remains in that state until another instruction changes its value.

Note that not all assembly language instructions affect all the flags. Some instructions affect all six status flags, whereas other instructions affect none of the flags. And there are other instructions that affect only a subset of these flags. For example, the arithmetic instructions `add` and `sub` affect all six flags, but `inc` and `dec` instructions affect all but the carry flag. The `mov`, `push`, and `pop` instructions, on the other hand, do not affect any of the flags.

Here is an example illustrating how the zero flag changes with instruction execution:

```
;initially, assume that ZF is 0
mov    AL,55H  ; ZF is still 0
sub    AL,55H  ; result is zero
               ; Thus, ZF is set (ZF = 1)
push   BX      ; ZF remains 1
mov    BX,AX   ; ZF remains 1
pop    DX      ; ZF remains 1
mov    CX,0    ; ZF remains 1
inc    CX      ; result is 1
               ; Thus, ZF is cleared (ZF = 0)
```

As we show later, these flags can be tested either individually or in combination to affect the flow control of a program.

In understanding the workings of these status flags, you should know how signed and unsigned integers are represented. At this point, it is a good idea to review the material presented in Appendix A.

12.1.1 The Zero Flag

The purpose of the zero flag is to indicate whether the execution of the last instruction that affects the zero flag has produced a zero result. If the result was zero, ZF = 1; otherwise, ZF = 0. This is slightly confusing! You may want to take a moment to see through the confusion.

Although it is fairly intuitive to understand how the `sub` instruction affects the zero flag, it is not so obvious with other instructions. The following examples show some typical cases.

The code

```
mov     AL,0FH
add     AL,0F1H
```

sets the zero flag (i.e., ZF = 1). This is because, after executing the `add` instruction, the AL would contain zero (all eight bits zero). In a similar fashion, the code

```
mov     AX,0FFFFH
inc     AX
```

also sets the zero flag. The same is true for the following code:

```
mov     AX,1
dec     AX
```

Related Instructions

`jz` jump if zero (jump is taken if ZF = 1)
`jnz` jump if not zero (jump is taken if ZF = 0)

Usage

There are two main uses for the zero flag: testing for equality, and counting to a preset value.

Testing for Equality: The `cmp` instruction is often used to do this. Recall that `cmp` performs subtraction. The main difference between `cmp` and `sub` is that `cmp` does not store the result of the subtract operation. `cmp` performs the subtract operation only to set the status flags.

Here are some examples:

```
cmp     char,'$'        ; ZF = 1 if char is $
```

Similarly, two registers can be compared to see if they both have the same value.

```
cmp     AX,BX
```

Counting to a Preset Value: Another important use of the zero flag is shown below. Consider the following code:

```
sum := 0
for (i = 1 to M)
    for (j = 1 to N)
        sum := sum + 1
    end for
end for
```

The equivalent code in the assembly language is written as follows (assume that both M and N are ≥ 1):

```
      sub    AX,AX    ; AX = 0 (AX stores sum)
      mov    DX,M
outer_loop:
      mov    CX,N
inner_loop:
      inc    AX
      loop   inner_loop
      dec    DX
      jnz    outer_loop
exit_loops:
      mov    sum,AX
```

In the above example, the inner loop count is placed in the CX register so that we can use the `loop` instruction to iterate. Incidentally, the `loop` instruction does not affect any of the flags.

Since we have two nested loops to handle, we are forced to use another register to keep the outer loop count. We use the `dec` instruction and the zero flag to see if the outer loop has executed M times. This code is more efficient than initializing the DX register to one and using the code

```
inc    DX
cmp    DX,M
jle    outer_loop
```

in place of the `dec`/`jnz` instruction combination.

12.1.2 The Carry Flag

The carry flag records the fact that the result of an arithmetic operation on unsigned numbers is out of range (too big or too small) to fit the destination register or memory location. Consider the example

```
mov    AL,0FH
add    AL,0F1H
```

The addition of 0FH and F1H would produce a result of 100H that requires 9 bits to store, as shown below:

```
  00001111B   (0FH = 15D)
  11110001B   (F1H = 241D)
1 00000000B   (100H = 256D)
```

Since the destination register AL is only 8 bits long, the carry flag would be set to indicate that the result is too big to be held in AL.

To understand when the carry flag would be set, it is helpful to remember the range of unsigned numbers that can be represented. The range is given below for easy reference:

Size (bits)	Range
8	0 to 255
16	0 to 65,535
32	0 to 4,294,967,295

Any operation that produces a result that is outside this range sets the carry flag to indicate an underflow or overflow condition. It is obvious that any negative result is out of range, as illustrated by the following example:

```
mov     AX,12AEH    ;AX = 4782D
sub     AX,12AFH    ;AX = 4782D - 4783D
```

Executing the above code will set the carry flag because 12AFH − 12AFH produces a negative result (i.e., the subtract operation generates a borrow), which is too small to be represented using unsigned numbers. Thus, the carry flag is set to indicate this underflow condition.

Executing the code

```
mov     AL,0FFH
inc     AL
```

or the code

```
mov     AX,0
dec     AX
```

does not set the carry flag as we might expect because `inc` and `dec` instructions do not affect the carry flag.

Related Instructions

Conditional jumps are as follows:

`jc` jump if carry (jump is taken if CF = 1)
`jnc` jump if not carry (jump is taken if CF = 0)

Usage

The carry flag is useful in several situations:

- To propagate carry or borrow in multiword addition or subtraction operations.
- To detect overflow/underflow conditions.
- To test a bit using the shift/rotate family of instructions.

To Propagate Carry/Borrow: The assembly language arithmetic instructions can operate on 8-, 16-, or 32-bit data. If two operands, each more than 32 bits, are to be added, the addition has to proceed in steps by adding two 32-bit numbers at a time. The following example illustrates how we can add two 64-bit unsigned numbers. For convenience, we use the hex representation.

```
                    1  ← carry from lower 32 bits
x  =  3710 26A8 1257 9AE7H
y  =  489B A321 FE60 4213H
      7FAB C9CA 10B7 DCFAH
```

To accomplish this, we need two addition operations. The first operation adds the least significant (lower half) 32 bits of the two operands. This produces the lower half of the result. This addition operation could produce a carry that should be added to the upper 32 bits of the input. The other add operation performs the addition of the most significant (upper half) 32 bits and any carry generated by the first addition. This operation produces the upper half of the 64-bit result. An example to add two 64-bit numbers is given on page 343.

Similarly, adding two 128-bit numbers involves a four-step process, where each step adds two 32-bit words. The `sub` and other operations also require multiple steps when the data size is more than 32 bits.

To Detect Overflow/Underflow Conditions: In the previous example of $x + y$, if the second addition produces a carry, the result is too big to be held by 64 bits. In this case, the carry flag would be set to indicate the overflow condition. It is up to the programmer to handle such error conditions.

Testing a Bit: When using shift and rotate instructions (introduced in Chapter 9), the bit that has been shifted or rotated out is captured in the carry flag. This bit can be either the most significant bit (in the case of a left-shift or rotate), or the least significant bit (in the case of a right-shift or rotate). Once the bit is in the carry flag, conditional execution of the code is possible using conditional jump instructions that test the carry flag: `jc` (jump on carry) and `jnc` (jump if no carry).

Why `inc` and `dec` Do Not Affect the Carry Flag

We have stated that the `inc` and `dec` instructions do not affect the carry flag. The rationale for this is twofold:

1. The instructions `inc` and `dec` are typically used to maintain iteration or loop count. Using 32 bits, the number of iterations can be as high as 4,294,967,295. This number is sufficiently large for most applications. What if we need a count that is greater than this? Do we have to use `add` instead of `inc`? This leads to the second, and the main, reason.
2. The condition detected by the carry flag can also be detected by the zero flag. Why? Because `inc` and `dec` change the number only by 1. For example, suppose that the ECX register has reached its maximum value 4,294,967,295 (FFFFFFFFH). If we then execute

```
inc     ECX
```

we would normally expect the carry flag to be set to 1. However, we can detect this condition by noting that ECX = 0, which sets the zero flag. Thus, setting the carry flag is really redundant for these instructions.

12.1.3 The Overflow Flag

The overflow flag, in some respects, is the carry flag counterpart for the signed number arithmetic. The main purpose of the overflow flag is to indicate whether an operation on signed numbers has produced a result that is out of range. It is helpful to recall the range of numbers that can be represented using 8, 16, and 32 bits. For your convenience, the range of the numbers is given below:

Size (bits)	Range
8	−128 to +127
16	−32,768 to +32,767
32	−2,147,483,648 to +2,147,483,647

Executing the code

```
mov     AL,72H  ; 72H = 114D
add     AL,0EH  ; 0EH = 14D
```

will set the overflow flag to indicate that the result 80H (128D) is too big to be represented as an 8-bit signed number. The AL register will contain 80H, the correct result if the two 8-bit operands are treated as unsigned numbers. But AL contains an incorrect answer for 8-bit signed numbers (80H represents −128 in signed representation, not +128 as required).

Here is another example that uses the `sub` instruction. The AX register is initialized to −5, which is FFFBH in 2's complement representation using 16 bits.

```
mov     AX,0FFFBH    ; AX = -5
sub     AX,7FFDH     ; subtract 32,765 from AX
```

Execution of the above code will set the overflow flag as the result

$$(-5)-(32{,}765) = -32{,}770,$$

which is too small to be represented as a 16-bit signed number.

Note that the result will not be out of range (and hence the overflow flag will not be set) when we are adding two signed numbers of opposite sign or subtracting two numbers of the same sign.

Signed or Unsigned: How Does the System Know?

The values of the carry and overflow flags depend on whether the operands are unsigned or signed numbers. Given that a bit pattern can be treated both as representing a signed and an

unsigned number, a question that naturally arises is: How does the system know how your program is interpreting a given bit pattern? The answer is that the processor does not have a clue. It is up to our program logic to interpret a given bit pattern correctly. The processor, however, assumes both interpretations and sets the carry and overflow flags. For example, when executing

```
mov     AL,72H
add     AL,0EH
```

the processor treats 72H and 0EH as unsigned numbers. And since the result 80H (128) is within the range of 8-bit unsigned numbers (0 to 255), the carry flag is cleared (i.e., CF = 0). At the same time, 72H and 0EH are also treated as representing signed numbers. Since the result 80H (128) is outside the range of 8-bit signed numbers (−128 to +127), the overflow flag is set.

Thus, after executing the above two lines of code, CF = 0 and OF = 1. It is up to our program logic to take whichever flag is appropriate. If you are indeed representing unsigned numbers, disregard the overflow flag. Since the carry flag indicates a valid result, no exception handling is needed.

```
        mov     AL,72H
        add     AL,0EH
        jc      overflow
no_overflow:
        (no overflow code here)
              . . .
overflow:
        (overflow code here)
              . . .
```

If, on the other hand, 72H and 0EH are representing 8-bit signed numbers, we can disregard the carry flag value. Since the overflow flag is 1, our program will have to handle the overflow condition.

```
        mov     AL,72H
        add     AL,0EH
        jo      overflow
no_overflow:
        (no overflow code here)
              . . .
overflow:
        (overflow code here)
              . . .
```

Related Instructions

Conditional jumps are as follows:

`jo` jump on overflow (jump is taken if OF = 1)
`jno` jump on no overflow (jump is taken if OF = 0)

In addition, a special software interrupt instruction

`into` interrupt on overflow

is provided to test the overflow flag. Interrupts are discussed in Chapter 20.

Usage

The main purpose of the overflow flag is to indicate whether an arithmetic operation on signed numbers has produced an out-of-range result. The overflow flag is also affected by shift, multiply, and divide operations. More details on some of these instructions can be found in later sections of this chapter.

12.1.4 The Sign Flag

As the name implies, the sign flag indicates the sign of the result of an operation. Therefore, it is useful only when dealing with signed numbers. Recall that the most significant bit is used to represent the sign of a number: 0 for positive numbers and 1 for negative numbers. The sign flag gets a copy of the sign bit of the result produced by arithmetic and related operations. The following sequence of instructions

```
mov     AL,15
add     AL,97
```

will clear the sign flag (i.e., SF = 0) because the result produced by the `add` instruction is a positive number: 112D (which in binary is 01110000, where the leftmost bit representing the sign is zero).

The result produced by

```
mov     AL,15
sub     AL,97
```

is a negative number and sets the sign flag to indicate this fact. Remember that negative numbers are represented in 2s complement notation (see Appendix A). As discussed in Appendix A, the subtract operation can be treated as the addition of the corresponding negative number. Thus, $15 - 97$ is treated as $15 + (-97)$, where, as usual, -97 is expressed in 2s complement form. Therefore, after executing the above two instructions, the AL register contains AEH, as shown below:

```
  00001111B     (8-bit signed form of 15)
+ 10011111B     (8-bit signed number for −97)
  ----------
  10101110B
```

Since the sign bit of the result is 1, the result is negative and is in 2s complement form. You can easily verify that AEH is the 8-bit signed form of -82, which is the correct answer.

Related Instructions

Conditional jumps are as follows:

`js` jump on sign (jump is taken if SF = 1)
`jns` jump on no sign (jump is taken if SF = 0)

The `js` instruction causes the jump if the last instruction that updated the sign flag produced a negative result. The `jns` instruction causes the jump if the result was nonnegative.

Usage

The main use of the sign flag is to test the sign of the result produced by arithmetic and related instructions. Another use for the sign flag is in implementing counting loops that should iterate until (and including) the control variable is zero. For example, consider the following code:

for (i = M downto 0)
    <loop body>
end for

This can be implemented without using a `cmp` instruction as follows:

```
        mov     CX,M
for_loop:
           . . .
        <loop body>
           . . .
        dec     CX
        jns     for_loop
```

If we do not use the `jns` instruction, we have to use

```
        cmp     CX,0
        jl      for_loop
```

in its place.

From the user point of view, the sign bit of a number can be easily tested by using a logical or shift instruction. Compared to the other three flags we have discussed so far, the sign flag is used relatively infrequently in user programs. However, the processor uses the sign flag when executing conditional jump instructions on signed numbers (details are in Section 12.3.2 on page 500).

12.1.5 The Auxiliary Flag

The auxiliary flag indicates whether an operation has produced a result that has generated a carry out of or a borrow into the low-order four bits of 8-, 16-, or 32-bit operands. In computer jargon, four bits are referred to as a nibble. The auxiliary flag is set if there is such a carry or borrow; otherwise it is cleared.

In the example

```
mov     AL,43
add     AL,94
```

the auxiliary flag is set because there is a carry out of bit 3, as shown below:

```
              1  ← carry generated from lower to upper nibble
 43D  = 00101011B
 94D  = 01011110B
-----   --------
137D  = 10001001B
```

You can verify that executing the following code will clear the auxiliary flag:

```
mov     AL,43
add     AL,84
```

Since the following instruction sequence

```
mov     AL,43
sub     AL,92
```

generates a borrow into the low-order 4 bits, the auxiliary flag is set. On the other hand, the instruction sequence

```
mov     AL,43
sub     AL,87
```

clears the auxiliary flag.

Related Instructions and Usage

There are no conditional jump instructions that test the auxiliary flag. However, arithmetic operations on numbers expressed in decimal form or binary coded decimal (BCD) form use the auxiliary flag. Some related instructions are as follows:

`aaa`	ASCII adjust for addition
`aas`	ASCII adjust for subtraction
`aam`	ASCII adjust for multiplication
`aad`	ASCII adjust for division
`daa`	Decimal adjust for addition
`das`	Decimal adjust for subtraction

For details on these instructions and BCD numbers, see Appendices A and I.

12.1.6 The Parity Flag

This flag indicates the parity of the 8-bit result produced by an operation; if this result is 16 or 32 bits long, only the lower-order 8 bits are considered to set or clear the parity flag. The parity flag is set if the byte contains an even number of 1 bits; if there are an odd number of 1 bits, it is cleared. In other words, the parity flag indicates an even parity condition of the byte.

Thus, executing the code

```
mov     AL,53
add     AL,89
```

will set the parity flag because the result contains an even number of 1s (four 1 bits), as shown below:

```
 53D = 00110101B
 89D = 01011001B
142D = 10001110B
```

The instruction sequence

```
mov     AX,23994
sub     AX,9182
```

on the other hand, clears the parity flag, as the low-order 8 bits contain an odd number of 1s (five 1 bits), as shown below:

```
   23994D = 01011101 10111010B
+  -9182D = 11011100 00100010B
   14813D = 00111001 11011100B
```

Related Instructions

Conditional jumps are as follows:

`jp`	jump on parity (jump is taken if PF = 1)
`jnp`	jump on no parity (jump is taken if PF = 0)

The `jp` instruction causes the jump if the last instruction that updated the parity flag produced an even parity byte; the `jnp` instruction causes the jump for an odd parity byte.

Usage

This flag is useful for writing data encoding programs. As a simple example, consider transmission of data via modems using the 7-bit ASCII code. To detect simple errors during data transmission, a single parity bit is added to the 7-bit data. Assume that we are using even parity encoding. That is, every 8-bit character code transmitted will contain an even number of 1 bits. Then, the receiver can count the number of 1s in each received byte and flag transmission error if the byte contains an odd number of 1 bits. Such a simple encoding scheme can detect single bit errors (in fact, it can detect an odd number of single bit errors).

To encode, the parity bit is set or cleared depending on whether the remaining 7 bits contain an odd or even number of 1s, respectively. For example, if we are transmitting character A, whose 7-bit ASCII representation is 41H, we set the parity bit to 0 so that there is an even number of 1s. In the following examples, the parity bit is the leftmost bit:

```
A = 01000001
```

Table 12.1 Examples illustrating the effect on flags

	Code		AL	CF	ZF	SF	OF	PF
Example 1	`mov`	`AL,-5`						
	`sub`	`AL,123`	80H	0	0	1	0	0
Example 2	`mov`	`AL,-5`						
	`sub`	`AL,124`	7FH	0	0	0	1	0
Example 3	`mov`	`AL,-5`						
	`add`	`AL,132`	7FH	1	0	0	1	0
	`add`	`AL,1`	80H	0	0	1	1	0
Example 4	`sub`	`AL,AL`	00H	0	1	0	0	1
Example 5	`mov`	`AL,127`						
	`add`	`AL,129`	00H	1	1	0	0	1

For character C, the parity bit is set because its 7-bit ASCII code is 43H.

```
C = 11000011
```

Here is a procedure that encodes the 7-bit ASCII character code present in the AL register. The most significant bit (i.e., leftmost bit) is assumed to be zero.

```
parity_encode PROC
      shl    AL
      jp     parity_zero
      stc           ; CF = 1
      jmp    move_parity_bit
parity_zero:
      clc           ; CF = 0
move_parity_bit:
      rcr    AL
parity_encode ENDP
```

12.1.7 Flag Examples

Here we present two examples to illustrate how the status flags are affected by the arithmetic instructions. You can verify the answers by using a debugger (see Appendix D for information on debuggers).

Example 12.1 *Add/subtract example.*
Table 12.1 gives some examples of `add` and `sub` instructions and how they affect the flags. Updating of ZF, SF, and PF is easy to understand. The ZF is set whenever the result is zero; SF

is simply a copy of the most significant bit of the result; and PF is set whenever there are an even number of 1s in the result. In the rest of this section, we focus on the carry and overflow flags.

Example 1 performs −5 − 123. Note that −5 is represented internally as FBH, which is treated as 251 in unsigned representation. Subtracting 123 (=7BH) leaves 80H (=128) in AL. Since the result is within the range of unsigned 8-bit numbers, CF is cleared. For the overflow flag, the operands are interpreted as signed numbers. Since the result is −128, OF is also cleared.

Example 2 subtracts 124 from −5. For reasons discussed in the previous example, the CF is cleared. The OF, however, is set because the result is −129, which is outside the range of signed 8-bit numbers.

In Example 3, the first `add` statement adds 132 to −5. However, when treating them as unsigned numbers, 132 is actually added to 251, which results in a number that is greater than 255D. Therefore, CF is set. When treating them as signed numbers, 132 is internally represented as 84H (=−124). Therefore, the result −129 is smaller than −128. Therefore, the OF is also set. After executing the first `add` instruction, AL will have 7FH. The second `add` instruction increments 7FH. This sets the OF, but not CF.

Example 4 causes the result to be zero irrespective of the contents of the AL register. This sets the zero flag. Also, since the number of 1s is even, PF is also set in this example.

The last example adds 127D to 129D. Treating them as unsigned numbers, the result 256D is just outside the range, and sets CF. However, if we treat them as representing signed numbers, 129D is stored internally as 81H (=−127). The result, therefore, is zero and the OF is cleared.

Example 12.2 *A compare example.*

This example shows how the status flags are affected by the compare instruction discussed in Chapter 9 on page 344. Table 12.2 gives some examples of executing the

```
cmp     AL,DL
```

instruction. We leave it as an exercise to verify (without using a debugger) the flag values.

12.2 Arithmetic Instructions

For the sake of completeness, we list the arithmetic instructions supported by the Pentium:

Addition: `add, adc, inc`
Subtraction: `sub, sbb, dec, neg, cmp`
Multiplication: `mul, imul`
Division: `div, idiv`
Related instructions: `cbw, cwd, cdq, cwde, movsx, movzx`

We have already looked at the addition and subtraction instructions in Chapter 9. Here we discuss the remaining instructions. There are a few other arithmetic instructions that operate on decimal and BCD numbers. Details of these instructions can be found in Appendix I.

Table 12.2 Some examples of `cmp AL,DL`

AL	DL	CF	ZF	SF	OF	PF	AF
56	57	1	0	1	0	1	1
200	101	0	0	0	1	1	0
101	200	1	0	1	1	0	1
200	200	0	1	0	0	1	0
−105	−105	0	1	0	0	1	0
−125	−124	1	0	1	0	1	1
−124	−125	0	0	0	0	0	0

12.2.1 Multiplication Instructions

Multiplication is more complicated than the addition and subtraction operations for two reasons:

1. First, multiplication produces double-length results. That is, multiplying two n-bit values produces a $2n$-bit result. To see that this is indeed the case, consider multiplying two 8-bit numbers. Assuming unsigned representation, `FFH` (255D) is the maximum number that the source operands can take. Thus, the multiplication produces the maximum result, as shown below:

$$\underset{(255D)}{11111111} \times \underset{(255D)}{11111111} = \underset{(65025D)}{11111110\ 11111111}.$$

   Similarly, multiplication of two 16-bit numbers requires 32 bits to store the result, and two 32-bit numbers require 64 bits for the result.
2. Second, unlike the addition and subtraction operations, multiplication of signed numbers should be treated differently from that of unsigned numbers. This is because the resulting bit pattern depends on the type of input, as illustrated by the following example:

   We have just seen that treating `FFH` as the unsigned number results in multiplying 255D × 255D.

$$11111111 \times 11111111 = 11111110\ 11111111.$$

   Now, what if `FFH` is representing a signed number? In this case, `FFH` is representing −1D and the result should be 1, as shown below:

$$11111111 \times 11111111 = 00000000\ 00000001.$$

   As you can see, the resulting bit patterns are different for the two cases.

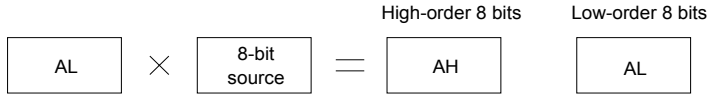
Thus, the Pentium provides two multiplication instructions: for unsigned numbers and for signed numbers. We first discuss the unsigned multiplication instruction, which has the format

```
mul     source
```

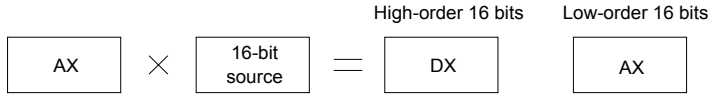
The `source` operand can be in a general-purpose register or in memory. Immediate operand specification is not allowed. Thus,

```
mul     10        ; invalid
```

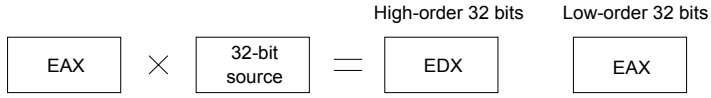
is an invalid instruction. The `mul` instruction works on 8-, 16-, and 32-bit unsigned numbers. But where is the second operand? The Pentium assumes that it is in the accumulator. If the source operand is a byte, it is multiplied by the contents of the AL register. The 16-bit result is placed in the AX register, as shown below:



If the source operand is a word, it is multiplied by the contents of the AX register and the doubleword result is placed in DX:AX, with the AX register holding the lower-order 16 bits, as shown below:



If the source operand is a doubleword, it is multiplied by the contents of the EAX register and the 64-bit result is placed in EDX:EAX, with the EAX register holding the lower-order 32 bits, as shown below:



The `mul` instruction affects all six status flags. However, it updates only the carry and overflow flags. The remaining four flags are undefined. The carry and overflow flags are set if the upper half of the result is nonzero; otherwise, they are both cleared.

Setting of the carry and overflow flags does not indicate an error condition. Instead, this condition implies that AH, DX, or EDX contains significant digits of the result.

For example, the code

```
mov     AL,10
mov     DL,25
mul     DL
```

will clear both the carry and overflow flags, as the result of the `mul` instruction is 250, which can be stored in the AL register (and the AH register contains `00000000`). On the other hand, executing

```
mov     AL,10
mov     DL,26
mul     DL
```

will set the carry and overflow flags indicating that the result is more than 255.

For signed numbers, we have to use the `imul` (integer multiplication) instruction, which has the same format[1] as the `mul` instruction

```
imul    source
```

The behavior of the `imul` instruction is similar to that of the `mul` instruction. The only difference to note is that the carry and overflow flags are set if the upper half of the result is not the sign extension of the lower half. To understand sign extension in signed numbers, consider the following example. We know that -66 is represented using 8 bits as

10111110.

Now, suppose that we can use 16 bits to represent the same number. Using 16 bits, -66 is represented as

1111111110111110.

The upper 8 bits are simply sign-extended (i.e., the sign bit is copied into these bits), and doing so does not change the magnitude.

Following the same logic, the positive number 66, represented using 8 bits as

01000010

can be sign-extended to 16 bits by adding eight leading zeros as shown below:

0000000001000010.

As with the `mul` instruction, setting of the carry and overflow flags does not indicate an error condition; it simply indicates that the result requires double length.

Here are some examples of the `imul` instruction. Execution of

```
mov     DL,0FFH     ; DL = -1
mov     AL,42H      ; AL = 66
imul    DL
```

causes the result

[1] The `imul` instruction supports several other formats, including specification of an immediate value. We do not discuss these details; see Intel's *Pentium Developer's Manual*.

1111111110111110

to be placed in the AX register. The carry and overflow flags are cleared, as AH contains the sign extension of the AL value. This is also the case for the following code:

```
mov     DL,0FFH    ; DL = -1
mov     AL,0BEH    ; AL = -66
imul    DL
```

which produces the result

0000000001000010 (+66)

in the AX register. Again, both the carry and overflow flags are cleared.

In contrast, both flags are set for the following code:

```
mov     DL,25    ; DL = 25
mov     AL,0F6H ; AL = -10
imul    DL
```

which produces the result

1111111100000110 (−250).

12.2.2 Division Instructions

The division operation is even more complicated than multiplication for two reasons:

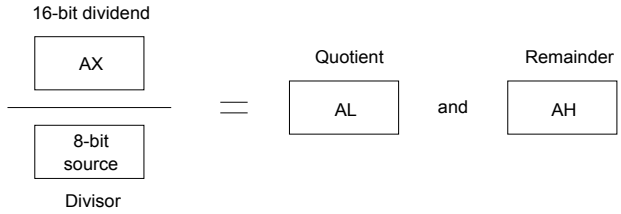
1. Division generates two result components: a quotient and a remainder.
2. In multiplication, by using double-length registers, overflow never occurs. In division, divide overflow is a real possibility. The Pentium generates a special software interrupt when a divide overflow occurs.

As with multiplication, two versions of the divide instruction are provided to work on unsigned and signed numbers.

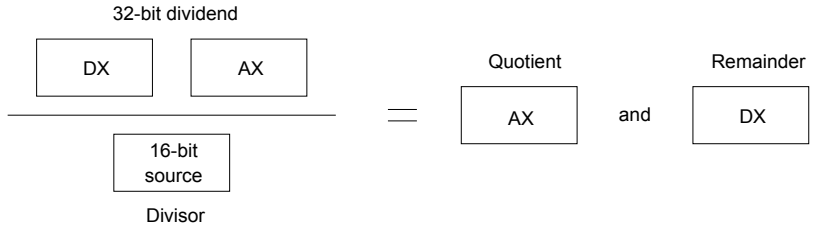
```
div     source  (unsigned)
idiv    source  (signed)
```

The source operand specified in the instruction is used as the divisor. As with the multiplication instruction, both division instructions can work on 8-, 16-, or 32-bit numbers. All six status flags are affected and are *undefined*. None of the flags are updated. We first consider the unsigned version.

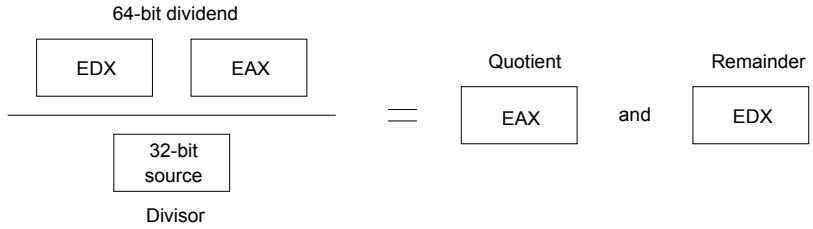
If the source operand is a byte, the dividend is assumed to be in the AX register and 16 bits long. After division, the quotient is returned in the AL register and the remainder in the AH register, as shown below:



For word operands, the dividend is assumed to be 32 bits long and in DX:AX (upper 16 bits in DX). After the division, the 16-bit quotient will be in the AX and the 16-bit remainder in the DX, as shown below:



For 32-bit operands, the dividend is assumed to be 64 bits long and in EDX:EAX. After the division, the 32-bit quotient will be in the EAX and the 32-bit remainder in the EDX, as shown below:



Example 12.3 *8-bit division.*
Consider dividing 251 by 12 (i.e., 251/12), which produces 20 as the quotient and 11 as the remainder. The code

```
mov    AX,251
mov    CL,12
div    CL
```

leaves 20 (14H) in the AL register and 11 (0BH) in the AH register. □

Example 12.4 *16-bit division.*
Consider the 16-bit division: 5147/300. Executing the code

```
xor     DX,DX       ; clear DX
mov     AX,141BH    ; AX = 5147D
mov     CX,012CH    ; CX = 300D
div     CX
```

leaves 17 (12H) in the AX and 47 (2FH) in the DX. □

Now let us turn our attention to the signed division operation. The `idiv` instruction has the same format and behavior as the unsigned `div` instruction including the registers used for the dividend, quotient, and remainder.

The `idiv` instruction introduces a slight complication when the dividend is a negative number. For example, assume that we want to perform the 16-bit division: −251/12. Since −251 = FF14H, the AX register is set to FF14H. However, the DX register has to be initialized to FFFFH by sign-extending the AX register. If the DX is set to 0000H as we did in the unsigned `div` operation, the dividend 0000FF14H is treated as a positive number 65300D. The 32-bit equivalent of −251 is FFFFFF14H. If the dividend is positive, DX should have 0000H.

To aid sign extension in instructions such as `idiv`, the Pentium provides several instructions:

`cbw` (convert byte to word)
`cwd` (convert word to doubleword)
`cdq` (convert doubleword to quadword)

These instructions take no operands. The first instruction can be used to sign-extend the AL register into the AH register and is useful with the 8-bit `idiv` instruction. The `cwd` instruction sign extends the AX into the DX register and is useful with the 16-bit `idiv` instruction. The `cdq` instruction sign extends the EAX into the EDX. In fact, both `cwd` and `cdq` use the same opcode 99H, and the operand size determines whether to sign-extend the AX or EAX register.

For completeness, we mention three other related instructions. The `cwde` instruction sign extends the AX into the EAX much as the `cbw` instruction. Just like the `cwd` and `cdq`, the same opcode 98H is used for both `cbw` and `cwde` instructions. The operand size determines which one should be applied. Note that `cwde` is different from `cwd` in that the `cwd` instruction uses the DX:AX register pair, whereas `cwde` uses the EAX register as the destination.

The Pentium also provides the following two move instructions:

`movsx dest,src` (move sign-extended `src` to `dest`)
`movzx dest,src` (move zero-extended `src` to `dest`)

In both these instructions, `dest` has to be a register, whereas the `src` operand can be in a register or memory. If the source is an 8-bit operand, the destination has to be either a 16- or 32-bit register. If the source is a 16-bit operand, the destination must be a 32-bit register.

Here are some examples of the `idiv` instruction:

Example 12.5 *Signed 8-bit division.*
The following sequence of instructions will perform the signed 8-bit division −95/12:

```
mov     AL,-95
cbw                 ; AH = FFH
mov     CL,12
idiv    CL
```

The `idiv` instruction will leave −7 (F9H) in the AL and −11 (F5H) in the AH. □

Example 12.6 *Signed 16-bit division.*
Suppose that we want to divide −5147 by 300. The sequence

```
mov     AX,-5147
cwd                   ; DX = FFFFH
mov     CX,300
idiv    CX
```

will perform this division and leave −17 (FFEFH) in AX and −47 (FFD1H) in DX as the remainder. □

Use of Shifts for Multiplication and Division

Shifts are more efficient to execute than the corresponding multiplication or division instructions. As an example, consider multiplying a signed 16-bit number in the AX register by 32. Using the `mul` instruction, we can write

```
; multiplicand is assumed to be in AX
mov     CX,32    ; multiplier in CX
mul     CX
```

This two-instruction sequence takes 12 clock cycles. Of this, `mul` takes about 11 clock cycles.

Let us look at how we can perform this multiplication with the `sal` instruction.

```
; multiplicand is assumed to be in AX
sal     AX,5     ; shift left by 5 bit positions
```

This code executes in just one clock cycle. This code also requires fewer bytes to encode. Whenever possible, use the shift instructions to perform multiplication and division by a power of two.

12.2.3 Application Examples

To demonstrate the application of the arithmetic instructions and flags, we write two procedures to input and output signed 8-bit integers in the range of −128 to +127. These procedures are as follows:

`GetInt8`	Reads a signed 8-bit integer from the keyboard into AL register;
`PutInt8`	Displays a signed 8-bit integer that is in AL register.

The following two subsections describe these procedures in detail.

PutInt8 Procedure

Our objective here is to write a procedure that displays the signed 8-bit integer that is in the AL register. In order to do this, we have to separate individual digits of the number to be displayed and convert them to their ASCII representation. The steps involved are illustrated by the following example, which assumes that AL has 108.

separate 1 → convert to ASCII → 31H → display
separate 0 → convert to ASCII → 30H → display
separate 8 → convert to ASCII → 38H → display

Separating individual digits is the heart of the procedure. This step is surprisingly simple! All we have to do is repeatedly divide the number by 10, as shown below (for a related discussion, see Appendix A):

		Quotient	Remainder
108/10	=	10	8
10/10	=	1	0
1/10	=	0	1

The only problem with this step is that the digits come out in the reverse order. Therefore, we need to buffer them before displaying. The pseudocode for the `PutInt8` procedure is shown below:

```
PutInt8 (number)
    if (number is negative)
    then
        display '–' sign
        number := –number {reverse sign}
    end if
    index := 0
    repeat
        quotient := number/10 {integer division}
        remainder := number % 10 {% is modulo operator}
        buffer[index] := remainder + 30H
        {save the ASCII character equivalent of remainder}
        index := index + 1
        number := quotient
    until (number = 0)
    repeat
        index := index – 1
        display digit at buffer[index]
    until (index = 0)
end PutInt8
```

Program 12.1 The `PutInt8` procedure to display an 8-bit signed number (in `getput.asm` file)

```
 1:  ;-------------------------------------------------------------
 2:  ;PutInt8 procedure displays a signed 8-bit integer that is
 3:  ;in AL register. All registers are preserved.
 4:  ;-------------------------------------------------------------
 5:  PutInt8 PROC
 6:          enter   3,0          ; reserves 3 bytes of buffer space
 7:          push    AX
 8:          push    BX
 9:          push    SI
10:          test    AL,80H       ; negative number?
11:          jz      positive
12:  negative:
13:          PutCh   '-'          ; sign for negative numbers
14:          neg     AL           ; convert to magnitude
15:  positive:
16:          mov     BL,10        ; divisor  = 10
17:          sub     SI,SI        ; SI = 0 (SI points to buffer)
18:  repeat1:
19:          sub     AH,AH        ; AH = 0 (AX is the dividend)
20:          div     BL
21:          ; AX/BL leaves AL = quotient & AH = remainder
22:          add     AH,'0'       ; convert remainder to ASCII
23:          mov     [BP+SI-3],AH ; copy into the buffer
24:          inc     SI
25:          cmp     AL,0         ; quotient = zero?
26:          jne     repeat1      ; if so, display the number
27:  display_digit:
28:          dec     SI
29:          mov     AL,[BP+SI-3] ; display digit pointed by SI
30:          PutCh   AL
31:          jnz     display_digit ; if SI<0, done displaying
32:  display_done:
33:          pop     SI           ; restore registers
34:          pop     BX
35:          pop     AX
36:          leave                ; clears local buffer space
37:          ret
38:  PutInt8 ENDP
```

The `PutInt8` procedure shown in Program 12.1 follows the logic of the pseudocode. Some points to note are the following:

- The buffer is considered as a local variable. Thus, we reserve three bytes on the stack using the `enter` instruction (see line 6).
- The code

```
test    AL,80H
jz      positive
```

  tests whether the number is negative or positive. Remember that the sign bit (the leftmost bit) is 1 for a negative number.
- Reversal of sign is done by the

```
neg     AL
```

  instruction on line 14.
- Note that we have to initialize AH with 0 (line 19), as the `div` instruction assumes a 16-bit dividend in the AX register when the divisor is an 8-bit number.
- Conversion to ASCII character representation is done on line 22 using

```
add     AH,'0'
```

- SI is used as the index into the buffer, which starts at [BP − 3]. Thus, [BP + SI − 3] points to the current byte in the buffer (line 29).
- The repeat while condition (index = 0) is tested by

```
jnz     display_digit
```

  on line 31.

GetInt8 Procedure

The `GetInt8` procedure reads a signed integer and returns the number in the AL register. Since only 8 bits are used to represent the number, the range is limited to −128 to +127 (both inclusive). The key part of the procedure converts a sequence of input digits received in character form to its binary equivalent. The conversion process, which involves repeated multiplication by 10, is illustrated for 158:

Input digit	Numeric value	Number = number ∗ 10 + numeric value
Initial value	—	0
'1' (31H)	1	0 ∗ 10 + 1 = 1
'5' (35H)	5	1 ∗ 10 + 5 = 15
'8' (38H)	8	15 ∗ 10 + 8 = 158

The pseudocode of the `GetInt8` procedure is as follows:

```
    GetInt8()
        read input character into char
        if ((char = '–') OR (char = '+'))
        then
            sign := char
            read the next character into char
        end if
        number := char – '0' {convert to numeric value}
        count := 2 {number of remaining digits to read}
    repeat
        read the next character into char
        if (char ≠ carriage return)
        then
            number := number * 10 + (char – '0')
        else
            goto convert_done
        end if
        count := count – 1
    until (count = 0)
convert_done:
    {check for out-of-range error}
    if ((number > 128) OR ((number = 128) AND (sign ≠ '–')))
    then
        out of range error
        set carry flag
    else {number is OK}
        clear carry flag
    end if
    if (sign = '–')
    then
        number = –number {reverse sign}
    end if
end GetInt8
```

Program 12.2 The `GetInt8` procedure to read a signed 8-bit integer (in `getput.asm` file)

```
1:  ;-------------------------------------------------------------
2:  ;GetInt8 procedure reads an integer from the keyboard and
3:  ;stores its equivalent binary in AL register. If the number
4:  ;is within -128 and +127 (both inclusive), CF is cleared;
5:  ;otherwise, CF is set to indicate out-of-range error.
6:  ;No error check is done to see if the input consists of
7:  ;digits only. All registers are preserved except for AX.
```

```
 8:  ;-------------------------------------------------------------
 9:  CR    EQU   0DH
10:
11:  GetInt8 PROC
12:          push    BX           ; save registers
13:          push    CX
14:          push    DX
15:          sub     DX,DX        ; DX = 0
16:          sub     BX,BX        ; BX = 0
17:  get_next_char:
18:          GetCh   DL           ; read input from keyboard
19:          cmp     DL,'-'       ; is it negative sign?
20:          je      sign         ; if so, save the sign
21:          cmp     DL,'+'       ; is it positive sign?
22:          jne     digit        ; if not, process the digit
23:  sign:
24:          mov     BH,DL        ; BH keeps sign of input number
25:          jmp     get_next_char
26:  digit:
27:          sub     AX,AX        ; AX = 0
28:          mov     BL,10        ; BL holds the multiplier
29:          sub     DL,'0'       ; convert ASCII to numeric
30:          mov     AL,DL
31:          mov     CX,2         ; maximum two more digits to read
32:  convert_loop:
33:          GetCh   DL
34:          cmp     DL,CR        ; carriage return?
35:          je      convert_done ; if so, done reading the number
36:          sub     DL,'0'       ; else, convert ASCII to numeric
37:          mul     BL           ; multiply total (in AL) by 10
38:          add     AX,DX        ; and add the current digit
39:          loop    convert_loop
40:  convert_done:
41:          cmp     AX,128
42:          ja      out_of_range ; if AX > 128, number out of range
43:          jb      number_OK    ; if AX < 128, number is valid
44:          cmp     BH,'-'       ; if AX = 128, must be a negative;
45:          jne     out_of_range ; otherwise, an invalid number
46:  number_OK:
47:          cmp     BH,'-'       ; number negative?
48:          jne     number_done  ; if not, we are done
49:          neg     AL           ; else, convert to 2's complement
50:  number_done:
51:          clc                  ; CF = 0 (no error)
52:          jmp     done
```

```
53:  out_of_range:
54:          stc                    ; CF = 1 (range error)
55:  done:
56:          pop     DX             ; restore registers
57:          pop     CX
58:          pop     BX
59:          ret
60:  GetInt8 ENDP
```

The assembly language code for the `GetInt8` procedure is given in Program 12.2. The procedure uses `GetCh` to read input digits into the DL register.

- The character input digits are converted to their numeric equivalent by subtracting '0' on line 29.
- The multiplication is done on line 37, which produces a 16-bit result in AX. Note that the numeric value of the current digit (in DX) is added (line 38) to detect the overflow condition rather than the 8-bit value in DL.
- When the conversion is done, AX will have the absolute value of the input number. Lines 41 to 45 perform the out-of-range error check. To do this check, the following conditions are tested:

  $AX > 128 \Rightarrow$ out of range
  $AX = 128 \Rightarrow$ input must be a negative number to be a valid number; otherwise, out of range

  The `ja` (jump if above) and `jb` (jump if below) on lines 42 and 43 are conditional jumps for unsigned numbers. These two instructions are discussed in the next section.
- If the input is a negative number, the value in AL is converted to 2's complement representation by using the `neg` instruction (line 49).
- The `clc` (clear CF) and `stc` (set CF) instructions are used to indicate the error condition (lines 51 and 54).

12.3 Conditional Execution

In Chapter 9, we have presented some of the Pentium instructions that support conditional execution. In that chapter, we briefly discussed the direct unconditional jump and some conditional jump instructions. Here we look at the indirect jump and the remaining conditional jump instructions.

12.3.1 Indirect Jumps

So far, we have used only the direct jump instruction. In direct jump, the target address (i.e., its relative offset value) is encoded into the jump instruction itself (see Figure 9.1 on page 348). We limit our discussion to jumps within a segment.

In an indirect jump, the target address is specified indirectly either through memory or a general-purpose register. Thus, we can write

```
jmp     CX
```

if the CX register contains the offset of the target. In indirect jumps, the target offset is the absolute value (unlike in direct jumps, which use relative offset values). We give an example next.

Example 12.7 *An example with an indirect jump.*
The objective here is to show how we can use the indirect jump instruction. To this end, we show a simple program that reads a digit from the user and prints the corresponding choice. The listing is shown in Program 12.3. An input between 0 and 9 is valid. Any other input to the program may cause the system to hang up or crash. Inputs 0 through 2 display the selection class (see lines 23 to 25). Other inputs terminate the program.

In order to use the indirect jump, we have to build a jump table of pointers (see lines 11 to 20). The input digit is converted to act as an index into this table and is used by the indirect jump instruction (line 40). Since the range of the index value is not checked, an input like 'a' produces an index value that is outside the range of the jump table. This can lead to unexpected system behavior. In one of the exercises, you are asked to remedy this problem.

Program 12.3 An example demonstrating the use of the indirect jump

```
 1:  TITLE    Sample indirect jump example    IJUMP.ASM
 2:  COMMENT |
 3:          Objective: To demonstrate the use of indirect jump.
 4:              Input: Requests a digit character from the user.
 5:                     WARNING: Typing any other character may
 6:                              crash the system!
 7:  |           Output: Appropriate class selection message.
 8:  .MODEL SMALL
 9:  .STACK  100H
10:  .DATA
11:  jump_table  DW  code_for_0   ; indirect jump pointer table
12:              DW  code_for_1
13:              DW  code_for_2
14:              DW  default_code ; default code for digits 3-9
15:              DW  default_code
16:              DW  default_code
17:              DW  default_code
18:              DW  default_code
19:              DW  default_code
20:              DW  default_code
21:
22:  prompt_msg  DB  'Type a character (digits ONLY): ',0
```

```
msg_0       DB  'Economy class selected.',0
msg_1       DB  'Business class selected.',0
msg_2       DB  'First class selected.',0
msg_default DB  'Not a valid code!',0

.CODE
INCLUDE  io.mac
main PROC
        .STARTUP
read_again:
        PutStr prompt_msg    ; request a digit
        sub    AX,AX         ; AX = 0
        GetCh  AL            ; read input digit and
        nwln
        sub    AL,'0'        ; convert to numeric equivalent
        mov    SI,AX         ; SI is index into jump table
        add    SI,SI         ; SI = SI * 2
        jmp    jump_table[SI] ; indirect jump based on SI
test_termination:
        cmp    AL,2
        ja     done
        jmp    read_again
code_for_0:
        PutStr msg_0
        nwln
        jmp    test_termination
code_for_1:
        PutStr msg_1
        nwln
        jmp    test_termination
code_for_2:
        PutStr msg_2
        nwln
        jmp    test_termination
default_code:
        PutStr msg_default
        nwln
        jmp    test_termination
done:
        .EXIT
main    ENDP
        END main
```

Multiway Conditional Statements

In high-level languages, a two- or three-way conditional execution can be easily expressed by `if` statements. For large multiway conditional execution, writing the code with nested `if` statements is tedious and errorprone. High-level languages such as C provide a special construct for multiway conditional execution. In this section we look at the C `switch` construct.

Example 12.8 *Multiway conditional execution in C.*
Consider the following code:

```
switch (ch)
{
    case '0':
             count[0]++; /* increment count[0] */
             break;
    case '1':
             count[1]++;
             break;
    case '2':
             count[2]++;
             break;
    case '3':
             count[3]++;
             break;
    default:
             count[4]++;
}
```

The semantics of the switch statement are as follows. If character `ch` is `0`, the `count[0]++` statement is executed. The `break` statement is necessary to escape from the `switch` statement. Similarly, if `ch` is `1`, `count[1]` is incremented, and so on. The `default` case statement is executed if `ch` is not one of the values specified in other case statements.

Turbo C produces the assembly language code shown in Figure 12.1. The jump table is constructed in the code segment (lines 31 to 34). As a result, the CS segment override prefix is used in the indirect jump statement on line 11. Register BX is used as an index into the jump table. Since each entry in the jump table is two bytes long, BX is multiplied by two using `shl` on line 10. □

12.3.2 Conditional Jumps

Conditional jump instructions can be divided into three groups:

1. Jumps based on the value of a single arithmetic flag,
2. Jumps based on unsigned comparisons, and
3. Jumps based on signed comparisons.

```
 1:  _main    PROC    NEAR
 2:               . . .
 3:               . . .
 4:           mov    AL,ch
 5:           cbw
 6:           sub    AX,48    ; 48 = ASCII for 0
 7:           mov    BX,AX
 8:           cmp    BX,3
 9:           ja     default
10:           shl    BX,1     ; BX = BX * 2
11:           jmp    WORD PTR CS:jump_table[BX]
12:  case_0:
13:           inc    WORD PTR [BP-10]
14:           jmp    SHORT end_switch
15:  case_1:
16:           inc    WORD PTR [BP-8]
17:           jmp    SHORT end_switch
18:  case_2:
19:           inc    WORD PTR [BP-6]
20:           jmp    SHORT end_switch
21:  case_3:
22:           inc    WORD PTR [BP-4]
23:           jmp    SHORT end_switch
24:  default:
25:           inc    WORD PTR [BP-2]
26:  end_switch:
27:             . . .
28:             . . .
29:  _main    ENDP
30:  jump_table  LABEL  WORD
31:              DW     case_0
32:              DW     case_1
33:              DW     case_2
34:              DW     case_3
35:                . . .
```

Figure 12.1 Assembly language code for the `switch` statement.

Jumps Based on Single Flags

The Pentium instruction set provides two conditional jump instructions—one for jumps if the flag tested is set, and the other for jumps when the tested flag is cleared—for each arithmetic flag except the auxiliary flag. Since we have discussed some of these instructions in Chapter 9, we just summarize them in Table 12.3.

Table 12.3 Jumps based on single flag value

Mnemonic		Meaning	Jumps if
Testing for zero:	jz	Jump if zero	ZF = 1
	je	Jump if equal	
	nz	Jump if not zero	ZF = 0
	jne	Jump if not equal	
	jcxz	Jump if CX = 0	CX = 0 (no flags tested)
Testing for carry:	jc	Jump if carry	CF = 1
	jnc	Jump if no carry	CF = 0
Testing for overflow:	jo	Jump if overflow	OF = 1
	jno	Jump if no overflow	OF = 0
Testing for sign:	js	Jump if (negative) sign	SF = 1
	jns	Jump if no (negative) sign	SF = 0
Testing for parity:	jp	Jump if parity	PF = 1
	jpe	Jump if parity is even	
	jnp	Jump if not parity	PF = 0
	jpo	Jump if parity is odd	

Jumps Based on Unsigned Comparisons

When comparing two numbers

```
cmp     num1,num2
```

it is necessary to know whether these numbers `num1` and `num2` are representing signed or unsigned numbers in order to establish a relationship between them. As an example, assume that AL = 10110111B and DL = 01101110B. Then the statement

```
cmp     AL,DL
```

should appropriately update flags to yield that AL > DL if we are treating their contents as unsigned numbers. This is because, in unsigned representation, AL = 183 and DL = 110. However, if the contents of AL and DL are treated as signed numbers, AL < DL as the AL register is storing a negative number (−73), and the DL register is storing a positive number (+110).

Note that when using a `cmp` statement such as

```
cmp     num1,num2
```

Table 12.4 Jumps based on unsigned comparison

Mnemonic	Meaning	Condition tested
je jz	Jump if equal Jump if zero	ZF = 1
jne jnz	Jump if not equal Jump if not zero	ZF = 0
ja jnbe	Jump if above Jump if not below or equal	CF = 0 and ZF = 0
jae jnb	Jump if above or equal Jump if not below	CF = 0
jb jnae	Jump if below Jump if not above or equal	CF = 1
jbe jna	Jump if below or equal Jump if not above	CF = 1 or ZF = 1

we are always comparing `num1` to `num2` (e.g., num1 $<$ num2, num1 $>$ num2, etc.). There are six possible relationships between two numbers:

$$\begin{aligned} \text{num1} &= \text{num2} \\ \text{num1} &\neq \text{num2} \\ \text{num1} &> \text{num2} \\ \text{num1} &\geq \text{num2} \\ \text{num1} &< \text{num2} \\ \text{num1} &\leq \text{num2}. \end{aligned}$$

For unsigned numbers, the carry and zero flags record the necessary information to establish one of the above six relationships.

The six conditional jump instructions (along with six aliases) and the flag conditions tested are shown in Table 12.4. Notice that "above" and "below" are used for $>$ and $<$ relationships for unsigned comparisons, reserving "greater" and "less" for signed comparisons, as we have seen in Chapter 9.

Jumps Based on Signed Comparisons

The $=$ and $\neq$ comparisons work with either signed or unsigned numbers, as we essentially compare the bit pattern for a match. For this reason, `je` and `jne` also appear in Table 12.6 for signed comparisons.

Table 12.5 Examples with Snum1 > Snum2

Snum1	Snum2	ZF	OF	SF
56	55	0	0	0
56	−55	0	0	0
−55	−56	0	0	0
55	−75	0	1	1

For signed comparisons, three flags record the necessary information: sign flag (SF), overflow flag (OF), and zero flag (ZF). Testing for $=$ and $\neq$ simply involves testing whether the ZF is set or cleared, respectively. With the signed numbers, establishing $<$ and $>$ relationships is somewhat tricky.

Let us assume that we are executing the following `cmp` instruction:

```
cmp     Snum1,Snum2
```

Conditions for Snum1 > Snum2

Table 12.5 shows some examples in which `Snum1` $>$ `Snum2` holds. It appears from these examples that `Snum1` $>$ `Snum2` holds if ZF = 0 and OF = SF. We cannot just use the OF = SF condition because, if two numbers are equal, ZF = 1 and OF = SF = 0. In fact, these conditions do imply the "greater than" relationship between `Snum1` and `Snum2`. As shown in Table 12.6, these conditions are tested for the `jg` conditional jump.

Conditions for Snum1 < Snum2

As in the previous case, we first develop our intuition by means of a few examples. Table 12.7 shows some examples in which the condition `Snum1` $<$ `Snum2` holds.

It appears from these examples that `Snum1` $<$ `Snum2` holds if ZF = 0 and OF $\neq$ SF. In this case, ZF = 0 is redundant, and the condition reduces to OF $\neq$ SF. As indicated in Table 12.6, this is the condition tested by the `jl` conditional jump instruction.

12.4 Implementing High-Level Language Decision Structures

In this section, we see how the jump instructions can be used to implement high-level language selection and iterative structures.

12.4.1 Selective Structures

Most high-level languages provide the `if-then-else` construct that allows selection from two alternative actions. The generic format of this type of construct is as follows:

Table 12.6 Jumps based on signed comparison

Mnemonic	Meaning	Condition tested
je jz	Jump if equal Jump if zero	ZF = 1
jne jnz	Jump if not equal Jump if not zero	ZF = 0
jg jnle	Jump if greater Jump if not less or equal	ZF = 0 and SF = OF
jge jnl	Jump if greater or equal Jump if not less	SF = OF
jl jnge	Jump if less Jump if not greater or equal	SF ≠ OF
jle jng	Jump if less or equal Jump if not greater	ZF = 1 or SF ≠ OF

Table 12.7 Examples with `Snum1` < `Snum2`

`Snum1`	`Snum2`	ZF	OF	SF
55	56	0	0	1
−55	56	0	0	1
−56	−55	0	0	1
−75	55	0	1	0

if (condition)
then
    true-alternative
else
    false-alternative
end if

The *true-alternative* is executed when the *condition* is true; otherwise, the *false-alternative* is executed. In C, the format is

```
if (condition)
  {
    statement-T1
    statement-T2
```

```
        ...
    statement-Tn
  }
else
  {
    statement-F1
    statement-F2
        ...
    statement-Fn
  };
```

We now consider some example C statements and the corresponding assembly language code generated by the Turbo C compiler.

Example 12.9 *An `if` example with a relational operator.*
Consider the following C code, which assigns the larger of `value1` and `value2` to `bigger`. All three variables are declared as integers (`int` data type):

```
if (value1 > value2)
    bigger = value1;
else
    bigger = value2;
```

The Turbo C compiler generates the following assembly language code (we have embellished the code a little to improve readability):

```
      mov    AX,value1
      cmp    AX,value2
      jle    else_part
then_part:
      mov    AX,value1   ; redundant
      mov    bigger,AX
      jmp    SHORT end_if
else_part:
      mov    AX,value2
      mov    bigger,AX
end_if:
            . . .
            . . .
```

In this example, the condition testing is done by the compare and conditional jump instructions. The label `then_part` is really not needed but included to improve readability of the code. The first statement in the `then_part` is redundant, but Turbo C generates it anyway. This is an example of the kind of inefficiencies introduced by compilers. □

Example 12.10 *An `if` example with a logical AND operator.*
The following code tests whether `ch` is a lowercase character. The condition in this example is a compound condition of two simple conditional statements connected by the logical `and` operator.

```
if ((ch >= 'a') && (ch <= 'z'))
    ch = ch - 32;
```

(*Note:* `&&` stands for the logical `and` operator in C.) The corresponding assembly language code generated by Turbo C is (the variable `ch` is mapped to the DL register):

```
      cmp    DL,'a'
      jb     not_lower_case
      cmp    DL,'z'
      ja     not_lower_case
lower_case:
      mov    AL,DL
      add    AL,224
      mov    DL,AL
not_lower_case:
            . . .
```

The compound condition is implemented by two compare and conditional jump instructions. Notice that `ch - 32` is implemented by

```
add     AL,224    ;AL = AL + (-32)
```

Since `jb` and `ja` are used, the characters are treated as unsigned numbers. Also, there is redundancy in the code generated by the compiler. An advantage of using the assembly language is that we can avoid such redundancies. □

Example 12.11 *An `if` example with a logical OR operator.*
Consider the following code with a compound condition using the logical `or` operator:

```
if ((index < 1) || (index > 100))
    index = 0;
```

(*Note:* `||` stands for the logical `or` operator in C.) The assembly language code generated is

```
      cmp    CX,1
      jl     zero_index
      cmp    CX,100
      jle    end_if
zero_index:
      xor    CX,CX      ; CX = 0
end_if:
            . . .
```

Turbo C maps the variable `index` to the CX register. Also, the code uses the exclusive-or (`xor`) logical operator to clear CX. □

12.4.2 Iterative Structures

High-level languages provide several looping constructs, including `while`, `repeat-until`, and `for` loops. Here we briefly look at how we can implement these iterative structures using the assembly language instructions.

While Loop

The `while` loop tests a condition before executing the loop body. For this reason, this loop is called the *pretest* or *entry-test* loop. The loop body is executed repeatedly as long as the condition is true.

Example 12.12 *A `while` loop example.*
Consider the following C code:

```
while(total < 700)
  {
    <loop body>
  }
```

Turbo C generates the following assembly language code:

```
      jmp    while_cond
while_body:
            . . .
      < instructions for
        while loop body >
            . . .
while_cond:
      cmp    BX,700
      jl     while_body
end_while:
            . . .
```

The variable `total` is mapped to the BX register. An initial unconditional jump transfers control to `while_cond` to test the loop condition. □

Repeat-Until Loop

This is a *post-test* or *exit-test* loop. This iterative construct tests the repeat condition after executing the loop body. Thus, the loop body is executed at least once.

Example 12.13 *A `repeat-until` example.*
Consider the following C code:

```
do
  {
    <loop body>
  }
while (number > 0);
```

The Turbo C compiler generates the following assembly language code:

```
loop_body:
            . . .
      < instructions for
        do-while loop body >
            . . .
cond_test:
      or     DI,DI
      jg     loop_body
end_do_while:
            . . .
```

The variable `number` is mapped to the DI register. To test the loop condition, it uses `or` rather than the `cmp` instruction. □

For Loop

The `for` loop is also called the *counting* loop because it iterates a fixed number of times. The `for` loop in C is much more flexible and powerful than the basic counting loop. Here we consider only the basic counting `for` loop.

Example 12.14 *Upward counting `for` loop.*

```
for (i = 0; i < SIZE; i++)    /* for (i = 0 to SIZE-1) */
  {
    <loop body>
  };
```

Turbo C generates the following assembly language code:

```
      xor    SI,SI
      jmp    SHORT for_cond
loop_body:
            . . .
      < instructions for
        the loop body >
            . . .
      inc    SI
for_cond:
      cmp    SI,SIZE
      jl     loop_body
            . . .
```

As with the `while` loop, an unconditional jump transfers control to `for_cond` to first test the iteration condition before executing the loop body. The counting variable `i` is mapped to the SI register. □

Example 12.15 *Downward counting `for` loop.*

```
for (i = SIZE-1; i >= 0; i--)      /* for (i = SIZE−1 downto 0) */
  {
    <loop body>
  };
```

Turbo C generates the following assembly language code:

```
        mov     SI,SIZE-1
        jmp     SHORT for_cond
loop_body:
              . . .
        < instructions for
          the loop body >
              . . .
        dec     SI
for_cond:
        or      SI,SI
        jge     loop_body
              . . .
```

The counting variable `i` is mapped to the SI register. Since our termination condition is `i = 0`, the `or` instruction is used to test this condition as in Example 12.13. □

12.5 Logical Expressions in High-Level Languages

Some high-level languages such as Pascal provide Boolean data types. Boolean variables can assume one of two values: `true` or `false`. Other languages such as C do not explicitly provide Boolean data types. This section discusses Boolean data representation and evaluation of compound logical expressions.

12.5.1 Representation of Boolean Data

In principle, only a single bit is needed to represent the Boolean data. However, such a representation, although compact, is not convenient, as testing a variable involves isolating the corresponding bit.

Most languages use a byte to represent the Boolean data. If the byte is zero, it represents `false`; otherwise, `true`. Note that any value other than 0 can represent `true`.

In C language, which does not provide an explicit Boolean data type, any data variable can be used in a logical expression to represent Boolean data. The same rules mentioned above apply: if the value is 0, it is treated as `false`, and any nonzero value is treated as `true`. Thus, for example, we can use integer variables as Boolean variables in logical expressions.

12.5.2 Logical Expressions

The logical instructions are useful in implementing logical expressions of high-level languages. For example, C provides the following four logical operators:

C operator	Meaning
`&&`	AND
`||`	OR
`^`	Exclusive-OR
`~`	NOT

To illustrate the use of logical instructions in implementing high-level language logical expressions, let us look at the following C example:

```
if (~(X && Y) ^ (Y || Z))
    X = Y + Z;
```

The corresponding assembly language code generated by the Turbo C compiler is shown in Figure 12.2.

The variable `X` is mapped to [BP−12], `Y` to CX, and `Z` to [BP−14]. The code on lines 1 to 8 implements partial evaluation of `(X && Y)`. That is, if `X` is false, it doesn't test the `Y` value. This is called partial evaluation, which is discussed on page 513. The result of the evaluation, 0 or 1, is stored in AX. The `not` instruction is used to implement the `~` operator (line 10), and the value of `~(X && Y)` is stored on the stack (line 11).

Similarly, lines 13 to 21 evaluate `(Y || Z)`, and the result is placed in AX. The value of `~(X && Y)` is recovered to DX (line 23), and the `xor` instruction is used to implement the `^` operator (line 24). If the result is zero (i.e., false), the body of the if statement is skipped (line 25).

12.5.3 Bit Manipulation

Some high-level languages provide bitwise logical operators. For example, C provides bitwise `and` (`&`), `or` (`|`), `xor` (`^`), and `not` (`~`) operators. These can be implemented by using the logical instructions provided in the assembly language.

The C language also provides shift operators: left shift (`<<`) and right shift (`>>`). These operators can be implemented with the assembly language shift instructions.

Table 12.8 shows how the logical and shift families of instructions are used to implement the bitwise logical and shift operators of the C language. The variable `mask` is assumed to be in the SI register.

12.5.4 Evaluation of Logical Expressions

Logical expressions can be evaluated in one of two ways: by full evaluation, or by partial evaluation. These methods are discussed next.

```
 1:        cmp    WORD PTR [BP-12],0  ; X = false?
 2:        je     false1              ; if so, (X && Y) = false
 3:        or     CX,CX               ; Y = false?
 4:        je     false1
 5:        mov    AX,1                ; (X && Y) = true
 6:        jmp    SHORT skip1
 7:  false1:
 8:        xor    AX,AX               ; (X && Y) = false
 9:  skip1:
10:        not    AX                  ; AX = ~(X && Y)
11:        push   AX                  ; save ~(X && Y)
12:        ; now evaluate the second term
13:        or     CX,CX               ; Y = true?
14:        jne    true2               ; if so, (Y || Z) = true
15:        cmp    WORD PTR [BP-14],0  ; Z = false?
16:        je     skip2
17:  true2:
18:        mov    AX,1                ; (X || Y) = true
19:        jmp    SHORT skip3
20:  skip2:
21:        xor    AX,AX               ; (X || Y) = false
22:  skip3:
23:        pop    DX                  ; DX = ~(X && Y)
24:        xor    DX,AX               ; ~(X && Y) ^ (Y || Z)
25:        je     end_if              ; if zero, whole exp. false
26:  if_body:
27:        mov    AX,CX               ; AX = Y
28:        add    AX,WORD PTR [BP-14] ; AX = Y + Z
29:        mov    WORD PTR [BP-12],AX ; X = Y + Z
30:  end_if:
31:              . . .
```

Figure 12.2 Assembly language code for the example logical expression.

Full Evaluation

In this method of evaluation, the entire logical expression is evaluated before assigning a value (true or false) to the expression. Full evaluation is used in Pascal.

For example, in full evaluation, the expression

if ((X ≥ 'a') AND (X ≤ 'z')) OR ((X ≥ 'A') AND (X ≤ 'Z'))

is evaluated by evaluating all four relational terms and then applying the logical operators. For example, the Turbo Pascal compiler generates the assembly language code shown in Figure 12.3 for this logical expression.

Table 12.8 Examples of bitwise operators

C statement	Assembly language code
`mask = mask>>2` (right-shift mask by two bit positions)	`shr SI,2`
`mask = mask<<4` (left-shift mask by four bit positions)	`shl SI,4`
`mask = ~mask` (complement mask)	`not SI`
`mask = mask & 85` (bitwise `and`)	`and SI,85`
`mask = mask | 85` (bitwise `or`)	`or SI,85`
`mask = mask ^ 85` (bitwise `xor`)	`xor SI,85`

Partial Evaluation

The final result of a logical expression can be obtained without evaluating the whole expression. The following rules help us in this:

1. In an expression of the form

   ```
   cond1 AND cond2
   ```

   the outcome is known to be false if one input is false. For example, if we follow the convention of evaluating logical expressions from left to right, as soon as we know that `cond1` is false, we can assign false to the entire logical expression. Only when `cond1` is true do we need to evaluate `cond2` to know the final value of the logical expression.
2. Similarly, in an expression of the form

   ```
   cond1 OR cond2
   ```

   the outcome is known if `cond1` is true. The evaluation can stop at that point. We need to evaluate `cond2` only if `cond1` is false.

This method of evaluation is used in C. The assembly language code for the previous logical expression, produced by the Turbo C compiler, is shown in Figure 12.4. The code does not use any logical instructions. Instead, the conditional jump instructions are used to implement the logical expression. Partial evaluation clearly results in efficient code.

```
 1:          cmp     ch,'Z'
 2:          mov     AL,0
 3:          ja      skip1
 4:          inc     AX
 5:   skip1:
 6:          mov     DL,AL
 7:          cmp     ch,'A'
 8:          mov     AL,0
 9:          jb      skip2
10:          inc     AX
11:   skip2:
12:          and     AL,DL
13:          mov     CL,AL
14:          cmp     ch,'z'
15:          mov     AL,0
16:          ja      skip3
17:          inc     AX
18:   skip3:
19:          mov     DL,AL
20:          cmp     ch,'a'
21:          mov     AL,0
22:          jb      skip4
23:          inc     AX
24:   skip4:
25:          and     AL,DL
26:          or      AL,CL
27:          or      AL,AL
28:          je      skip_if
29:          << if body here >>
30:   skip_if:
31:          << code following the if >>
```

Figure 12.3 Assembly language code for full evaluation.

Partial evaluation also has an important advantage beyond the obvious reduction in evaluation time. Suppose `X` and `Y` are inputs to the program. A statement such as

```
if ((X > 0) AND (Y/X > 100))
   . . .
```

can cause a divide-by-zero error if `X = 0` when full evaluation is used. However, with partial evaluation, when `X` is zero, `(X > 0)` is false, and the second term `(Y/X > 100)` is not evaluated at all. This is used frequently in C programs to test if a pointer is NULL before manipulating the data to which it points.

```
 1:          cmp     ch,'a'
 2:          jb      skip1
 3:          cmp     ch,'z'
 4:          jbe     skip2
 5:   skip1:
 6:          cmp     ch,'A'
 7:          jb      skip_if
 8:          cmp     ch,'Z'
 9:          ja      skip_if
10:   skip2:
11:          << if body here >>
12:   skip_if:
13:          << code following the if >>
```

Figure 12.4 Assembly language code for partial evaluation.

Of course, with full evaluation we can rewrite the last condition to avoid the divide-by-zero error as

```
if (X > 0)
    if (Y/X > 100)
          . . .
```

12.6 Bit Instructions

The Pentium provides bit test and modification instructions as well as bit scan instructions. This section briefly reviews these two instruction groups. An example that uses these instructions is given later (see Example 12.19).

12.6.1 Bit Test and Modify Instructions

There are four bit test instructions. Each instruction takes the position of the bit to be tested. The least significant bit is considered as bit position zero. A summary of the four instructions is given below:

Instruction	Effect on Selected Bit
`bt` (Bit Test)	No effect
`bts` (Bit Test and Set)	Selected bit ← 1
`btr` (Bit Test and Reset)	Selected bit ← 0
`btc` (Bit Test and Complement)	Selected bit ← NOT(Selected bit)

All four instructions copy the selected bit into the carry flag. The format of these instructions is the same. We use the `bt` instruction to illustrate the format,

```
bt    operand,bit_pos
```

where `operand` can be a word or doubleword located either in a register or memory. The `bit_pos` specifies the bit position to be tested. It can be specified as an immediate value or in a 16- or 32-bit register. Instructions in this group affect only the carry flag. The other five status flags are undefined following a bit test instruction.

12.6.2 Bit Scan Instructions

Bit scan instructions scan the operand for a 1 bit and return the bit position in a register. There are two instructions: one to scan forward and the other to scan backward. The format is

```
bsf    dest_reg,operand     ;bit scan forward
bsr    dest_reg,operand     ;bit scan reverse
```

where `operand` can be a word or doubleword located either in a register or memory. The `dest_reg` receives the bit position. It must be a 16- or 32-bit register. The zero flag is set if all bits of `operand` are 0; otherwise, the ZF is cleared and the `dest_reg` is loaded with the bit position of the first 1 bit while scanning forward (for `bsf`), or reverse (for `bsr`). These two instructions affect only the zero flag. The other five status flags are undefined following a bit scan instruction.

12.7 Illustrative Examples

In this section, we present four examples to show the use of the selection and iteration instructions discussed in this chapter. The first example uses linear search for locating a number in an unsorted array, and the second example sorts an array of integers using the selection sort algorithm. The last two examples show how multiplication can be done using shift and add instructions.

Example 12.16 *Linear search of an array of integers.*

In this example, the user is asked to input an array of nonnegative integers and a number to be searched. The program uses linear search to locate the number in the unsorted array.

The `main` procedure initializes the input array by reading a maximum of MAX_SIZE number of nonnegative integers into the array. The user, however, can terminate the input by entering a negative number. The `loop` instruction, with CX initialized to MAX_SIZE (line 29), is used to iterate a maximum of MAX_SIZE times. The other loop termination condition (i.e., a negative input) is tested on lines 33 and 34. The rest of the main program queries the user for a number and calls the linear search procedure to locate the number. This process is repeated as long as the user appropriately answers the query.

The linear search procedure receives a pointer to an array, its size, and the number to be searched via the stack. The search process starts at the first element of the array and proceeds

until either the element is located or the array is exhausted. We use the `loopne` to test these two conditions for termination of the search loop. CX is initialized (line 83) to the size of the array. In addition, a compare (line 88) tests if there is a match between the two numbers. If so, the zero flag is set and `loopne` terminates the search loop. If the number is found, the index of the number is computed (lines 92 and 93) and returned in AX.

Program 12.4 Linear search of an integer array

```
 1:  TITLE      Linear search of integer array      LIN_SRCH.ASM
 2:  COMMENT |
 3:          Objective: To implement linear search of an integer
 4:                     array; demonstrates the use of loopne.
 5:              Input: Requests numbers to fill array and a
 6:                     number to be searched for from user.
 7:             Output: Displays the position of the number in
 8:                     the array if found; otherwise, not found
 9:  |                  message.
10:  .MODEL SMALL
11:  .STACK 100H
12:  .DATA
13:  MAX_SIZE       EQU 100
14:  array          DW  MAX_SIZE DUP (?)
15:  input_prompt   DB  'Please enter input array: '
16:                 DB  '(negative number terminates input)',0
17:  query_number   DB  'Enter the number to be searched: ',0
18:  out_msg        DB  'The number is at position ',0
19:  not_found_msg  DB  'Number not in the array!',0
20:  query_msg      DB  'Do you want to quit (Y/N): ',0
21:
22:  .CODE
23:  .486
24:  INCLUDE io.mac
25:  main    PROC
26:          .STARTUP
27:          PutStr  input_prompt ; request input array
28:          mov     BX,OFFSET array
29:          mov     CX,MAX_SIZE
30:  array_loop:
31:          GetInt  AX           ; read an array number
32:          nwln
33:          cmp     AX,0         ; negative number?
34:          jl      exit_loop    ; if so, stop reading numbers
35:          mov     [BX],AX      ; otherwise, copy into array
36:          inc     BX           ; increment array address
37:          inc     BX
```

```
38:         loop    array_loop   ; iterates a maximum of MAX_SIZE
39:  exit_loop:
40:         mov     DX,BX        ; DX keeps the actual array size
41:         sub     DX,OFFSET array  ; DX = array size in bytes
42:         sar     DX,1         ; divide by 2 to get array size
43:  read_input:
44:         PutStr  query_number ; request number to be searched for
45:         GetInt  AX           ; read the number
46:         nwln
47:         push    AX           ; push number, size & array pointer
48:         push    DX
49:         push    OFFSET array
50:         call    linear_search
51:         ; linear_search returns in AX the position of the number
52:         ; in the array; if not found, it returns 0.
53:         cmp     AX,0         ; number found?
54:         je      not_found    ; if not, display number not found
55:         PutStr  out_msg      ; else, display number position
56:         PutInt  AX
57:         jmp     SHORT user_query
58:  not_found:
59:         PutStr  not_found_msg
60:  user_query:
61:         nwln
62:         PutStr  query_msg    ; query user whether to terminate
63:         GetCh   AL           ; read response
64:         nwln
65:         cmp     AL,'Y'       ; if response is not 'Y'
66:         jne     read_input   ; repeat the loop
67:  done:                       ; otherwise, terminate program
68:         .EXIT
69:  main    ENDP
70:
71:  ;-------------------------------------------------------------
72:  ; This procedure receives a pointer to an array of integers,
73:  ; the array size, and a number to be searched via the stack.
74:  ; If found, it returns in AX the position of the number in
75:  ; the array; otherwise, returns 0.
76:  ; All registers, except AX, are preserved.
77:  ;-------------------------------------------------------------
78:  linear_search  PROC
79:        enter   0,0
80:        push    BX           ; save registers
81:        push    CX
82:        mov     BX,[BP+4]    ; copy array pointer
```

```
 83:         mov     CX,[BP+6]      ; copy array size
 84:         mov     AX,[BP+8]      ; copy number to be searched
 85:         sub     BX,2           ; adjust index to enter loop
 86:  search_loop:
 87:         add     BX,2           ; update array index
 88:         cmp     AX,[BX]        ; compare the numbers
 89:         loopne  search_loop
 90:         mov     AX,0           ; set return value to zero
 91:         jne     number_not_found  ; modify it if number found
 92:         mov     AX,[BP+6]      ; copy array size
 93:         sub     AX,CX          ; compute array index of number
 94:  number_not_found:
 95:         pop     CX             ; restore registers
 96:         pop     BX
 97:         leave
 98:         ret     6
 99:  linear_search  ENDP
100:         END   main
```

Example 12.17 *Sorting of an array of integers using the selection sort algorithm.*
The main program is very similar to that in the last example, except for the portion that displays the sorted array. The sort procedure receives a pointer to the array to be sorted and its size via the stack. It uses the selection sort algorithm to sort the array in ascending order. The basic idea is as follows:

1. Search the array for the smallest element,
2. Move the smallest element to the first position by exchanging values of the first and smallest element positions,
3. Search the array for the smallest element from the second position of the array,
4. Move this element to position 2 by exchanging values as in Step 2,
5. Continue this process until the array is sorted.

The selection sort procedure implements the following algorithm:

```
selection_sort (array, size)
    for (position = 0 to size−2)
        min_value := array[position]
        min_position := position
        for (j = position+1 to size−1)
            if (array[j] < min_value)
            then
                min_value := array[j]
```

```
                    min_position := j
                end if
            end for
            if (position ≠ min_position)
            then
                array[min_position] := array[position]
                array[position] := min_value
            end if
        end for
    end selection_sort
```

The selection sort procedure shown in Program 12.5 implements this pseudocode with the following mapping of variables: `position` is maintained in SI and DI is used for the index variable `j`. `min_value` is maintained in DX and `min_position` in AX. The number of elements to be searched for finding the minimum value is kept in CX.

Program 12.5 Sorting of an array of integers using the selection sort algorithm

```
 1:  TITLE    Sorting an array by selection sort    SEL_SORT.ASM
 2:  COMMENT |
 3:          Objective: To sort an integer array using selection sort.
 4:              Input: Requests numbers to fill array.
 5:  |          Output: Displays sorted array.
 6:  .MODEL SMALL
 7:  .STACK 100H
 8:  .DATA
 9:  MAX_SIZE       EQU 100
10:  array          DW  MAX_SIZE DUP (?)
11:  input_prompt   DB  'Please enter input array: '
12:                 DB  '(negative number terminates input)',0
13:  out_msg        DB  'The sorted array is:',0
14:
15:  .CODE
16:  .486
17:  INCLUDE io.mac
18:  main    PROC
19:          .STARTUP
20:          PutStr  input_prompt ; request input array
21:          mov     BX,OFFSET array
22:          mov     CX,MAX_SIZE
23:  array_loop:
24:          GetInt  AX           ; read an array number
25:          nwln
26:          cmp     AX,0         ; negative number?
```

```
27:          jl      exit_loop    ; if so, stop reading numbers
28:          mov     [BX],AX      ; otherwise, copy into array
29:          add     BX,2         ; increment array address
30:          loop    array_loop   ; iterates a maximum of MAX_SIZE
31:  exit_loop:
32:          mov     DX,BX        ; DX keeps the actual array size
33:          sub     DX,OFFSET array  ; DX = array size in bytes
34:          sar     DX,1         ; divide by 2 to get array size
35:          push    DX           ; push array size & array pointer
36:          push    OFFSET array
37:          call    selection_sort
38:          PutStr  out_msg      ; display sorted array
39:          nwln
40:          mov     CX,DX
41:          mov     BX,OFFSET array
42:  display_loop:
43:          PutInt  [BX]
44:          nwln
45:          add     BX,2
46:          loop    display_loop
47:  done:
48:          .EXIT
49:  main    ENDP
50:
51:  ;---------------------------------------------------------
52:  ; This procedure receives a pointer to an array of integers
53:  ; and the array size via the stack. The array is sorted by
54:  ; using the selection sort. All registers are preserved.
55:  ;---------------------------------------------------------
56:  SORT_ARRAY  EQU  [BX]
57:  selection_sort PROC
58:         pusha                ; save registers
59:         mov     BP,SP
60:         mov     BX,[BP+18]   ; copy array pointer
61:         mov     CX,[BP+20]   ; copy array size
62:         sub     SI,SI        ; array left of SI is sorted
63:  sort_outer_loop:
64:         mov     DI,SI
65:         ; DX is used to maintain the minimum value and AX
66:         ; stores the pointer to the minimum value
67:         mov     DX,SORT_ARRAY[SI]  ; min. value is in DX
68:         mov     AX,SI        ; AX = pointer to min. value
69:         push    CX
70:         dec     CX           ; size of array left of SI
71:  sort_inner_loop:
```

```
72:         add     DI,2            ; move to next element
73:         cmp     DX,SORT_ARRAY[DI] ; less than min. value?
74:         jle     skip1           ; if not, no change to min. value
75:         mov     DX,SORT_ARRAY[DI] ; else, update min. value (DX)
76:         mov     AX,DI                ;       & its pointer (AX)
77:  skip1:
78:         loop    sort_inner_loop
79:         pop     CX
80:         cmp     AX,SI           ; AX = SI?
81:         je      skip2           ; if so, element at SI is its place
82:         mov     DI,AX           ; otherwise, exchange
83:         mov     AX,SORT_ARRAY[SI]  ; exchange min. value
84:         xchg    AX,SORT_ARRAY[DI]  ; & element at SI
85:         mov     SORT_ARRAY[SI],AX
86:  skip2:
87:         add     SI,2            ; move SI to next element
88:         dec     CX
89:         cmp     CX,1            ; if CX = 1, we are done
90:         jne     sort_outer_loop
91:         popa                    ; restore registers
92:         ret     4
93:  selection_sort ENDP
94:         END   main
```

Example 12.18 *Multiplication using only shifts and adds.*

The objective of this example is to show how multiplication can be done entirely by shift and add operations. We consider multiplication of two unsigned 8-bit numbers. In order to use the shift operation, we have to express the multiplier as a power of 2. For example, if the multiplier is 64, the result can be obtained by shifting the multiplicand left by six bit positions (because $2^6 = 64$).

What if the multiplier is not a power of 2? In this case, we have to express this number as a sum of powers of 2. For example, if the multiplier is 10, it can be expressed as $8 + 2$, where each term is a power of 2. Then the required multiplication can be done by two shifts and one addition.

The question now is: How do we express the multiplier in this form? If we look at the binary representation of the multiplicand (10D = 00001010B), there is a 1 in bit positions with weights 8 and 2. Thus, for each 1 bit in the multiplier, the multiplicand should be shifted left by a number of positions equal to the bit position number. In the above example, the multiplicand should be shifted left by 3 and 1 bit positions and then added. This procedure is formalized in the following algorithm:

```
mult8 (number1, number2)
    result := 0
    for (i = 7 downto 0)
        if (bit(number2, i) = 1)
            result := result + number1 * 2^i
        end if
    end for
end mult8
```

The function `bit` returns the ith bit of `number2`. The program listing is given in Program 12.6.

Program 12.6 Multiplication of two 8-bit numbers using only shifts and adds

```
 1:  TITLE   8-bit multiplication using shifts   SHL_MLT.ASM
 2:  COMMENT |
 3:          Objective: To multiply two 8-bit unsigned numbers
 4:                     using SHL rather than MUL instruction.
 5:              Input: Requests two unsigned numbers from user.
 6:  |           Output: Prints the multiplication result.
 7:  .MODEL SMALL
 8:  .STACK 100H
 9:  .DATA
10:  input_prompt   DB  'Please input two short numbers: ',0
11:  out_msg1       DB  'The multiplication result is: ',0
12:  query_msg      DB  'Do you want to quit (Y/N): ',0
13:
14:  .CODE
15:  INCLUDE io.mac
16:  main    PROC
17:          .STARTUP
18:  read_input:
19:          PutStr  input_prompt ; request two numbers
20:          GetInt  AX           ; read the first number
21:          nwln
22:          GetInt  BX           ; read the second number
23:          nwln
24:          call    mult8        ; mult8 uses SHL instruction
25:          PutStr  out_msg1
26:          PutInt  AX           ; mult8 leaves result in AX
27:          nwln
28:          PutStr  query_msg    ; query user whether to terminate
29:          GetCh   AL           ; read response
30:          nwln
31:          cmp     AL,'Y'       ; if response is not 'Y'
```

```
32:          jne     read_input    ; repeat the loop
33:  done:                         ; otherwise, terminate program
34:          .EXIT
35:  main    ENDP
36:
37:  ;-------------------------------------------------------------
38:  ; mult8 multiplies two 8-bit unsigned numbers passed on to
39:  ; it in registers AL and BL. The 16-bit result is returned
40:  ; in AX. This procedure uses only SHL instruction to do the
41:  ; multiplication. All registers, except AX, are preserved.
42:  ;-------------------------------------------------------------
43:  mult8  PROC
44:         push    CX            ; save registers
45:         push    DX
46:         push    SI
47:         xor     DX,DX         ; DX = 0 (keeps mult. result)
48:         mov     CX,7          ; CX = # of shifts required
49:         mov     SI,AX         ; save original number in SI
50:  repeat1:       ; multiply loop - iterates 7 times
51:         rol     BL,1          ; test bits of number2 from left
52:         jnc     skip1         ; if 0, do nothing
53:         mov     AX,SI         ; else, AX = number1*bit weight
54:         shl     AX,CL
55:         add     DX,AX         ; update running total in DX
56:  skip1:
57:         loop    repeat1
58:         rol     BL,1          ; test the rightmost bit of AL
59:         jnc     skip2         ; if 0, do nothing
60:         add     DX,SI         ; else, add number1
61:  skip2:
62:         mov     AX,DX         ; move final result into AX
63:         pop     SI            ; restore registers
64:         pop     DX
65:         pop     CX
66:         ret
67:  mult8  ENDP
68:         END   main
```

The main program requests two numbers from the user and calls the procedure `mult8` and displays the result. The main program then queries the user whether to quit and proceeds according to the response.

The `mult8` procedure multiplies two 8-bit unsigned numbers and returns the result in AX. It follows the algorithm discussed on page 523. The multiply loop (lines 50 to 57) tests the most significant seven bits of the multiplier. The least significant bit is tested on line 58. Notice

that the procedure uses `rol` rather than `shl` to test each bit (lines 51 and 58). The use of `rol` automatically restores the BL register after eight rotates. □

Example 12.19 *Multiplication using only shifts and adds—version 2.*
In this example, we rewrite the `mult8` procedure of the last example by using the bit test and scan instructions. In the previous version, we used a loop (see lines 50 to 57) to test each bit. Since we are interested only in 1 bits, we can use a bit scan instruction to do this job. The modified `mult8` procedure is shown below:

```
 1:  ;-----------------------------------------------------------
 2:  ; mult8 multiplies two 8-bit unsigned numbers passed on to
 3:  ; it in registers AL and BL. The 16-bit result is returned
 4:  ; in AX. This procedure uses only SHL instruction to do the
 5:  ; multiplication. All registers, except AX, are preserved.
 6:  ; Demonstrates the use of bit instructions BSF and BTC.
 7:  ;-----------------------------------------------------------
 8:  mult8  PROC
 9:         push    CX          ; save registers
10:         push    DX
11:         push    SI
12:         xor     DX,DX       ; DX = 0 (keeps mult. result)
13:         mov     SI,AX       ; save original number in SI
14:  repeat1:
15:         bsf     CX,BX       ; returns first 1 bit position in CX
16:         jz      skip1       ; if ZF=1, no 1 bit in BX - done
17:         mov     AX,SI       ; else, AX = number1*bit weight
18:         shl     AX,CL
19:         add     DX,AX       ; update running total in DX
20:         btc     BX,CX       ; complement the bit found by BSF
21:         jmp     repeat1
22:  skip1:
23:         mov     AX,DX       ; move final result into AX
24:         pop     SI          ; restore registers
25:         pop     DX
26:         pop     CX
27:         ret
28:  mult8  ENDP
```

The modified loop (lines 14 to 21) replaces the loop in the previous version. This code is more efficient because the number of times the loop iterates is equal to the number of 1 bits in BX. The previous version, on the other hand, always iterates seven times. Also note that we can replace the `btc` instruction on line 20 by a `btr` instruction. Similarly, the `bsf` instruction on line 15 can be replaced by a `bsr` instruction. □

12.8 String Instructions

Even though the instructions we discuss here are called string instructions, they are not used just for string processing. In fact, these instructions can be used for block movement of data.

12.8.1 String Representation

A string can be represented either as a *fixed-length* string or as a *variable-length* string. In the fixed-length representation, each string occupies exactly the same number of character positions. That is, each string has the same length, where the length refers to the number of characters in the string. In such a representation, if a string has fewer characters, it is extended by padding, for example, with blank characters. On the other hand, if a string has more characters, it is usually truncated to fit the storage space available.

Clearly, if we want to avoid truncation of larger strings, we need to fix the string length carefully so that it can accommodate the largest string. In practice, it may be difficult to guess this value. A further disadvantage is that memory space is wasted if the majority of strings are shorter than the fixed length used.

The variable-length representation avoids these problems. In this scheme, a string can have as many characters as required (usually, within some system-imposed limit). Associated with each string, there is a string length attribute giving the number of characters in the string. The length attribute is given in one of two ways:

1. Explicitly storing the string length, or
2. Using a sentinel character.

These two methods are discussed next.

Explicitly Storing String Length

In this method, the string length attribute is explicitly stored along with the string, as shown in the following example:

```
string    DB    'Error message'
str_len   DW    $ - string
```

where `$` is the location counter symbol that represents the current value of the location counter. In this example, `$` points to the byte after the last character of `string`. Therefore,

```
$ - string
```

gives the length of the string. Of course, we could also write

```
string    DB    'Error message'
str_len   DW    13
```

However, if we modify the contents of `string` later, we have to update the string length value as well. On the other hand, by using `$ - string`, we let the assembler do the job for us at assembly time.

Using a Sentinel Character

In this method, strings are stored with a trailing sentinel character. Therefore, there is no need to store string length explicitly. The assumption here is that the sentinel character is a special character that cannot appear within a string. We normally use a special, nonprintable character as the sentinel character. We have been using the ASCII NULL character (00H) to terminate strings. Such NULL-terminated strings are called *ASCIIZ strings*. Here are two example strings:

```
string1    DB    'This is OK',0
string2    DB    'Price = $9.99',0
```

The zero at the end represents the ASCII NULL character. The C language, for example, uses this representation to store strings. In the remainder of this chapter, we will use this representation for storing strings.

12.8.2 String Instructions

The Pentium provides five main string-processing instructions. These can be used to copy a string, to compare two strings, and so on. The five basic instructions are as follows:

Mnemonic	Meaning	Operand(s) required
LODS	Load string	Source
STOS	Store string	Destination
MOVS	Move string	Source and destination
CMPS	Compare strings	Source and destination
SCAS	Scan string	Destination

Specifying Operands: As indicated, each string instruction requires a source operand, a destination operand, or both. For 32-bit segments, string instructions use the ESI and EDI registers to point to the source and destination operands, respectively. The source operand is assumed to be at DS:ESI in memory, and the destination operand at ES:EDI in memory. For 16-bit segments, the SI and DI registers are used instead of the ESI and EDI registers. If both operands are in the same data segment, we can let both DS and ES point to the data segment to use the string instructions. String instructions do not allow segment override prefixes.

Variations: Each string instruction can operate on 8-, 16-, or 32-bit operands. As part of execution, string instructions automatically update (i.e., increment or decrement) the index register(s) used by them. For byte operands, source and destination index registers are updated by one. These registers are updated by two and four for word and doubleword operands, respectively. In this chapter, we focus on byte operand strings. String instructions derive much of their power from the fact that they can accept a repetition prefix to repeatedly execute the operation. These prefixes are discussed next. The direction of string processing—forward or backward—is controlled by the direction flag.

Repetition Prefixes

There are three prefixes that fall into two categories: *unconditional* or *conditional* repetition. These are as follows:

Unconditional repeat `rep`	REPeat
Conditional repeat `repe/repz`	REPeat while Equal REPeat while Zero
`repne/repnz`	REPeat while Not Equal REPeat while Not Zero

None of the flags are affected by these instructions.

rep

This is an unconditional repeat prefix and causes the instruction to repeat according to the value in the CX register. The semantics of `rep` are as follows:

while (CX $\neq$ 0)
    execute the string instruction;
    CX := CX–1;
end while

The CX register is first checked and if it is not 0, only then is the string instruction executed. Thus, if CX is 0 to start with, the string instruction is not executed at all. This is in contrast to the `loop` instruction, which first decrements and then tests if CX is 0. Thus, with `loop`, CX = 0 results in a maximum number of iterations, and usually a `jcxz` check is needed.

repe/repz

This is one of the two conditional repeat prefixes. Its operation is similar to that of `rep` except that repetition is also conditional on the zero flag (ZF), as shown below:

while (CX $\neq$ 0)
    execute the string instruction;
    CX := CX–1;
    if (ZF = 0)
    then
        exit loop
    end if
end while

The maximum number of times the string instruction is executed is determined by the contents of CX, as in the `rep` prefix. But the actual number of times the instruction is repeated

is determined by the status of ZF. As shown later, conditional repeat prefixes are useful with `cmps` and `scas` string instructions.

repne/repnz

This prefix is similar to the `repe/repz` prefix except that the condition tested for termination is ZF = 1.

```
while (CX ≠ 0)
    execute the string instruction;
    CX := CX–1;
    if (ZF = 1)
    then
        exit loop
    end if
end while
```

Direction Flag

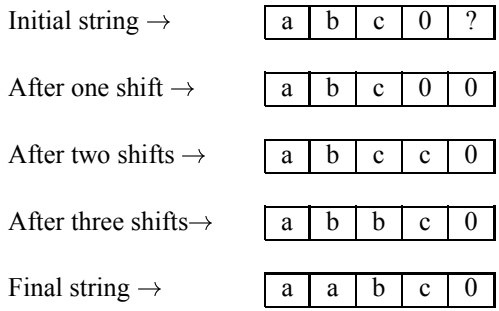
The direction of a string operation depends on the value of the direction flag. Recall that this is one of the bits of the flag's register (see page 259). If the direction flag (DF) is clear (i.e., DF = 0), string operations proceed in the forward direction (from head to tail of a string); otherwise, string processing is done in the opposite direction.

Two instructions are available to explicitly manipulate the direction flag:

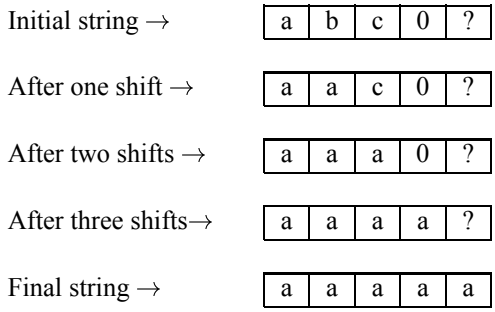
`std`	set direction flag (DF = 1)
`cld`	clear direction flag (DF = 0)

Neither of these instructions requires any operands. Each instruction is encoded using a single byte and takes two clock cycles to execute.

Usually it does not matter whether the string processing direction is forward or backward. For sentinel character-terminated strings, the forward direction is preferred. However, there are situations where one particular direction is mandatory. For example, if we want to shift a string right by one position, we have to start with the tail and proceed toward the head (i.e., in the backward direction) as in the following example:



If we proceed in the forward direction, only the first character is copied through the string, as shown below:



String Move Instructions

There are three basic instructions in this group: `movs`, `lods`, and `stos`. Each instruction can take one of four forms. We start our discussion with the first instruction.

Move a String (movs): The `movs` instruction can be written in one of the following formats:

```
movs    dest_string,source_string
movsb
movsw
movsd
```

Using the first form, we can specify the source and destination strings. This specification will be sufficient to determine whether it is a byte, word, or doubleword operand. However, this form is not used frequently.

In the other three forms, the suffix `b`, `w`, or `d` is used to indicate byte, word, or doubleword operands. This format applies to all the string instructions.

```
movsb — move a byte string
    ES:DI := (DS:SI)    ; copy a byte
    if (DF = 0)         ; forward direction
    then
        SI := SI+1
        DI := DI+1
    else                ; backward direction
        SI := SI–1
        DI := DI–1
    end if
Flags affected: none
```

The `movs` instruction is used to copy a value (byte, word, or doubleword) from the source string to the destination string. As mentioned earlier, DS:SI points to the source string and

ES:DI to the destination string. After copying, the SI and DI registers are updated according to the value of the direction flag and the operand size. Thus, before executing the `movs` instruction, all four registers should be set up appropriately. (This is necessary even if you use the first format.) Note that our focus is on 16-bit segments. For 32-bit segments, we have to use ESI and EDI registers.

For word and doubleword operands, the index registers are updated by two and four, respectively. This instruction, along with the `rep` prefix, is useful for copying a string. More generally, we can use them to perform memory-to-memory block transfers. Here is an example that copies `string1` to `string2`.

```
.DATA
string1     DB     'The original string',0
strLen      EQU    $ - string1
string2     DB     80 DUP (?)
.CODE
    .STARTUP
    mov     AX,DS           ; set up ES
    mov     ES,AX           ;  to the data segment
    mov     CX,strLen       ; strLen includes NULL
    mov     SI,OFFSET string1
    mov     DI,OFFSET string2
    cld                     ; forward direction
    rep     movsb
```

To make ES point to the data segment, we need to copy the contents of DS into ES. Since the Pentium does not allow the instruction

```
mov     ES,DS
```

we have to use a temporary register (we are using AX) for this purpose. Since the `movs` instruction does not change any of the flags, conditional repeat (`repe` or `repne`) should not be used with this instruction.

Load a String (lods): This instruction copies the value at DS:SI from the source string to AL (for byte operands, `lodsb`), AX (for word operands, `lodsw`), or EAX (for doubleword operands, `lodsd`).

`lodsb` — load a byte string

```
        AL := (DS:SI)       ; copy a byte
        if (DF = 0)         ; forward direction
        then
            SI := SI+1
        else                ; backward direction
            SI := SI−1
        end if
```

Flags affected: none

Use of the `rep` prefix does not make sense, as it will leave only the last value in AL, AX, or EAX. This instruction, along with the `stos` instruction, is often used when processing is required while copying a string. This point is elaborated upon after describing the `stos` instruction.

Store a String (stos): This instruction performs the complementary operation. It copies the value in AL (for `stosb`), AX (for `stosw`), or EAX (for `stosd`) to the destination string (pointed to by ES:DI) in memory.

`stosb` — store a byte string

```
ES:DI := AL          ; copy a byte
if (DF = 0)          ; forward direction
then
    DI := DI+1
else                 ; backward direction
    DI := DI−1
end if
```

Flags affected: none

We can use the `rep` prefix with the `stos` instruction if our intention is to initialize a block of memory with a specific character, word, or doubleword value. For example, the code

```
.DATA
array1    DW    100 DUP (?)
.CODE
    .STARTUP
    mov    AX,DS          ; set up ES
    mov    ES,AX          ;  to the data segment
    mov    CX,100
    mov    DI,OFFSET array1
    mov    AX,-1
    cld                   ; forward direction
    rep    stosw
```

initializes `array1` with −1. Of course, we could have done the same with

```
array1    DW    100 DUP (-1)
```

at assembly time if we wanted to initialize only once.

In general, the `rep` prefix is not useful with `lods` and `stos` instructions. These two instructions are often used in a loop to do value conversions while copying data. For example, if `string1` only contains letters and blanks,

```
    mov    CX,strLen
    mov    SI,OFFSET string1
    mov    DI,OFFSET string2
```

```
    cld                     ; forward direction
loop1:
    lodsb
    or    AL,20H
    stosb
    loop  loop1
done:
          . . .
```

can convert it to a lowercase string. Note that blank characters are not affected because 20H represents blank in ASCII, and the

```
or    AL,20H
```

instruction does not have any effect on it. The advantage of `lods` and `stos` is that they automatically increment SI and DI registers.

String Compare Instruction: The `cmps` instruction can be used to compare two strings.

`cmpsb` — compare two byte strings
Compare the two bytes at DS:SI and ES:DI and set flags
if (DF = 0) ; forward direction
then
SI := SI+1
DI := DI+1
else ; backward direction
SI := SI−1
DI := DI−1
end if
Flags affected: As per `cmp` instruction

The `cmps` instruction compares the two bytes, words, or doublewords at DS:SI and ES:DI and sets the flags just as the `cmp` instruction by performing

```
(DS:SI) – (ES:DI)
```

We can use conditional jumps such as `ja`, `jg`, `jc`, and the like to test the relationship of the two values. As usual, SI and DI registers are updated according to the value of the direction flag and operand size. The `cmps` instruction is typically used with the `repe/repz` or the `repne/repnz` prefix.

The code

```
.DATA
string1    DB    'abcdfghi',0
strLen     EQU   $ - string1
```

```
string2    DB    'abcdefgh',0
.CODE
    .STARTUP
    mov    AX,DS          ; set up ES
    mov    ES,AX          ;  to the data segment
    mov    CX,strLen
    mov    SI,OFFSET string1
    mov    DI,OFFSET string2
    cld                   ; forward direction
    repe   cmpsb
```

leaves SI pointing to `g` in `string1` and DI to `f` in `string2`. Therefore, adding

```
    dec    SI
    dec    DI
```

leaves SI and DI pointing to the first character that differs. Then we can use, for example,

```
    ja    str1Above
```

to test if `string1` is greater (in the collating sequence) than `string2`. This, of course, is true in this example.

To find the first matching instance, we can use `repne/repnz`. These prefixes make `cmps` continue comparison as long as the comparison fails; the loop terminates when a match is found. For example,

```
.DATA
string1    DB    'abcdfghi',0
strLen     EQU   $ - string1 - 1
string2    DB    'abcdefgh',0
.CODE
    .STARTUP
    mov    AX,DS        ; set up ES
    mov    ES,AX        ;  to the data segment
    mov    CX,strLen
    mov    SI,OFFSET string1 + strLen - 1
    mov    DI,OFFSET string2 + strLen - 1
    std                 ; backward direction
    repne  cmpsb
    inc    SI
    inc    DI
```

leaves SI and DI pointing to the first character that matches in the backward direction.

Scanning a String: The `scas` (scanning a string) instruction is useful in searching for a particular value or character in a string. The value should be in AL (for `scasb`), AX (for `scasw`), or EAX (for `scasd`), and ES:DI should point to the string to be searched.

`scasb` — scan a byte string

```
Compare AL to the byte at ES:DI and set flags
if (DF = 0)          ; forward direction
then
    DI := DI+1
else                 ; backward direction
    DI := DI−1
end if
```

Flags affected: As per `cmp` instruction

As with the `cmps` instruction, the `repe/repz` or the `repne/repnz` prefix can be used.

```
.DATA
string1     DB     'abcdefgh',0
strLen      EQU    $ - string1
.CODE
    .STARTUP
    mov     AX,DS          ; set up ES
    mov     ES,AX          ;  to the data segment
    mov     CX,strLen
    mov     DI,OFFSET string1
    mov     AL,'e'         ; character to be searched
    cld                    ; forward direction
    repne   scasb
    dec     DI
```

This program leaves DI pointing to `e` in `string1`. The following example can be used to skip initial blanks.

```
.DATA
string1     DB     '      abc',0
strLen      EQU    $ - string1
.CODE
    .STARTUP
    mov     AX,DS          ; set up ES
    mov     ES,AX          ;  to the data segment
    mov     CX,strLen
    mov     DI,OFFSET string1
    mov     AL,' '         ; character to be searched
    cld                    ; forward direction
    repe    scasb
    dec     DI
```

This program leaves DI pointing to the first nonblank character (`a` in the example) in `string1`.

12.8.3 String Processing Examples

We now give some examples that illustrate the use of the string instructions discussed in this chapter. All these procedures are available in the `string.asm` file. These procedures receive the parameters via the stack. A string pointer is received in `segment:offset` form (i.e., two words from the stack), which is loaded into either DS:SI or ES:DI using `lds` or `les` instructions. Details on these instructions are given next.

LDS and LES Instructions

The syntax of these instructions is

```
lds    register,source
les    register,source
```

where `register` should be a 16-bit general-purpose register, and `source` is a pointer to a 32-bit memory operand. The instructions perform the following actions:

```
lds
     register = (source)
           DS = (source + 2)
les
     register = (source)
           ES = (source + 2)
```

The 16-bit value at `source` is copied to `register` and the next 16-bit value (i.e., at `source`+2) is copied to the DS or ES register. Both instructions affect none of the flags. By specifying SI as the register operand, `lds` can be conveniently used to set up the source string. Similarly, the destination string can be set up by specifying DI with `les`. For completeness, you should note that the Pentium also supports `lfs`, `lgs`, and `lss` instructions to load the other segment registers.

Examples

Next we give two simple string-processing procedures. These procedures use the carry flag (CF) to report *not a string* error. This error results if the input given to the procedure is not a string with length less than the constant `STR_MAX` defined in `string.asm`. The carry flag is set if there is an input error; otherwise, it is cleared.

The following constants are defined in `string.asm`:

```
STR_MAX    EQU    128
STRING1    EQU    DWORD PTR [BP+4]
STRING2    EQU    DWORD PTR [BP+8]
```

Example 12.20 *Write a procedure* `str_len` *to return the string length.*

String length is the number of characters in a string, excluding the NULL character. We use the `scasb` instruction and search for the NULL character. Since `scasb` works on the destination

string, `les` is used to load the string pointer into ES and DI registers from the stack. `STR_MAX`, the maximum length of a string, is moved into CX, and the NULL character (i.e., 0) is moved into the AL register. The direction flag is cleared to initiate a forward search. The string length is obtained by taking the difference between the end of the string (pointed to by DI) and the start of the string available at [BP+4]. The AX register is used to return the string length.

```
;-------------------------------------------------------
;String length procedure. Receives a string pointer
;(seg:offset) via the stack. If not a string, CF is set;
;otherwise, string length is returned in AX with CF = 0.
;Preserves all registers.
;-------------------------------------------------------
str_len PROC
        enter   0,0
        push    CX
        push    DI
        push    ES
        les     DI,STRING1 ; copy string pointer to ES:DI
        mov     CX,STR_MAX ; needed to terminate loop if BX
                           ;  is not pointing to a string
        cld                ; forward search
        mov     AL,0       ; NULL character
        repne   scasb
        jcxz    sl_no_string  ; if CX = 0, not a string
        dec     DI         ; back up to point to NULL
        mov     AX,DI
        sub     AX,[BP+4]  ; string length in AX
        clc                ; no error
        jmp     SHORT sl_done
sl_no_string:
        stc                ; carry set => no string
sl_done:
        pop     ES
        pop     DI
        pop     CX
        leave
        ret     4          ; clear stack and return
str_len ENDP
```

Example 12.21 *Write a procedure* `str_mov` *to move a string (*`string1`*) left or right by* `num` *number of positions.*

The objective of this example is to show how a particular direction of string copying is important. This procedure receives a pointer to `string1` and an integer `num` indicating the number of positions the string value is to be moved *within* the string. A positive `num` value is treated as a move to the right and a negative value as a move to the left. A 0 value has no effect. Note

that the pointer received by this function need not point to the beginning of `string1`. It is important to make sure that there is enough room in the original string in the intended direction of the move.

```
;-------------------------------------------------------------
;String move procedure. Receives a signed integer
;and a string pointer (seg:offset) via the stack.
;The integer indicates the number of positions to move
;the string:
;       -ve number => left move
;       +ve number => right move
;If string1 is not a string, CF is set;
;otherwise, string is moved left or right and returns
;a pointer to the modified string in AX with CF = 0.
;Preserves all registers.
;-------------------------------------------------------------
str_mov PROC
        enter   0,0
        push    CX
        push    DI
        push    SI
        push    DS
        push    ES
        ; find string length first
        lds     SI,STRING1  ; string pointer
        push    DS
        push    SI
        call    str_len
        jnc     sv_skip1
        jmp     sv_no_string
sv_skip1:
        mov     CX,AX       ; string length in CX
        inc     CX          ; add 1 to include NULL
        les     DI,STRING1
        mov     AX,[BP+8]   ; copy # of positions to move
        cmp     AX,0        ; -ve number => left move
        jl      move_left   ; +ve number => right move
        je      finish      ; zero => no move
move_right:
        ; prepare SI and DI for backward copy
        add     SI,CX       ; SI points to the
        dec     SI          ;  NULL character
        mov     DI,SI       ; DI = SI + # of positions to move
        add     DI,AX
        std                 ; backward copy
        rep     movsb
```

```
        ; now erase the remainder of the old string
        ;  by writing blanks
        mov     CX,[BP+8]  ; # of positions moved
        ; DI points to the first char of left-over string
        mov     AL,' '     ; blank char to fill
        ; direction flag is set previously
        rep     stosb
        jmp     SHORT finish
move_left:
        add     DI,AX
        cld                ; forward copy
        rep     movsb
finish:
        mov     AX,[BP+8]  ; add # of positions to move
        add     AX,[BP+4]  ; to string pointer (ret value)
        clc                ; no error
        jmp     SHORT sv_done
sv_no_string:
        stc                ; carry set => no string
sv_done:
        pop     ES
        pop     DS
        pop     SI
        pop     DI
        pop     CX
        leave
        ret     6          ; clear stack and return
str_mov ENDP
```

To move left, we let SI point to the same character of `string1` as the pointer received by the procedure. We set DI = SI + `num`. Since `num` is negative for a left move, DI points to where the character pointed by SI should move. A simple forward copy according to the string length (plus one) will move the string value. The extraneous characters left will not cause any problems, as shown below,

`string1` before `str_mov`
␣␣␣␣`abcd0`
`string1` after `str_mov` with the string pointing to `a` and `num` = -2
␣␣`abcd0d0`

where ␣ indicates a blank.

To move right, we let SI point to the NULL character of `string1` and DI to its right by `num` positions. A straightforward copy in the backward direction will move the string to its destination position. However, this leaves remnants of the old values on the left, as shown in the following example:

```
string1 before str_mov
   ␣␣abcd0␣␣
string1 after str_mov with the string pointing to a and num = 2
   ␣␣ababcd0
```

To eliminate this problem, `str_mov` erases the contents of the remaining characters of the original value by filling them with blanks. In this example, the first `ab` characters will be filled with blanks.

12.8.4 Testing String Procedures

Now we turn our attention to testing the string procedures developed in the last section. A partial listing of this program is given in Program 12.7. You can find the full program in the `str_test.asm` file.

Our main interest in this section is to show how an indirect procedure call would substantially simplify calling the appropriate procedure according to user selection. Let us first look at the indirect call instruction for 16-bit segments.

Indirect Procedure Call

In our discussions so far, we have been using only direct procedure calls, where the offset of the target procedure is provided directly. In indirect procedure calls, this offset is given with one level of indirection as in the indirect jump (see Section 12.3.1). That is, the call instruction itself will contain either a memory address (through a label), or a 16-bit general-purpose register. The actual offset of the target procedure is obtained either from the memory or register. For example, we could use

```
call    BX
```

if BX contains the offset of the target procedure. When this `call` instruction is executed, the BX register contents are used to load IP in order to transfer control to the target procedure. Similarly, we can use

```
call    target_proc_ptr
```

if the word in memory at `target_proc_ptr` contains the offset of the target procedure.

Back to the Example

To facilitate calling the appropriate string procedure, we keep the procedure pointers in the `proc_ptr_table` table. The user query response is used as an index into this table to get the target procedure offset. The BX register is used as the index into this table. The instruction

```
call    proc_ptr_table[BX]
```

causes the indirect procedure call. The rest of the program is straightforward.

Program 12.7 String test program `str_test.asm`

```
                    . . .
.DATA
proc_ptr_table  DW  str_len_fun,str_cpy_fun,str_cat_fun
                DW  str_cmp_fun,str_chr_fun,str_cnv_fun
                DW  str_mov_fun
MAX_FUNCTIONS   EQU ($ - proc_ptr_table)/2

choice_prompt   DB  'You can test several functions.',CR,LF
                DB  '    To test       enter',CR,LF
                DB  'String length       1',CR,LF
                DB  'String copy         2',CR,LF
                DB  'String concatenate  3',CR,LF
                DB  'String compare      4',CR,LF
                DB  'Locate character    5',CR,LF
                DB  'Convert string      6',CR,LF
                DB  'Move string         7',CR,LF
                DB  'Invalid response terminates program.',CR,LF
                DB  'Please enter your choice: ',0

invalid_choice  DB  'Invalid choice - program terminates.',0

string1         DB  STR_MAX DUP (?)
string2         DB  STR_MAX DUP (?)
                    . . .
.CODE
                    . . .
main    PROC
        .STARTUP
        mov     AX,DS
        mov     ES,AX

query_choice:
        xor     BX,BX
        PutStr  choice_prompt    ; display menu
        GetCh   BL               ; read response
        nwln
        sub     BL,'1'
        cmp     BL,0
        jb      invalid_response
        cmp     BL,MAX_FUNCTIONS
        jb      response_ok
invalid_response:
        PutStr  invalid_choice
        jmp     SHORT done
```

```
response_ok:
        shl     BL,1                ; multiply BL by 2
        call    proc_ptr_table[BX]  ; indirect call
        jmp     query_choice
done:
        .EXIT
main    ENDP
                    . . .
        END     main
```

12.9 Summary

We have discussed the utility of the six status flags in detail. In particular, these flags are useful in supporting conditional execution. We have presented details of the Pentium's multiplication and division instructions. Both instructions support operations on signed and unsigned integers.

The Pentium supports a variety of unconditional and conditional jump instructions. We have introduced some of these instructions in Chapter 9. Here we looked at the indirect jump and conditional jump instructions. We have presented details on selection and iterative constructs in order to see how the jump instructions are useful in implementing these high-level language constructs. In particular, we have seen how compilers use the assembly instructions to implement these high-level language constructs. Similarly, we have presented details on logical expressions.

The Pentium supports several string instructions. These instructions are useful not only for manipulating strings but also for moving blocks of data. By using the repeat prefixes, we can efficiently implement string manipulation and block movement.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Auxiliary flag
- Bit manipulation
- Carry flag
- Conditional jump
- Direction flag
- Indirect jump
- Indirect procedure call
- Linear search
- Logical expressions—full evaluation
- Logical expressions—partial evaluation
- Overflow flag
- Parity flag
- Selection sort
- Sign flag
- Status flags
- String representation
- Zero flag

12.10 Exercises

12–1 What is the significance of the carry flag?

12–2 What is the significance of the overflow flag?

12–3 Suppose the sign flag is not available. Is there a way to detect the sign of a number? Is there more than one way?

12–4 When is the parity flag set? What is a typical application that uses this flag?

12–5 When subtracting two numbers, suppose the carry flag is set. What does it imply in terms of the relationship between the two numbers?

12–6 In the last example, suppose the overflow flag is set. What does it imply in terms of the relationship between the two numbers?

12–7 Is it possible to set both the carry and zero flags? If so, give an example that could set both these flags; otherwise, explain why not.

12–8 Is it possible to set both the overflow and zero flags? If so, give an example that could set both these flags; otherwise, explain why not.

12–9 When the zero flag is set, the parity flag is also set. The converse, however, is not true. Explain with examples why this is so.

12–10 The zero flag is useful in implementing countdown loops (loops in which the counting variable is decremented until zero). Justify the statement by means of an example.

12–11 Fill in the blanks in the following table:

	AL	CF	ZF	SF	OF	PF
`mov AL,127` `add AL,-128`						
`mov AL,127` `sub AL,-128`						
`mov AL,-1` `add AL,1`						
`mov AL,127` `inc AL`						
`mov AL,127` `neg AL`						
`mov AL,0` `neg AL`						

You do not have to fill in the lines with the `mov` instruction. The AL column represents the AL value after executing the corresponding instruction.

12–12 Fill in the blanks in the following table:

Instruction	Before execution		After execution			
	AL	BL	AL	ZF	SF	PF
`and AL,BL`	79H	86H				
`or AL,BL`	79H	86H				
`xor AL,BL`	79H	86H				
`test AL,BL`	79H	86H				
`and AL,BL`	36H	24H				
`or AL,BL`	36H	24H				
`xor AL,BL`	36H	24H				
`test AL,BL`	36H	24H				

12–13 Assuming that the value in AL is a signed number, fill in the blanks in the following table:

Instruction	Before execution		After execution	
	AL	CF	AL	CF
`shl AL,1`	−1	?		
`rol AL,1`	−1	?		
`shr AL,1`	50	?		
`ror AL,1`	50	?		
`sal AL,1`	−20	?		
`sar AL,1`	−20	?		
`rcl AL,1`	−20	1		
`rcr AL,1`	−20	1		

12–14 Assuming that the CL register is initialized to three, fill in the blanks in the following table:

Instruction	Before execution		After execution	
	AL	CF	AL	CF
`shl AL,CL`	76H	?		
`sal AL,CL`	76H	?		
`rcl AL,CL`	76H	1		
`rcr AL,CL`	76H	1		
`ror AL,CL`	76H	?		
`rol AL,CL`	76H	?		

12–15 Explain why multiplication requires two separate instructions to work on signed and unsigned data.

12–16 We have stated that, if we use double-length registers, multiplication does not result in an overflow. Prove this statement for 8-, 16-, and 32-bit operands.

12–17 We have discussed how the ZF, OF, and SF flags can be used to establish relationships such as $<$ and $>$ between two signed numbers (see Table 12.6 on page 505). Show that the following conditions are equivalent:

	Condition given in Table 12.6	Equivalent condition
`jg`	ZF = 0 and SF = OF	((SF xor OF) or ZF) = 0
`jge`	SF = OF	(SF xor OF) = 0
`jl`	SF $\neq$ OF	(SF xor OF) = 1
`jle`	ZF = 1 or SF $\neq$ OF	((SF xor OF) or ZF) = 1

12–18 What are the advantages and disadvantages of the fixed-length string representation?

12–19 What are the advantages and disadvantages of the variable-length string representation?

12–20 Discuss the pros and cons of storing the string length explicitly versus using a sentinel character for storing variable-length strings.

12–21 We can write procedures to perform string operations without using the string instructions. What is the advantage of using the string instructions? Explain why.

12–22 Why doesn't it make sense to use the `rep` prefix with the `lods` instruction?

12–23 Explain why it does not make sense to use conditional repeat prefixes with `lods`, `stos`, or `movs` string instructions.

12–24 Both `loop` and repeat prefixes use the CX register to indicate the repetition count. Yet there is one significant difference between them in how they use the CX value. What is this difference?

12–25 Identify a situation in which the direction of string processing is important.

12–26 Identify a situation in which a particular direction of string processing is mandatory.

12–27 Suppose that the `lds` instruction is not supported by the Pentium. Write a piece of code that implements the semantics of the `lds` instruction. Make sure that your code does not disturb any other registers.

12–28 Compare the space and time requirements of `lds` and the code you have written in the last exercise. To do this exercise, you need to refer to the Pentium data book.

12–29 What is the difference between the direct and indirect procedure calls?

12.11 Programming Exercises

12–P1 Write a program to multiply two signed 8-bit numbers using only shift and add instructions. Your program can read the two input numbers with `GetInt` and display the result by `PutInt`.

12–P2 In Appendix A, we discuss the format of short floating-point numbers. Write a program that reads the floating-point internal representation from the user as a string of eight hexadecimal digits and displays the three components—mantissa, exponent, and sign—in binary. For example, if the input to the program is 429DA000, the output should be

sign = 0
mantissa = 1.0011101101
exponent = 110.

12–P3 Modify the program for the last exercise to work with the long floating-point representation.

12–P4 Suppose you are given an integer that requires 16 bits to store. You are asked to find whether its binary representation has an odd or even number of 1s. Write a program that reads an integer (it should accept both positive and negative numbers) from the user and outputs whether it contains an odd or even number of 1s. Your program should also print the number of 1s in the binary representation.

12–P5 Modify the indirect jump program given in Program 12.3 on page 498 so that it works for any input without hanging up or crashing the system. That is, make the program safe to run.

12–P6 Suppose you are given a positive integer. You can add individual digits of this number to get another integer. Now apply the same procedure to the new integer. If we repeat this procedure, eventually we will end up with a single digit. Here is an example:

7391928 = 7 + 3 + 9 + 1 + 9 + 2 + 8 = 39
39 = 3 + 9 = 12
12 = 1 + 2 = 3.

Write a program that reads a positive integer from the user and displays the single digit as obtained by the above procedure. For the example, the output should be 3.

Your program should detect negative number input as an error and terminate after displaying an appropriate error message.

12–P7 Repeat the above exercise with the following modification. Use multiplication instead of addition. Here is an example:

7391928 = 7 * 3 * 9 * 1 * 9 * 2 * 8 = 27216
27216 = 2 * 7 * 2 * 1 * 6 = 168
168 = 1 * 6 * 8 = 48
48 = 4 * 8 = 32
32 = 3 * 2 = 6.

12–P8 The `PutInt8` procedure has used repeated division by 10. Alternatively, you can display an 8-bit number by first dividing it by 100 and displaying the quotient; then divide the remainder by 10 and display the quotient and remainder (in that order). Modify the `PutInt8` procedure to incorporate this method. Discuss the pros and cons of the two methods.

12–P9 Write a program to multiply a two-dimensional matrix in the following way: multiply all elements in row i by $(-1)^i i$. That is, multiply row 1 by -1, row 2 by $+2$, row 3 by -3, and so on. Your program should be able to read matrices of size up to 10×10. You should query the user for number of rows, number of columns, and then read the matrix element values. These values should be within the range of 8-bit signed numbers (i.e., between -128 to +127). Internally, use words to store the number so that there will not be overflow problems with the multiplication. Make sure to do proper error checking, for example, asking for more than 10 rows or columns, entering an out-of-range value, and so on.

12–P10 We know that

$$1 + 2 + 3 + \cdots + N = \frac{N \times (N + 1)}{2} .$$

Write a program that requests N as input and computes the left- and the right-hand sides of the equation, verifies that they are equal, and displays the result.

12–P11 Write a program that reads a set of test scores as input and outputs the truncated average value (i.e., discard any fraction generated). The input test scores cannot be negative. So use this condition to terminate the input. Furthermore, assume that the first number entered is not the test score but the maximum score that can be obtained for that test. Use this information to display the average test score as a percentage. For example, if the average is 18 and the maximum obtainable test score is 20, the average is 90 percent.

12–P12 Modify the above program to round the average test score. For example, if the average is 15.55, it should be rounded to 16.

12–P13 Modify the average test score program to display the fractional part as well. Display the average test score in `dd.dd` format.

12–P14 Write a program to convert temperature from Celsius to Fahrenheit. The formula is

$$F = \frac{9}{5} \times C + 32 .$$

12–P15 Write a program to read length L, width W, and height H of a box (all integers). It computes and displays the volume and surface area of the box.

$$\begin{aligned} \text{Volume} &= L \times W \times H, \\ \text{Surface volume} &= 2 \times (L \times H + L \times W + W \times H) . \end{aligned}$$

12–P16 Write an assembly language program to read a string of characters from the user. It counts the number of vowels and displays the value. For each vowel, the count includes both uppercase and lowercase letters. For example, the input string

Advanced Programming in UNIX Environment

produces the following output:

Vowel	Count
a or A	3
e or E	3
i or I	4
o or O	2
u or U	1

12–P17 Do the last exercise using an indirect jump. *Hint:* Use `xlat` to translate vowels to five consecutive numbers so that you can use the number as an index into the jump table.

12–P18 Suppose that we want to list uppercase and lowercase vowels separately (i.e., a total of 10 count values). Modify the programs of the last two exercises to do this. After doing this exercise, express your opinion on the usefulness of the indirect jump instruction.

12–P19 Merge sort is a technique to combine two sorted arrays. Merge sort takes two sorted input arrays `X` and `Y`—say, of size m and n—and produces a sorted array `Z` of size $m + n$ that contains all elements of the two input arrays. The pseudocode of merge sort is as follows:

```
mergesort (X, Y, Z, m, n)
    i := 0 {index variables for arrays X, Y, and Z}
    j := 0
    k := 0
    while ((i < m) AND (j < n))
        if (X[i] ≤ Y[j]) {find largest of two}
        then
            Z[k] := X[i] {copy and update indices}
            k := k+1
            i := i+1
        else
            Z[k] := Y[j] {copy and update indices}
            k := k+1
            j := j+1
        end if
    end while
    if (i < m) {copy remainder of input array}
        while (i < m)
            Z[k] := X[i]
            k := k+1
            i := i+1
        end while
    else
        while (j < m)
```

```
                Z[k] := Y[j]
                k := k+1
                j := j+1
            end while
        end if
    end mergesort
```

The merge sort algorithm scans the two input arrays while copying the smallest of the two elements from X and Y into Z. It updates indices appropriately. The first while loop terminates when one of the arrays is exhausted. Then the other array is copied into Z. Write a merge sort procedure and test it with two sorted arrays. Assume that the user will enter the two input arrays in sorted (ascending) order. The merge sort procedure receives the five parameters via the stack.

Chapter 13

High-Level Language Interface

Objectives

- To review motivation for writing mixed-mode programs;
- To discuss the principles of mixed-mode programming;
- To describe how assembly language procedures are called from C;
- To illustrate how C functions are called from assembly language procedures;
- To explain how inline assembly language code is written.

Thus far, we have written standalone assembly programs. This chapter considers mixed-mode programming. In this mode, part of a program is written in a high-level language and part in assembly language. We use C and Pentium assembly languages to illustrate how such mixed-mode programs are written. The motivation for mixed-mode programming is discussed in Section 13.1. Section 13.2 gives an overview of mixed-mode programming, which can be done either by inline assembly code or by separate assembly modules. The inline assembly method is discussed in Section 13.5. Other sections focus on the separate assembly module method.

Section 13.3 describes the mechanics involved in calling assembly language procedures from a C program. This section presents details about parameter passing, returning values to C functions, and so on. Section 13.4 shows how a C function can be called from an assembly language procedure. The last section summarizes the chapter.

13.1 Why Program in Mixed-Mode?

Mixed-mode programming refers to writing parts of a program in different languages. In this chapter we focus on programming in C and assembly languages. Thus, in our case, part of a program is written in C and the other part in the Pentium assembly language. We use the Borland C++ compiler and Turbo Assembler to explain the principles involved in mixed-mode programming. This discussion can be easily extended to a different set of languages and compilers/assemblers.

In Chapter 9, we discussed several reasons why one would want to program in assembly language. Although it is possible to write a program entirely in assembly language, there are several disadvantages in doing so. These include

- Low productivity,
- High maintenance cost, and
- Lack of portability.

Low productivity is due to the fact that assembly language is a low-level language. As a result, a single high-level language instruction may require several assembly language instructions. It has been observed that programmers tend to produce the same number of lines of debugged and tested source code per unit time irrespective of the level of the language used. As the assembly language requires more lines of source code, programmer productivity tends to be low.

Programs written in assembly language are difficult to maintain. This is a direct consequence of it being a low-level language. In addition, assembly language programs are not portable.

As a result of these pros and cons, some programs are written in mixed-mode using both high-level and low-level languages. System software often requires mixed-mode programming. In such programs, it is possible for a high-level procedure to call a low-level procedure and vice versa. The remainder of the chapter discusses how mixed-mode programming is done in C and assembly languages. Our goal is to illustrate only the principles involved. Once these principles are understood, the discussion can be generalized to any type of mixed-mode programming.

13.2 Overview

There are two ways of writing mixed-mode C and assembly programs: inline assembly code or separate assembly modules. In the inline assembly method, the C program module can contain assembly language instructions. Most C compilers allow embedding assembly language instructions within a C program by prefixing them with **asm** to let the compiler know that it is an assembly language instruction. This method is useful if you have only a small amount of assembly code to be embedded. Otherwise, separate assembly modules are preferred. Section 13.5 discusses how the inline assembly method works with an example. Until then, we focus on separate assembly modules.

When separate modules are used for C and assembly languages, each module can be translated into the corresponding object (`.obj`) file. To do this translation, we use a C compiler for

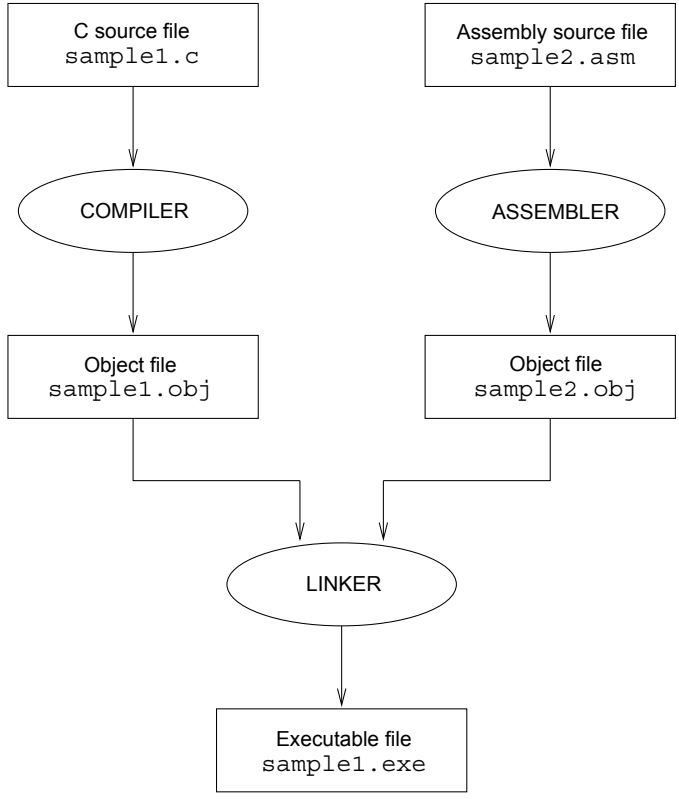


Figure 13.1 Steps involved in compiling mixed-mode programs.

the C modules and an assembler for the assembly modules, as shown in Figure 13.1. Then the linker can be used to produce the executable (`.exe`) file from these object files.

Suppose our mixed-mode program consists of two modules:

- One C module, file `sample1.c`, and
- One assembly module, file `sample2.asm`.

The process involved in producing the executable file is shown in Figure 13.1. We can instruct the Borland C++ compiler to initiate this cycle with the following:

```
bcc sample1.c sample2.asm
```

This command instructs the compiler to first compile `sample1.c` to `sample1.obj` and then invoke the TASM assembler to assemble `sample2.asm` to `sample2.obj`. The linker TLINK is finally invoked to link `sample1.obj` and `sample2.obj` to produce the executable file `sample1.exe`.

Figure 13.2 Two ways of pushing parameters onto the stack.

13.3 Calling Assembly Procedures from C

Let us now discuss how we can call an assembly language procedure from a C program. The first thing we have to know is what communication medium is used between the C and assembly language procedures, as the two procedures may exchange parameters and results. You are right if you guessed it to be the stack.

Given that the stack is used for communication purposes, we still need to know how the C function places the parameters on the stack, and where it expects the assembly language procedure to return the result. In addition, we should also know which registers we can use freely without worrying about preserving their values. Next we discuss these issues in detail.

13.3.1 Parameter Passing

There are two ways in which arguments (i.e., parameter values) are pushed onto the stack: from left to right or from right to left. Most high-level languages such as FORTRAN and Pascal push the arguments from left to right. These are called *left-pusher* languages. C, on the other hand, pushes arguments from right to left. Thus, C is a *right-pusher* language. The stack state after executing

```
sum(a,b,c,d)
```

is shown in Figure 13.2. From now on, we consider only right-pushing of arguments as we focus on the C language.

To see how Borland C++ pushes arguments onto the stack, take a look at the following C program (this is a partial listing of Example 13.1):

```
int main(void)
{
    int     x=25, y=70;
    int     value;
    extern  int test(int, int, int);
```

```
        value = test (x, y, 5);
        . . .
        . . .
    }
```

This program is compiled (use `-S` option to generate the assembly source code) as follows:

```
;           int     x=25, y=70;
;
        mov     word ptr [bp-2],25
        mov     word ptr [bp-4],70
;
;           int     value;
;           extern  int test(int, int, int);
;
;           value = test (x, y, 5);
;
        push    5
        push    word ptr [bp-4]
        push    word ptr [bp-2]
        call    near ptr _test
        add     sp,6
        mov     word ptr [bp-6],ax
```

The compiler assigns space for variables `x`, `y`, and `value` on the stack at BP−2, BP−4, and BP−6, respectively. When the `test` function is called, the arguments are pushed from right to left, starting with the constant 5. Also notice that the stack is cleared of the arguments by the C program after the call by

```
add   sp,6
```

So, when we write our assembly procedures, we should not bother clearing the arguments from the stack as we did in our programs in the previous chapters. This convention is used because C allows a variable number of arguments to be passed in a function call. However, we should note that the arguments are cleared by the called procedure if we use Pascal instead of C. The Borland C++ compiler allows you to specify the desired parameter passing mechanism (C or Pascal). For example, by using the `-p` option to use Pascal calls, the same program is compiled as

```
;           int     x=25, y=70;
;
        mov     si,25
        mov     word ptr [bp-2],70
;
;           int     value;
;           extern  int test(int, int, int);
```

Table 13.1 Registers used to return values

Return value type	Register used
unsigned char	AX
char	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX
long	DX:AX
near pointer	AX
far pointer	DX:AX

```
;
;           value = test (x, y, 5);
;
     push    si
     push    word ptr [bp-2]
     push    5
     call    near ptr TEST
     mov     di,ax
```

We can clearly see that left pushing of arguments is used. In addition, the stack is not cleared of the arguments. Thus, in this case, it is the responsibility of the called procedure to clear the arguments on the stack, which is what we have been doing in our assembly programs in the previous chapters.

13.3.2 Returning Values

We can see from the C and Pascal assembly codes given in the last subsection that the AX register returns the value of the `test` function. In fact, the AX is used to return 8- and 16-bit values. To return a 32-bit value, use the DX:AX pair with the DX holding the upper 16 bits. Table 13.1 shows how various values are returned to the Borland C++ function. This list does not include floats and doubles. These are returned via the 8087 stack. We do not discuss these details.

13.3.3 Preserving Registers

In general, the called assembler procedure can use the registers as needed, except that the following registers should be preserved:

```
BP, SP, CS, DS, SS
```

In addition, if register variables are enabled, both SI and DI registers should also be preserved. When register variables are enabled, both SI and DI registers are used for variable storage, as shown below:

```
;          int     x=25, y=70;
;
     mov     si,25
     mov     word ptr [bp-2],70
;
;          int     value;
;          extern  int test(int, int, int);
;
;          value = test (x, y, 5);
;
     push    5
     push    word ptr [bp-2]
     push    si
     call    near ptr _test
     add     sp,6
     mov     di,ax
```

Compare this version, with register variables enabled, to the previous version given on page 555. Instead of using the stack, SI and DI are used to map variables `x` and `value`, respectively. Since we never know whether the C code was compiled with or without enabling the register variables, it is good practice to preserve SI and DI registers as well.

13.3.4 Publics and Externals

Mixed-mode programming involves at least two program modules: a C module and an assembly module. Thus, we have to declare those functions and procedures that are not defined in the same module as external. Similarly, those procedures that are accessed by another module should be declared as public, as discussed in Chapter 10. Before proceeding further, you may want to review the material on multimodule programs presented in Chapter 10. Here we mention only those details that are specific to the mixed-mode programming involving C and assembly language.

In C, all external labels should start with an underscore character (_). The C and C++ compilers automatically append the required underscore character to all external functions and variables. A consequence of this characteristic is that when we write an assembly procedure that is called from a C program, we have to make sure that we prefix an underscore character to its name.

13.3.5 Illustrative Examples

We now look at three examples to illustrate the interface between C and assembly programs. We start with a simple example, whose C part has been dissected in the previous subsections.

Example 13.1 *Our first mixed-mode example.*
This example passes three parameters to the assembly language function `test`. The C code is shown in Program 13.1 and the assembly code in Program 13.2. Since the `test` procedure is called from the C program, we have to prefix an underscore character to the procedure name. The function `test` is declared as external in the C program (line 11) and public in the assembly program (line 7). Since C clears the arguments from the stack, the assembly procedure uses a simple `ret` to transfer control back to the C program. Other than these differences, the assembly procedure is similar to several others we have written before.

Program 13.1 An example illustrating assembly calls from C: C code (in file `testex_c.c`)

```
 1:  /*********************************************************
 2:   * A simple example to illustrate C and assembly language *
 3:   * interface. The test function is written in assembly    *
 4:   * language (in file testex_a.asm).                       *
 5:   *********************************************************/
 6:  #include <stdio.h>
 7:  int main(void)
 8:  {
 9:        int     x=25, y=70;
10:        int     value;
11:        extern  int test(int, int, int);
12:
13:        value = test (x, y, 5);
14:        printf("result = %d\n", value);
15:        return 0;
16:  }
```

Program 13.2 An example illustrating assembly calls from C: Assembly code (in file `testex_a.asm`)

```
 1:  ;-------------------------------------------------------
 2:  ; Assembly program for the test function - called from the
 3:  ; C program in file testex_c.c
 4:  ;-------------------------------------------------------
 5:  .MODEL SMALL
 6:  .CODE
 7:  .486
 8:  PUBLIC _test
 9:  _test   PROC
10:         enter   0,0
11:         mov     AX,[BP+4] ; get argument1 x
12:         add     AX,[BP+6] ; add argument2 y
13:         sub     AX,[BP+8] ; subtract argument3 from sum
```

```
14:          leave
15:          ret                ; stack cleared by C function
16:  _test   ENDP
17:          END
```

Example 13.2 *An example to show parameter passing by call-by-value as well as call-by-reference.*

This example shows how pointer parameters are handled. The C main function requests three integers and passes them to the assembly procedure. The C program is given in Program 13.3. The assembly procedure `min_max`, shown in Program 13.4, receives the three integer values and two pointers to variables minimum and maximum. It finds the minimum and maximum of the three integers and returns them to the main C function via the two pointer variables. The minimum value is kept in AX and the maximum in DX. The code given on lines 29 to 32 in Program 13.4 stores the return values by using the BX register in the indirect addressing mode.

Program 13.3 An example with the C program passing pointers to the assembly program: C code (in file `minmax_c.c`)

```
 1:  /*********************************************************
 2:   * An example to illustrate call-by-value and             *
 3:   * call-by-reference parameter passing between C and      *
 4:   * assembly language modules. The min_max function is     *
 5:   * written in assembly language (in file minmax_a.asm).   *
 6:   *********************************************************/
 7:  #include <stdio.h>
 8:  int main(void)
 9:  {
10:        int     value1, value2, value3;
11:        int     minimum, maximum;
12:        extern  void min_max (int, int, int, int*, int*);
13:
14:        printf("Enter number 1 = ");
15:        scanf("%d", &value1);
16:        printf("Enter number 2 = ");
17:        scanf("%d", &value2);
18:        printf("Enter number 3 = ");
19:        scanf("%d", &value3);
20:
21:        min_max(value1, value2, value3, &minimum, &maximum);
22:        printf("Minimum = %d, Maximum = %d\n", minimum, maximum);
23:        return 0;
24:  }
```

Program 13.4 An example with the C program passing pointers to the assembly program: Assembly code (in file `minmax_a.asm`)

```
 1:  ;-------------------------------------------------------------
 2:  ; Assembly program for the min_max function - called from
 3:  ; the C program in file minmax_c.c. This function finds the
 4:  ; minimum and maximum of the three integers received by it.
 5:  ;-------------------------------------------------------------
 6:  .MODEL SMALL
 7:  .CODE
 8:  .486
 9:  PUBLIC _min_max
10:  _min_max   PROC
11:          enter   0,0
12:          ; AX keeps minimum number and DX maximum
13:          mov     AX,[BP+4]    ; get value 1
14:          mov     DX,[BP+6]    ; get value 2
15:          cmp     AX,DX        ; value 1 < value 2?
16:          jl      skip1        ; if so, do nothing
17:          xchg    AX,DX        ; else, exchange
18:  skip1:
19:          mov     CX,[BP+8]    ; get value 3
20:          cmp     CX,AX        ; value 3 < min in AX?
21:          jl      new_min
22:          cmp     CX,DX        ; value 3 < max in DX?
23:          jl      store_result
24:          mov     DX,CX
25:          jmp     store_result
26:  new_min:
27:          mov     AX,CX
28:  store_result:
29:          mov     BX,[BP+10]   ; BX := &minimum
30:          mov     [BX],AX
31:          mov     BX,[BP+12]   ; BX := &maximum
32:          mov     [BX],DX
33:          leave
34:          ret
35:  _min_max   ENDP
36:          END
```

Example 13.3 *String processing example.*
This example illustrates how global variables, declared in C, are accessed by assembly procedures. The string variable is declared as a global variable in the C program, as shown in Program 13.5 (line 9). The assembly language procedure computes the string length by accessing the global string variable, as shown in Program 13.6. The procedure call is parameterless in this example (see line 16 of the C program). The string variable is declared as an external variable in the assembly code (line 7) with an underscore, as it is an external variable.

Program 13.5 A string processing example: C code (in file `string_c.c`)

```
 1:  /*********************************************************
 2:   * A string processing example. Demonstrates processing   *
 3:   * global variables. Calls the string_length              *
 4:   * assembly language program in file string_a.asm file.   *
 5:   *********************************************************/
 6:  #include <stdio.h>
 7:  #define LENGTH 256
 8:
 9:  char string[LENGTH];
10:  int main(void)
11:  {
12:   extern int string_length (void);
13:
14:   printf("Enter string: ");
15:   scanf("%s", string);
16:   printf("string length = %d\n", string_length());
17:   return 0;
18:  }
```

Program 13.6 A string processing example: Assembly code (in file `string_a.asm`)

```
 1:  ;-------------------------------------------------------
 2:  ; String length function works on the global string
 3:  ; (defined in the C function). It returns string length.
 4:  ;-------------------------------------------------------
 5:  .MODEL SMALL
 6:  .DATA
 7:          EXTRN   _string:byte
 8:  .CODE
 9:  PUBLIC  _string_length
10:  _string_length  PROC
```

```
11:          mov     AX,0              ; AX keeps the character count
12:          mov     BX,OFFSET _string ; load BX with string address
13:  repeat:
14:          cmp     BYTE PTR[BX],0    ; compare with NULL character
15:          jz      done
16:          inc     AX                ; increment string length
17:          inc     BX                ; inc. BX to point to next char.
18:          jmp     repeat
19:  done:
20:          ret
21:  _string_length  ENDP
22:          END
```

13.4 Calling C Functions from Assembly

So far, we have considered how a C function can call an assembler procedure. Sometimes it is desirable to call a C function from an assembler procedure. This scenario often arises when we want to avoid writing assembly code for performing complex tasks. Instead, a C function could be written for those tasks. This section illustrates how we can access C functions from assembly procedures. Essentially, the mechanism is the same: we use the stack as the communication medium, as shown in the next example.

Example 13.4 *An example to illustrate a C function call from an assembly procedure.*
The main C function requests a set of marks of a class and passes this array to the assembly procedure `stats`, as shown in Program 13.7. The `stats` procedure computes the minimum, maximum, and rounded average marks and returns these three values to the C main function (see Program 13.8). To compute the rounded average mark, the C function `find_avg` is called from the assembly procedure. The required arguments `total` and `size` are pushed onto the stack (lines 42 and 43) before calling the C function on line 44. Since the convention for C calls for the caller to clear the stack, line 45 adds 4 to SP to clear the two arguments passed onto the `find_avg` C function. The rounded average integer is returned in the AX register.

Program 13.7 An example to illustrate C calls from assembly programs: C code (in file `marks_c.c`)

```
1:  /*******************************************************
2:   * An example to illustrate C program calling assembly    *
3:   * procedure and assembly procedure calling a C function. *
4:   * This program calls the assembly language procedure     *
5:   * in file MARKS_A.ASM. The program outputs minimum,      *
6:   * maximum, and rounded average of a set of marks.        *
7:   *******************************************************/
8:  #include <stdio.h>
9:
```

```
10:  #define  CLASS_SIZE  50
11:
12:  int main(void)
13:  {
14:        int     marks[CLASS_SIZE];
15:        int     minimum, maximum, average;
16:        int     class_size, i;
17:        int     find_avg(int, int);
18:        extern  void stats(int*, int, int*, int*, int*);
19:
20:        printf("Please enter class size (<50): ");
21:        scanf("%d", &class_size);
22:        printf("Please enter marks:\n");
23:        for (i=0; i<class_size; i++)
24:            scanf("%d", &marks[i]);
25:
26:        stats(marks, class_size, &minimum, &maximum, &average);
27:        printf("Minimum = %d, Maximum = %d, Average = %d\n",
28:                           minimum, maximum, average);
29:        return 0;
30:  }
31:   /*******************************************************
32:    * Returns the rounded average required by the assembly
33:    * procedure STATS in file MARKS_A.ASM.
34:    *******************************************************/
35:  int find_avg(int total, int number)
36:  {
37:        return((int)((double)total/number + 0.5));
38:  }
```

Program 13.8 An example to illustrate C calls from assembly programs: Assembly code (in file `marks_a.asm`)

```
 1:  ;-------------------------------------------------------
 2:  ; Assembly program example to show call to a C function.
 3:  ; This procedure receives a marks array and class size
 4:  ; and returns minimum, maximum, and rounded average marks.
 5:  ;-------------------------------------------------------
 6:  .MODEL SMALL
 7:  EXTRN   _find_avg:PROC
 8:  .CODE
 9:  .486
10:  PUBLIC _stats
```

```
11:  _stats  PROC
12:          enter   0,0
13:          push    SI
14:          push    DI
15:          ; AX keeps minimum number and DX maximum
16:          ; Marks total is maintained in SI
17:          mov     BX,[BP+4]     ; BX := marks array address
18:          mov     AX,[BX]       ; min := first element
19:          mov     DX,AX         ; max := first element
20:          xor     SI,SI         ; total := 0
21:          mov     CX,[BP+6]     ; CX := class size
22:  repeat1:
23:          mov     DI,[BX]       ; DI := current mark
24:          ; compare and update minimum
25:          cmp     DI,AX
26:          ja      skip1
27:          mov     AX,DI
28:  skip1:
29:          ; compare and update maximum
30:          cmp     DI,DX
31:          jb      skip2
32:          mov     DX,DI
33:  skip2:
34:          add     SI,DI         ; update marks total
35:          add     BX,2
36:          loop    repeat1
37:          mov     BX,[BP+8]     ; return minimum
38:          mov     [BX],AX
39:          mov     BX,[BP+10]    ; return maximum
40:          mov     [BX],DX
41:          ; now call find_avg C function to compute average
42:          push    WORD PTR[BP+6] ; push class size
43:          push    SI            ; push total marks
44:          call    _find_avg     ; returns average in AX
45:          add     SP,4          ; clear stack
46:          mov     BX,[BP+12]    ; return average
47:          mov     [BX],AX
48:          pop     DI
49:          pop     SI
50:          leave
51:          ret
52:  _stats  ENDP
53:          END
```

13.5 Inline Assembly Code

In the inline assembly method, assembly language statements are embedded into the C code. Such assembly statements are identified by placing the `asm` keyword before the assembly language instruction. The end of an inline `asm` statement is indicated by either a semicolon (;) or a newline. Multiple assembly language instructions can be written on the same `asm` line provided they are separated by semicolons, as shown below:

```
asm xor    AX,AX;  mov    AL,DH
```

If there are multiple assembly language instructions, we can also use braces to compound them, as shown below:

```
asm {
      xor    AX,AX
      mov    AL,DH
    }
```

Make sure to place the first brace on the same line as the `asm` keyword. To include comments on `asm` statements, we use C-style comments. We cannot use assembly language-style comments that start with a semicolon.

13.5.1 Compiling Inline Assembly Programs

Borland C++ can handle inline assembly code in one of two ways:

- Convert the C code first into assembly language and then invoke TASM to produce the object (`.obj`) file. We call this the TASM method.
- Use the built-in assembler (BASM) to assemble the `asm` statements in the C code. We call this the BASM method.

The compiling process using these two methods is shown in Figure 13.3. The BASM approach is restricted in the sense that only 16-bit instructions can be used. If we use 32-bit Pentium instructions, the Borland C++ compiler generates an error message. Then we can either simplify the inline code to avoid the instructions that BASM will not accept, or use the other method that invokes TASM. Using the BASM method does not require any special attention to compile an inline program.

In the TASM method, we can use the `-B` compiler option so that the compiler first generates an assembly language file and then invokes TASM to assemble it into the `.obj` file. Alternatively, we can include

```
#pragma   inline
```

at the beginning of the C file to instruct the compiler to use the `-B` option. The TASM method has the advantage of utilizing the capability of TASM, and hence we are not restricted to a subset of the assembly language instructions as in the BASM method.

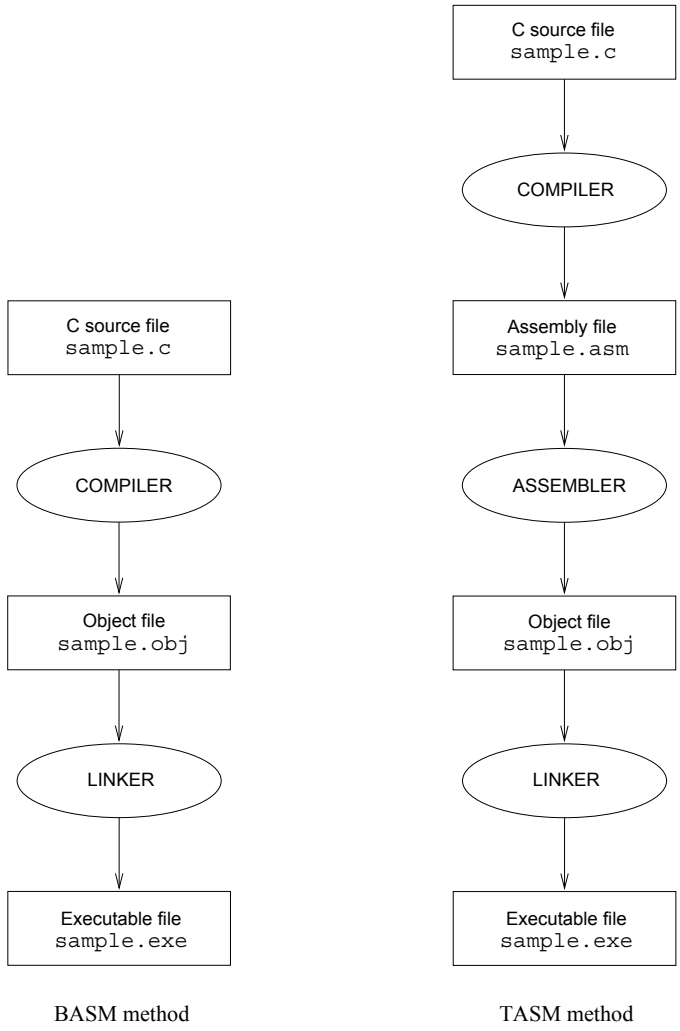


Figure 13.3 Steps involved in compiling mixed-mode programs with inline assembly code.

13.6 Summary

We introduced the principles involved in mixed-mode programming. We discussed the main motivation for writing mixed-mode programs. This chapter focused on mixed-mode programming involving C and the assembly language. Using the Borland C++ compiler and Turbo Assembler software, we demonstrated how assembly language procedures are called from C, and vice versa. Once you understand the principles discussed in this chapter, you can easily handle any type of mixed-mode programming activity.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- `#pragma` directive
- Inline assembly
- Left-pusher language
- Mixed-mode programs
- Parameter passing
- Right-pusher language

13.7 Exercises

13–1 Why do we need to write mixed-mode programs?

13–2 Find out details about how you can compile mixed-mode programs with your compiler (if it is other than the Borland C++ compiler).

13–3 Describe how parameters are passed from a C calling function to an assembly language procedure.

13–4 For your compiler, describe how 8-, 16-, and 32-bit values are returned to a C function.

13–5 For your compiler, which registers should be preserved by an assembly procedure?

13–6 What is the difference between right-pusher and left-pusher languages (as far as parameter passing is concerned)?

13–7 Explain why, in C, the calling function is responsible for clearing the stack.

13–8 What are the pros and cons of inline assembly as opposed to separate assembly modules?

13–9 What are the significant differences between the BASM and TASM methods for writing inline assembly code?

13.8 Programming Exercises

13–P1 Write a program that requests a string and a substring from the user and reports the location of the first occurrence of the substring in the string. Write a C main program to receive the two strings from the user. Have the C main program then call an assembly language procedure to find the location of the substring. This procedure should receive two pointers to strings `string` and `substring` and search for `substring` in `string`. If a match is found, it returns the starting position of the first match. Matching should be case sensitive. A negative value should be returned if no match is found. For example, if

`string` = Good things come in small packages.

and

`substring` = in

the procedure should return 8, indicating a match of `in` in `things`.

13–P2 Write a program to read a matrix (maximum size 10×10) from the user, and display the transpose of the matrix. To obtain the transpose of matrix **A**, write rows of **A** as columns. Here is an example:
If the input matrix is

$$\begin{bmatrix} 12 & 34 & 56 & 78 \\ 23 & 45 & 67 & 89 \\ 34 & 56 & 78 & 90 \\ 45 & 67 & 89 & 10 \end{bmatrix},$$

the transpose of the matrix is

$$\begin{bmatrix} 12 & 23 & 34 & 45 \\ 34 & 45 & 56 & 67 \\ 56 & 67 & 78 & 89 \\ 78 & 89 & 90 & 10 \end{bmatrix}.$$

The C part of your program is responsible for getting the matrix and for displaying the result. The transpose should be done by an assembly procedure. Devise an appropriate interface between the two procedures.

13–P3 Write a mixed-mode program that reads a string of characters as input and displays the number of alphabetic characters (i.e., A to Z and a to z) and number of digit characters (i.e., 0 to 9). The C main function prompts the user for a string and passes this string to an assembly procedure (say, `count`), along with two pointers for the two counts to be returned. The assembly procedure `count` calls the C library functions `isalpha` and `isdigit` to determine if a character is an alpha or digit character, respectively.

13–P4 We know that

$$1 + 2 + 3 + \cdots + N = \frac{N \times (N + 1)}{2}.$$

Write a program that requests N as input and computes the left- and the right-hand sides of the equation, verifies that they are equal, and displays the value. Organize your program as follows. The C main function should request the N value and also display the output. It should call an assembly procedure that verifies the equation and returns the value to the C main function. The assembly program computes the left-hand side and calls a C function to compute the right-hand side (it passes the N value to the C function). If the left-hand side is equal to the right-hand side, the assembly procedure returns the result of the calculation. Otherwise, a negative value is returned to the main C function.

Chapter 14

RISC Processors

Objectives

- To discuss the motivation for RISC processors;
- To present RISC design philosophy;
- To give details on PowerPC and Itanium processors;
- To discuss advanced processor features such as speculative execution and predication;
- To look at branch handling strategies.

As our discussion of the Pentium processor architecture and its instruction set indicates, the Pentium supports several complex instructions and a variety of addressing modes. A large number of addressing modes are provided to efficiently support manipulation of complex data structures such as multidimensional arrays and structures. Such processors are referred to as CISCs (complex instruction set computers). RISC (reduced instruction set computer) processors, on the other hand, use simple instructions and addressing modes. We start the chapter with an introduction to RISC processors. Section 14.2 describes the historical reasons for designing CISC processors. In this section, we also identify the reasons for the popularity of RISC designs. The next section discusses the principal characteristics of RISC processors, which include simple instructions and few addressing modes. RISC processors use the load/store architecture in which only the load and store instructions access memory. All other instructions get their operands from registers and write their results into registers.

We discuss two commercial processors that follow the RISC design philosophy. Section 14.4 gives details on the PowerPC processor, and the Intel 64-bit Itanium processor is discussed in Section 14.5. The Itanium has several interesting features such as speculative execution and predication. These features are discussed in detail. We conclude the chapter with a summary.

14.1 Introduction

The Pentium processor, discussed in the previous chapters, belongs to what is known as the complex instruction set computer (CISC) design. The obvious reason for this classification is the "complex" nature of its instruction set architecture (ISA). We define in the next couple of sections what we mean by complex ISA. For now, it is sufficient to know that the Pentium ISA provides a lot of support for higher-level languages (HLLs) at the expense of increased instruction set complexity. Two examples of complexity are the large number of addressing modes provided and wide range of operations—from simple to complex—supported. The motivation for designing such a complex instruction set is to provide an instruction set that closely supports the operations and data structures used by HLLs. However, the side effects of this design effort are far too serious to ignore.

The decision of CISC processor designers to provide a variety of addressing modes leads to variable-length instructions. For example, instruction length increases if an operand is in memory as opposed to in a register. This is because we have to specify the memory address as part of instruction encoding, which takes many more bits. As we show later, this complicates instruction decoding and scheduling. The side effect of providing a wide range of instruction types is that the number of clocks required to execute instructions also varies widely. This again leads to problems in instruction scheduling and pipelining.

For these and other reasons, in the early 1980s designers started looking at simple ISAs. Since these ISAs tend to produce instruction sets with far fewer instructions, they coined the term reduced instruction set computer (RISC). Even though the main goal was not to reduce the number of instructions, but the complexity, the term has stuck.

There is no precise definition of what constitutes a RISC system. However, we can identify certain characteristics that are present in most RISC systems. We identify these RISC design principles after looking at why the designers took the route of CISC in the first place. Since CISC and RISC have their advantages and disadvantages, modern processors take features from both classes. For example, the PowerPC, which follows the RISC philosophy, has quite a few complex instructions.

We look at two RISC processors in this chapter: the PowerPC and the Intel IA-64 Itanium processors. The next chapter describes another RISC processor—the MIPS processor—in detail, including its assembly language programming.

14.2 Evolution of CISC Processors

The evolution of CISC designs can be attributed to the desire of early designers to efficiently use two of the most expensive resources, memory and processor, in a computer system. In the early days of computing, memory was very expensive and small in capacity. Even in the mid-1970s, the cost of 16 KB RAM was about $500. This forced the designers to devise high-density code: that is, each instruction should do more work so that the total program size can be reduced. Since instructions are implemented in hardware, this goal could not be achieved until the late 1950s due to implementation complexity. In 1953, Wilkes and Stinger proposed microprogramming to deal with the complexity of implementing complex instructions [40].

Table 14.1 Characteristics of some CISC and RISC processors

	CISC		RISC
Characteristic	VAX 11/780	Intel 486	MIPS R4000
Number of instructions	303	235	94
Addressing modes	22	11	1
Instructions size (bytes)	2–57	1–12	4
Number of general-purpose registers	16	8	32

The introduction of microprogramming facilitated cost-effective implementation of complex instructions by using microcode. Microprogramming has not only aided in implementing complex instructions, it has also provided some additional advantages. Since microprogrammed control units use small fast memories to hold the microcode, the impact of memory access latency on performance could be reduced. Microprogramming also facilitates development of low-cost members of a processor family by simply changing the microcode.

Another advantage of implementing complex instructions in microcode is that the instructions can be tailored to high-level language constructs such as `while` loops. For example, the Pentium's `loop` instruction can be used to implement `for` loops. Similarly, memory block copying can be implemented by its string instructions. Thus, by using these complex instructions, we are closing the "semantic gap" that exists between HLLs and machine languages.

So far, we have concentrated on the memory resource. In the early days, effective processor utilization was also important. High code density also helps improve execution efficiency. As an example, consider the Pentium's string instructions, which autoincrement the index registers. Each string instruction typically requires two instructions on a RISC processor. As another example, consider the VAX-11/780, the ultimate CISC processor. It was introduced in 1978 and supported 22 addressing modes as opposed to 11 on the Intel 486 that was introduced more than a decade later. The VAX instruction size can range from 2 to 57 bytes, as shown in Table 14.1.

To illustrate how code density affects execution efficiency, consider the autoincrement addressing mode of the VAX processor. In this addressing mode, a single instruction can read data from memory, add contents of a register to it, write back the result to the memory, and increment the memory pointer. Actions of this instruction are summarized below:

```
(R2) = (R2)+ R3; R2 = R2+1
```

In this example, the R2 register holds the memory pointer. To implement this CISC instruction, we need four RISC instructions:

```
R4 = (R2)          ; load memory contents
R4 = R4+R3         ; add contents of R3
(R2) = R4          ; store the result
R2 = R2+1          ; increment memory address
```

The CISC instruction, in general, executes faster than the four RISC instructions. That, of course, was the reason for designing complex instructions in the first place. However, execution of a *single* instruction is not the only measure of performance. In fact, we should consider the overall system performance. We explore this topic further in the following pages.

Why RISC?

Designers make choices based on the available technology. As the technology—both hardware and software—evolves, design choices also evolve. Furthermore, as we get more experience in designing processors, we can design better systems. The RISC proposal is a response to the changing technology and the accumulation of knowledge from the CISC designs. CISC processors were designed to simplify compilers and to improve performance under constraints such as small and slow memories. The rest of the section identifies some of the important observations that motivated designers to consider alternatives to CISC designs.

Simple Instructions

The designers of CISC architectures anticipated extensive use of complex instructions because they close the semantic gap. In reality, it turns out that compilers mostly ignore these instructions. Several empirical studies have shown that this is the case. One reason for this is that different high-level languages use different semantics. For example, the semantics of the C `for` loop is not exactly the same as that in Pascal. Thus, compilers tend to synthesize the code using simpler instructions.

Few Data Types

CISC ISA tends to support a variety of data structures from simple data types such as integers and characters to complex data structures such as records and structures. Empirical data suggest that complex data structures are used relatively infrequently. Thus, it is beneficial to design a system that supports a few simple data types efficiently and from which the missing complex data types can be synthesized.

Simple Addressing Modes

CISC designs provide a large number of addressing modes. The main motivations are (i) to support complex data structures and (ii) to provide flexibility to access operands. For example, the Pentium provides "based-indexed addressing with scale-factor" to access complex data structures such as multidimensional arrays (see Chapter 11 for details). The Pentium also allows one of the source operands to be in the memory or register. Although this allows flexibility, it also introduces problems. First, it causes variable instruction execution times, depending on the location of the operands. Second, it leads to variable-length instructions. For example, on the Pentium, instruction length can range from 1 to 12 bytes. Variable instruction lengths lead to inefficient instruction decoding and scheduling.

Large Register Set

Several researchers have studied the characteristics of procedure calls in HLLs. We quote two studies—one by Patterson and Sequin [29] and the other by Tanenbaum [35]—in this section. Several other studies, in fact, support the findings of these two studies.

Patterson and Sequin's study of C and Pascal programs found that procedure call/return constitutes about 12 to 15% of HLL statements. As a percentage of the total machine language instructions, call/return instructions are about 31 to 33%. More interesting is the fact that call/return generates nearly half (about 45%) of all memory references. This is understandable as procedure call/return instructions use memory to store activation records. An activation record consists of parameters, local variables, and return values (see our discussion on page 421). In the Pentium, for example, the stack is extensively used for these activities. This explains why procedure call/return activities account for a large number of memory references. Thus, it is worth providing efficient support for procedure calls and returns.

In another study, Tanenbaum [35] found that only 1.25% of the called procedures had more than six arguments. Furthermore, more than 93% of them had less then six local scalar variables. These figures, supported by other studies, suggest that the activation record is not large. If we provide a large register set, we can avoid memory references for most procedure calls and returns. In this context, we note that the Pentium's eight general-purpose registers are a limiting factor in providing such support. The Itanium, which is described later in this chapter, provides a large register set (128 registers), and most procedure calls on the Itanium can completely avoid accessing memory.

14.3 RISC Design Principles

The best way to understand RISC is to treat it as a concept to design processors. Although initial RISC processors had fewer instructions compared to their CISC counterparts, the new generation of RISC processors has hundreds of instructions, some of which are as complex as the CISC instructions. It could be argued that such systems are really hybrids of CISC and RISC. In any case, there are certain principles that most RISC designs follow. We identify the important ones in this section. Note that some of these characteristics are intertwined. For example, designing an instruction set in which each instruction takes only one clock cycle to execute demands register-based operands, which in turn suggests that we need a large number of registers.

14.3.1 Simple Operations

The objective is to design simple instructions so that each can execute in one cycle. This property simplifies processor design. Note that a cycle is defined as the time required to fetch two operands from registers, perform an ALU operation, and store the result in a register. The advantage of simple instructions is that there is no need for microcode and operations can be hardwired. In terms of efficiency, these instructions should execute with the same efficiency as microinstructions of a CISC machine. If we design the cache subsystem properly to cap-

ture these instructions, the overall execution efficiency can be as good as a microcoded CISC machine.

Complex operations such as multiply and divide should be interpreted. For example, the Itanium provides a multiply instruction but does not have a divide instruction. On the other hand, the PowerPC provides both multiply and divide instructions. This reinforces our contention that the basic features we are discussing here should be viewed as concepts rather than treating them as "must have" features in a RISC implementation.

14.3.2 Register-to-Register Operations

A typical CISC instruction set includes not only register-to-register operations, but also register-to-memory and memory-to-memory operations. The Pentium, for instance, allows register-to-register as well as register-to-memory operations; it does not allow memory-to-memory operations. The VAX 11/780, on the other hand, allows memory-to-memory operations as well.

RISC processors allow only special `load` and `store` operations to access memory. The rest of the operations work on a register-to-register basis. This feature simplifies instruction set design as it allows execution of instructions at a one-instruction-per-cycle rate. Restricting most instruction operands to registers also simplifies the control unit. The three RISC processors we discuss in this part of the book—PowerPC, Itanium, and MIPS—use this load/store architecture. The SPARC processor, another RISC processor described in Appendix H, also uses the load/store architecture. This architecture was the basis for the Cray vector computer systems.

14.3.3 Simple Addressing Modes

Simple addressing modes allow fast address computation of operands. Since RISC processors employ register-to-register instructions, most instructions use register-based addressing. Only the load and store instructions need a memory addressing mode. RISC processors provide very few addressing modes: often just one or two. They provide the basic register indirect addressing mode, often allowing a small displacement that is either relative or absolute.

The Itanium supports three register addressing modes to access memory. The address can be obtained directly from a register by adding the contents of two registers, or by adding a 9-bit constant to the contents of a register. The PowerPC and MIPS support simple register-indirect addressing modes.

14.3.4 Large Number of Registers

Since RISC processors use register-to-register operations, we need to have a large number of registers. A large register set can provide ample opportunities for the compiler to optimize their usage. Another advantage with a large register set is that we can minimize the overhead associated with procedure calls and returns.

To speed up procedure calls, we can use registers to store local variables as well as for passing arguments. Local variables are accessed only by the procedure in which they are declared. Local variables come into existence at the time of a procedure call and die when the procedure exits. General-purpose registers of a processor can be divided into register windows, with each

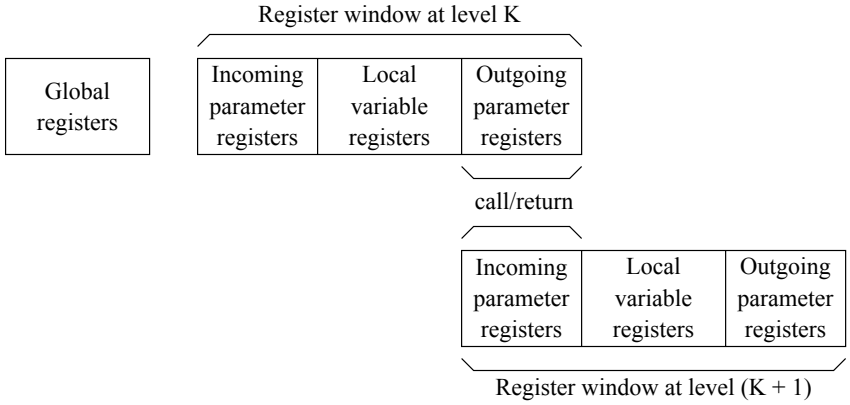


Figure 14.1 Register windows storing activation records can avoid memory access to speed up procedure call and return. The Intel Itanium, for instance, reserves the first 32 registers for global variables, and the remaining 96 registers can be used for local variables and parameters.

window allocated to a procedure invocation. The registers are organized as a circular buffer. To reduce overhead, outgoing parameter registers of a procedure can be overlapped with the incoming parameter registers of the called procedure as shown in Figure 14.1.

The circular buffer scheme avoids memory access for local variables. However, we also need to find a way to store global variables, which are accessed by more than one procedure. A simple solution is to store global variables in memory so that all procedures can access them. A better way is to reserve a set of general registers for global variables; this way, we can completely avoid accessing memory for procedure calls and returns. For example, the Itanium reserves the first 32 of the 128 registers for global variables. The remaining 96 registers can be used to store local variables and parameters. The Itanium also uses parameter register mapping similar to the scheme shown in Figure 14.1. More details on the Itanium processor are given in Section 14.5.

14.3.5 Fixed-Length, Simple Instruction Format

RISC processors use fixed-length instructions. Variable-length instructions can cause implementation and execution inefficiencies. For example, we may not know if there is another word that needs to be fetched until we decode the first word. Along with fixed-length instruction size, RISC processors also use a simple instruction format. The boundaries of various fields in an instruction such as opcode and source operands are fixed. This allows for efficient decoding and scheduling of instructions. For example, both PowerPC and MIPS processors use six bits for opcode specification.

Other Features

Most RISC implementations use the Harvard architecture, which allows independent paths for data and instructions. Harvard architecture thus doubles the memory bandwidth. However, processors typically use the Harvard architecture only at the CPU-cache interface. This requires two cache memories: one for data and the other for instructions.

RISC processors, like their CISC counterparts, use pipelining to speed up the instruction unit. Since RISC architectures use fixed-length instructions, the pipelines tend to have fewer stages than comparable CISC processors. Since RISC processors use simple instructions, there will be more instructions in a program than in their CISC counterparts. This increase in the number of instructions tends to increase dependencies, data as well as control dependencies. A unique feature of RISC instruction pipelines is that their implementation is visible at the architecture level. Due to this visibility, pipeline dependencies can be resolved by software, rather than in hardware. In addition, prefetching and speculative execution can also be employed easily due to features such as fixed-size instructions and simple addressing modes. We discuss these issues in detail when we describe the Itanium architecture.

14.4 PowerPC Processor

In this section, we present an overview of the PowerPC architecture and its instruction set. The Intel Itanium processor is discussed in the next section. Both the Itanium and PowerPC are more complex than the MIPS processor, which is discussed in detail in the next chapter. There is also an opportunity to use a simulator to run MIPS programs. The MIPS follows the RISC design principles much more closely than PowerPC and Itanium processors. After reading this chapter and the next, you will see a lot of commonality among these three RISC processors. You will also notice several significant differences among their principal characteristics.

Motorola, IBM, and Apple jointly developed the PowerPC architecture. IBM developed many of the concepts in 1975 for a prototype system. An unsuccessful commercial version of this prototype was introduced around 1986. Four years later, IBM introduced the RS/6000 family of processors based on these principles. The PowerPC architecture is based on IBM's POWER architectures implemented by the RS/6000 family of processors.

14.4.1 Architecture

The PowerPC is based on the load/store architecture and satisfies many of the RISC characteristics mentioned before. The PowerPC family of processors is available in both 32- and 64-bit implementations. Since we want to compare RISC processors with the Pentium, we discuss the 32-bit implementation in this section.

Registers

The PowerPC, being a RISC processor, provides a large number of general-purpose registers compared to the Pentium. The PowerPC has 32 general-purpose registers for integer data (GPR0 to GPR31). An equal number of registers are available for floating-point data (FPR0 to FPR31),

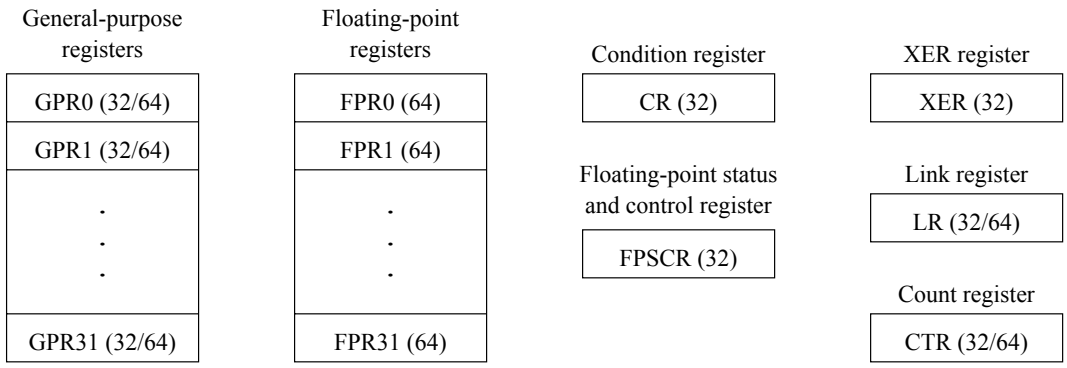


Figure 14.2 PowerPC registers: General-purpose, link, and count registers are 32-bits wide in 32-bit implementations and 64-bits wide in 64-bit implementations. The other registers are of fixed size independent of the implementation.

as shown in Figure 14.2. The integer registers (GPRs) are 32-bits wide, and the floating-point registers (FPRs) are 64-bits wide. In addition, we also look the following four special registers:

- *Condition Register (CR):* This is somewhat similar to the flags register in the Pentium. The 32-bit register is divided into eight CR fields of 4 bits each (CR0 to CR7). CR0 is used to capture the result of an integer instruction. The 4 bits in each CR field represent "less than" (LT), "greater than" (GT), "equal to" (EQ), and "overflow" (SO) conditions. CR1 is used to capture floating-point exceptions. The remaining CR fields can be used for either integer or floating-point instructions to capture integer or floating-point LT, GT, EQ, and SO conditions. Branch instructions are available to test a specific CR field bit. Instructions can specify the CR field that should be used to set the appropriate bit.
- *XER Register (XER):* The 32-bit XER register serves two distinct purposes. Bits 0, 1, and 2 are used to record summary overflow (SO), overflow (OV), and carry (CA). The OV bit is similar to the overflow flag bit in the Pentium. It records the fact that an overflow has occurred during the execution of an instruction. The SO bit is different in the sense that it is set whenever the OV bit is set. However, once set, the SO bit remains set until a special instruction is executed to clear it. The CA bit is similar to the carry bit in the Pentium. It is set by add and subtract arithmetic operations and shift right instructions.

  Bits 25 to 31 are used as a 7-bit byte count to specify the number of bytes to be transferred between memory and registers. This field is used by Load String Word Indexed (`lswx`) and Store String Word Indexed (`stswx`) instructions. Using just one `lswx` instruction we can load 128 contiguous bytes from memory into all 32 general-purpose registers. Similarly, reverse transfer can be done by `stswx` instruction.
- *Link Register (LR):* The 32-bit link register is used to store the return address in a procedure call. Procedure calls are implemented by branch (`bl`/`bla`) or conditional branch (`bc`/`bca`) instructions. For these instructions, the LR register receives the effective address of the instruction following the branch instruction. This is similar to the `call`

instruction in the Pentium, except for the fact that the Pentium places the return address on the stack.

- *Count Register (CTR):* The 32-bit CTR register holds the loop count value (similar to the ECX register in the Pentium). Branch instructions can specify a variety of conditions in many more ways than in the Pentium. For example, a conditional branch instruction can decrement CTR and branch only if CTR $\neq$ 0 or if CTR = 0. Even more complex branch conditions can be tested. More details on the branch instruction are presented on page 589.

Of these four registers, 64-bit implementations use 64-bit LR and CTR registers. The other two registers remain as 32-bit registers even in 64-bit implementations.

Like the Pentium, 32-bit implementations of the PowerPC use segmentation. As a result, a set of 16 segmentation registers are available, each 32-bits wide. The 64-bit implementations use a 64-bit address space register (ASR) instead. We have not shown these registers in Figure 14.2.

Addressing Modes

Load and store instructions support three addressing modes. We can specify three general-purpose registers `rA`, `rB`, and `rD/rS` in load/store instructions. Registers `rA` and `rB` are used to compute the effective address. The third register is treated as either the destination register `rD` for load operations or the source register `rS` for store operations. We discuss the instruction format in the next section.

- *Register Indirect Addressing:* This addressing mode uses the contents of the specified general-purpose register `rA` as the effective address. Interestingly, we can also specify `0` for `rA`, which generates the address 0.

  Effective address = Contents of `rA`.

- *Register Indirect with Immediate Index Addressing:* In this addressing mode, instructions contain a signed 16-bit immediate index value `imm16`. The effective address is computed by adding this value to the contents of a general-purpose register `rA` specified in the instruction. As in the last addressing mode, a `0` can be specified in place of `rA`. In this case, the effective address is the immediate value given in the instruction. Thus, it is straightforward to convert indirect addressing to direct addressing.

  Effective address = Contents of `rA` or `0` + `imm16`.

- *Register Indirect with Index Addressing:* Instructions using this addressing mode specify two general-purpose registers `rA` and `rB`. The effective address is computed as the sum of the contents of these two registers. As in the other addressing modes, we can specify `0` in place of `rA`.

  Effective address = Contents of `rA` or `0` + Contents of `rB`.

It is interesting to see how the addressing modes of the Pentium and PowerPC compare. The Pentium supports six memory addressing modes in 32-bit mode (see Figure 11.2 on page 437). The PowerPC also seems to support most of these addressing modes. The exceptions are that we cannot use a scale factor for the index and the based-indexed addressing mode cannot specify a displacement.

We show in the next chapter that MIPS processors support even fewer addressing modes.

14.4.2 PowerPC Instruction Set

Unlike the Pentium, the PowerPC uses a fixed-length instruction format that is characteristic of RISC processors. Only load and store instructions access data in memory.

Instruction Format

All PowerPC instructions are encoded into four bytes. Instructions are assumed to be word-aligned, so that the processor can ignore the least significant two bits of all instruction addresses.

As shown in Figure 14.3, bits 0 to 5 specify the primary opcode. Many instructions also have an extended opcode. The remaining bits are used for various fields depending on the instruction type. Here we discuss some basic instruction formats.

Most instructions use the register format shown in Figure 14.3a. In arithmetic, logical, and other similar instruction types, registers `rA` and `rB` specify the source operands and `rD` specifies the destination register. The `OE` and `rC` bits are explained on page 584.

The immediate format shown in Figure 14.3b is used by instructions that specify an immediate operand. For example, the `addi` instruction, described in the next section, uses this format.

The format shown in Figure 14.3c is used by branch instructions. The unconditional branch format allows specification of a 24-bit target address. This figure also shows the format used by direct and indirect conditional branch instructions. The `AA` bit is used to indicate whether the address is an absolute address or a relative address. The `LK` bit is used to link the return address so that we can use the branch instruction for procedure calls. More details on the branch instruction fields, including the `BO` and `BI` fields, are given on page 589.

The format of load/store instructions is shown in Figure 14.3d. The first format is used by load/store instructions that specify the address in the first two addressing modes on page 580. As mentioned before, these instructions use `rA` and the signed 16-bit operand (`imm16`) to compute the effective address. The second format is used by load/store instructions that use the index addressing mode (i.e., the third addressing mode on page 580).

We have skipped several other formats used by the PowerPC to encode instructions. The PowerPC manual [26] contains a more detailed description of the instruction formats.

Data Transfer Instructions

The PowerPC supports a variety of load and store instructions to move data between memory and registers. We discuss some of these instructions to convey the type of instructions available. As in the Pentium, load/store instructions do not affect any of the status bits.

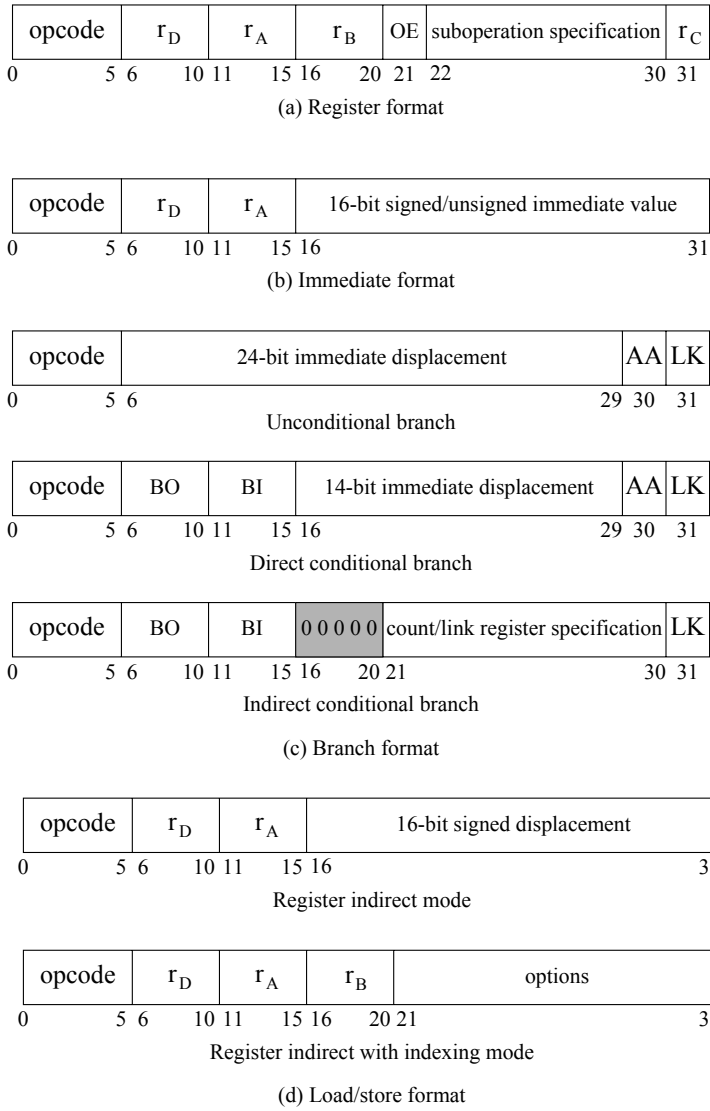


Figure 14.3 Sample PowerPC instruction formats.

Load instructions operate on byte, half-word, and word data. Note that load instructions that operate on floating-point numbers are also available, but we do not discuss them here. The following is a list of load instructions that work on byte data:

`lbz`	`rD,disp(rA)`	Load Byte and Zero
`lbzu`	`rD,disp(rA)`	Load Byte and Zero with Update

`lbzx`	`rD,rA,rB`	Load Byte and Zero Indexed
`lbzux`	`rD,rA,rB`	Load Byte and Zero with Update Indexed

The first instruction `lbz` loads the byte at the effective address (EA) into the lower-order byte of `rD`. The remaining three bytes in `rD` are cleared (i.e., zeroed). The EA is computed as the sum of the contents of `rA` and the displacement `disp`. Note that if we specify 0 for `rA`, `disp` is used as the effective address.

The second instruction `lbzu` performs the same operation as the `lbz` instruction. In addition, it loads the computed effective address into `rA` (i.e., it updates `rA` with EA).

The last two instructions use the indexed addressing mode. The effective address is computed as the sum of the contents of registers `rA` and `rB`. Except for the computation of the EA, `lbzx` is similar to the `lbz` instruction, and `lbzxu` is similar to the `lbzu` instruction.

To move halfwords, there are four instructions corresponding to the four-byte load instructions. Halfword instructions use the mnemonics `lhz`, `lhzu`, `lhzx`, and `lhzux`. Similarly, word instructions use the mnemonics `lwz`, `lwzu`, `lwzx`, and `lwzux`.

When loading halfwords, instead of zeroing the upper two bytes, we can also sign-extend halfword to word. Remember that sign extension copies the sign bit to the remaining higher order bits. We can use the following load instructions to sign extend the halfword in `rD`:

`lha`	`rD,disp(rA)`	Load Halfword Algebraic
`lhau`	`rD,disp(rA)`	Load Halfword Algebraic with Update
`lhax`	`rD,rA,rB`	Load Halfword Algebraic Indexed
`lhaux`	`rD,rA,rB`	Load Halfword Algebraic with Update Indexed

The PowerPC provides some interesting multiword load instructions. As an example, consider the following instruction:

`lmw`	`rD,disp(rA)`	Load Multiple Words

It loads n consecutive words from memory starting at EA, which is computed as in the previous instructions. Since the target is a register, what is n? This instruction loads words starting with `rD` and proceeds until `r31`. For example, if we specify `r20` as `rD`, 12 consecutive words from memory are loaded into registers `r20`, `r21`, ..., `r31`.

There is a store instruction corresponding to each load instruction to move data to memory. For example, to move a byte into memory, these instructions can be used:

`stb`	`rD,disp(rA)`	Store Byte
`stbu`	`rD,disp(rA)`	Store Byte with Update
`stbx`	`rD,rA,rB`	Store Byte Indexed
`stbux`	`rD,rA,rB`	Store Byte with Update Indexed

There are corresponding instructions to store halfwords: `sth`, `sthu`, `sthx`, and `sthux`. Similar instructions are available to store words. To store multiple words, we can use the following:

```
stmw      rD,disp(rA)      Store Multiple Words
```

This instruction has the same semantics as the `lmw` instruction except for the direction of data movement.

Arithmetic Instructions

The PowerPC supports the four basic arithmetic instructions: add, subtract, multiply, and divide. We start our discussion with the addition instructions.

Addition Instructions

The basic add instruction

```
add     rD,rA,rB
```

adds contents of registers `rA` and `rB` and stores the result in `rD`. Status and overflow bits of the `CR0` field as well as the `XER` register are not altered. Three variations on the basic add are possible to affect these bits:

`add.`	`rD,rA,rB`	LT, GT, EQ, SO bits of CR0 field are altered
`addo`	`rD,rA,rB`	SO and OV bits of XER register are altered
`addo.`	`rD,rA,rB`	LT, GT, EQ, SO bits of CR0 field and SO and OV bits of XER register are altered

The `OE` bit in Figure 14.3*a* indicates whether the SO and OV bits of the XER register are altered (`OE` = 1) or not (`OE` = 0). Thus, this bit is set for the last two add instructions (`addo` and `addo.`) and cleared for the other two instructions.

The `Rc` bit specifies if the LT, GT, EQ, SO bits of CR0 field should be altered. This bit is set for those instructions that alter these bits. Thus, `rc` = 1 for the two "dot" versions of the add instruction (`add.` and `addo.`).

These four add instructions do not affect the carry (CA) bit in the XER register. To alter the carry bit, we have to use `addc`, `addc.`, `addco`, and `addco.` instructions. The other bits are updated as in the basic `add` instruction variations. Yet another variation is the `adde` instruction. The instruction

```
adde    rD,rA,rB
```

adds contents of registers `rA`, `rB` and the CA bit of the XER register. As usual, the result goes into the `rD` register. Like the `add` instruction, it does not alter any of the status bits. We can use `adde.`, `addeo`, or `addeo.` to affect these status and condition bits.

We can also have an immediate operand specified in an add instruction. The add instruction

```
addi    rD,rA,Simm16
```

is similar to the `add` instruction except that the second operand is a signed 16-bit immediate value. It does not affect any status/condition bits. If `rA` is specified as 0, it uses the value 0,

not the contents of GPR0. Thus, we can use the `addi` instruction to implement load immediate (`li`), load address (`la`), and subtract immediate (`subi`) as shown below:

```
li     rD,value      equivalent to  addi   rD,0,value
la     rD,disp(rA)   equivalent to  addi   rD,rA,disp
subi   rD,rA,value   equivalent to  addi   rD,rA,-value
```

Since the processor does not directly support these instructions, we refer to these instructions as pseudoinstructions. The assembler translates these pseudoinstructions to equivalent processor instructions. As we show in the next chapter, the MIPS also uses pseudoinstructions to simplify assembly language programming.

Subtract Instructions

Subtract instructions use the mnemonic `subf` standing for "subtract from." The subtract instruction

```
subf     rD,rA,rB          /* rD = rB−rA */
```

subtracts the contents of `rA` from `rB` and places the result in the `rD` register. As with the `add` instruction, no status/condition bits are affected. We can also use the simplified mnemonic `sub` for this instruction. Other subtract instructions—`subf.`, `subfo`, and `subfo.`—are available to alter these bits as in the `add` instruction.

To alter the carry (CA) bit, use `subfc` instead of the `subf` mnemonic. Similarly, to include carry, use the `subfe` mnemonic. There is no "subtract immediate" instruction as it can be implemented using the `addi` instruction, as explained before.

As does the Pentium, the PowerPC also provides the negate instruction. The negate instruction

```
neg      rD,rA             /* rD = 0−rA */
```

essentially negates the sign of the integer in the `rA` register. The processor actually performs a 2's complement operation (i.e., complements the bits of `rA` and adds 1).

Multiply Instructions

The PowerPC multiply instruction is slightly different from that of the Pentium in that it does not produce the full 64-bit result. Remember that we get a 64-bit result when multiplying two 32-bit integers. The PowerPC provides two instructions to get the lower- and higher-order 32 bits of the 64-bit result. First, we look at the signed integers. The instruction

```
mullw      rD,rA,rB
```

multiplies the contents of registers `rA` and `rB` and stores the lower-order 32 bits of the result in `rD` register. We have to use

```
mulhw     rD,rA,rB
```

to get the higher-order 32 bits of the result.

For unsigned numbers, we have to use `mulhwu` instead of `mulhw` to get the higher-order 32 bits of the result. The lower-order 32 bits are given by `mullw` for both signed and unsigned integers.

There is also a multiply immediate instruction that takes an immediate value. The format is

```
mulli     rD,rA,Simm16
```

The immediate value `Simm16` is a 16-bit signed integer. Note that this operation produces a 48-bit result. But the `mulli` instruction stores only the lower 32 bits of the result.

Divide Instructions

As with the Pentium, two divide instructions—one for signed integers and the other for unsigned integers—are provided. The instruction

```
divw     rD,rA,rB          /* rD  = rA/rB */
```

stores the quotient of `rA`/`rB` in `rD`. The operands are treated as signed integers. The remainder is not available. To get both quotient and remainder, we have to use the following three-instruction sequence:

```
divw      rD,rA,rB         /* quotient in rD */
mullw     rX,rD,rB
subf      rC,rX,rA         /* remainder in rC */
```

For unsigned integers, use the `divwu` (divide word unsigned) instruction instead.

Logical Instructions

The PowerPC instruction set has several logical instructions including `and`, `or`, `nor`, `nand`, `equ`, and `xor` instructions. It does not have the `not` instruction; however, the `not` operation can be implemented using the `nor` instruction.

We give the complete set of `and` instructions provided by the PowerPC:

```
and       rA,rS,rB             and.       rA,rS,rB
andi.     rA,rS,Uimm16         andis.     rA,rS,Uimm16
andc      rA,rS,rB             andc.      rA,rS,rB
```

The `and` instruction performs bit-wise AND of `rS` and `rB` and places the result in the `rA` register. The condition register field `CR0` is not affected. The `and.` instruction is similar to `and` but updates the LT, GT, EQ, and SO bits of the `CR0` field. This is true with all the instructions ending in a period. The `andi` takes a 16-bit unsigned integer as one of the source operands. The `andis.` instruction is similar to `andi.` except that the immediate value is

left-shifted by four bit positions before ANDing. In the `andc` instruction, the contents of `rB` are complemented before performing the AND operation.

The logical `or` instruction also has six versions like the `and` instruction. The only difference is that the immediate versions do not update the `CR0` field. That is, the immediate `or` instructions are `ori` and `oris` with no period. We can use the `or` instruction to move register contents as shown below:

```
mr      rA,rS      is equivalent to      or      rA,rS,rS
```

A no-operation (`nop`) is implemented as

```
ori     0,0,0
```

The NAND instructions (`nand` and `nand.`) perform a NOT operation followed by an AND operation. Similarly, NOR instructions (`nor` and `nor.`) perform a NOT operation after an OR operation.

The PowerPC instruction set includes four `xor` instructions as shown below:

```
xor      rA,rS,rB            xor.      rA,rS,rB
xori     rA,rS,Uimm16        xoris     rA,rS,Uimm16
```

The semantics of these four instructions is similar to the `and` instructions, except for the actual operation performed.

In addition, the PowerPC provides the `eqv` (equivalence) logical function. The *equivalence* function is defined as the exclusive-NOR operation (i.e., the output of XOR is complemented; see page 44). Two instructions—`eqv` and `eqv.`— are available.

Shift and Rotate Instructions

The PowerPC provides four types of shift instructions to shift left or right. Each type of instruction is available in two forms: one that does not affect the four status bits in `CR0` and the other that updates these four bits (i.e., the "dot" version). We first discuss the left-shift instruction. The `slw` (shift left word) instruction

```
slw     rA,rS,rB
```

left-shifts the contents of `rS` by the shift count value specified in `rB`, and the result is placed in `rA`. Shifted bits on the right receive zeros. For the shift count, only the least significant five bits are taken from the `rB` register. We can use `slw.` to update the four status bits in `CR0`.

There are two shift right instructions—`srw` and `srw.`—that are similar to the two left-shift instructions except for the direction of shift. Zeros replace the vacated bits on the left. These two right-shift instructions perform logical shift operations. In logical shifts, vacated bits receive zeros. On the other hand, in arithmetic right-shift, vacated bits receive the sign bit. For details on the differences and the need for logical and arithmetic shift operations and why we need them only for the right-shifts, see our discussion in Section 9.6.5 on page 357.

The PowerPC provides two types of arithmetic right-shift operations: one type assumes that the shift count is in a register as in the previous shift instructions, and the other type can accept the shift count as an immediate value.

The register versions use the following format:

```
sraw     rA,rS,rB
sarw.    rA,rS,rB
```

The instructions

```
srawi     rA,rS,count
sarwi.    rA,rS,count
```

use the 5-bit immediate value `count` as the shift count. One difference between the arithmetic shift instructions and the other shift instructions is that these four arithmetic instructions affect the carry (CA) bit in the XER register.

The PowerPC provides several rotate left instructions. We describe only one of them to see an interesting feature of this instruction. The `rlwnm` (rotate left word then AND with mask) instruction takes five operands as shown below:

```
rlwnm     rA,rS,rB,MB,ME
```

The contents of `rS` are rotated left by the count value specified in the lower-order five bits of `rB`. A mask value that contains 1s from the `MB` bit to the `ME` bit and 0s in all the other bit positions is generated. The rotated value is ANDed with the mask value and the result is placed in `rA`. This instruction is useful to extract and rotate bit fields. It is straightforward to implement a simple rotate left instruction as shown below:

```
rotlw     rA,rS,rB     is equivalent to     rlwnm     rA,rS,rB,0,31
```

Comparison Instructions

We describe two compare instructions: for comparing signed numbers and unsigned numbers. Each of these instructions is available in two formats: register version and immediate version. We first look at the signed compare instructions. This instruction compares two numbers and updates the specified `CRx` field of the `CR` register. The format is

```
cmp     crfD,rA,rB
```

If the contents of `rA` are less than the contents of `rB`, the LT bit in the `crfD` is set: if greater, the GT bit is set; otherwise, the EQ bit is set. It also updates the SO field. The `CR0` field is used if no `CR` field is specified as shown below:

```
cmpd     rA,rB     is equivalent to     cmp     0,rA,rB
```

The immediate version of this statement

```
cmpi     crfD,rA,Simm16
```

takes an immediate 16-bit signed integer in place of `rB`. To treat the operands as unsigned integers, we can use `cmpl` (compare logical) and `cmpli` (compare logical immediate) instructions.

Branch Instructions

The PowerPC implements branch and procedure invocation operations using more flexible branch instructions than the Pentium's branch instructions. Branch instruction encodings are shown in Figure 14.3 on page 582.

As in the other instructions, the most significant 6 bits are used for op-code. The remaining 26 bits are used for specifying the target. Since all instructions take four bytes and are aligned, the least significant 2 bits are always zero. These 2 bits are used to make the branch instruction more flexible. The AA bit is used to indicate whether the address is the absolute address (AA = 1) or PC-relative address (AA = 0). In the absolute address mode, the 26-bit value is treated as the branch target address. In the PC-relative mode, this value is used as an offset relative to the contents of the program counter (PC). Thus, the PC-relative mode works as do the Pentium's jump instructions.

The second bit LK is used to convert the branch instruction into a procedure `call` instruction. When the LK bit is set, the return address (i.e., the address of the instruction following the branch) is placed in the link (LR) register.

There are four unconditional branch variations, depending on the values specified for the AA and LK bits:

`b`	`target`	(AA = 0, LK = 0)	Branch
`ba`	`target`	(AA = 1, LK = 0)	Branch Absolute
`bl`	`target`	(AA = 0, LK = 1)	Branch then Link
`bla`	`target`	(AA = 1, LK = 1)	Branch Absolute then Link

All instructions transfer control to the `target` address. The last two instructions, `bl` and `bla`, also load the LR register with the address of the instruction following the branch instruction. Thus, these instructions are similar to the Pentium's `call` instruction.

There are also three types of conditional branch instructions. The first type uses direct address as do the previous branch instructions. The remaining two types use register indirect branching. One uses the count register (CTR) to supply the target address, and the other uses the link register (LR) for this purpose. This last type of branch where the target is given by the link register is essentially used to return from a procedure.

Just as with the unconditional branch instructions, four versions are available:

`bc`	`BO,BI,target`	(AA = 0, LK = 0)	Branch Conditional
`bca`	`BO,BI,target`	(AA = 1, LK = 0)	Branch Conditional Absolute
`bcl`	`BO,BI,target`	(AA = 0, LK = 1)	Branch Conditional then Link
`bcla`	`BO,BI,target`	(AA = 1, LK = 1)	Branch Conditional Absolute then Link

The BO (branch options) operand, which is five bits long, specifies the condition under which the branch is taken. The BI (branch input) operand specifies the bit in the CR field that should be used as the branch condition.

We can specify the following nine different branch conditions:

- Decrement CTR; branch if CTR $\neq$ 0 and the condition is false.
- Decrement CTR; branch if CTR $=$ 0 and the condition is false.
- Decrement CTR; branch if CTR $\neq$ 0 and the condition is true.
- Decrement CTR; branch if CTR $=$ 0 and the condition is true.
- Branch if the condition is false.
- Branch if the condition is true.
- Decrement CTR; branch if CTR $\neq$ 0.
- Decrement CTR; branch if CTR $=$ 0.
- Branch always.

The link register-based branch instructions are shown below:

`bclr`	`BO,BI`	(LK = 0)	Branch Conditional to Link Register
`bclrl`	`BO,BI`	(LK = 1)	Branch Conditional to Link Register then Link

The BO and BI operands play the same role as in the previous direct conditional branch instructions. For these instructions, the target address is taken from the LR register. These instructions are used to return from a procedure call.

The final two branch instructions

`bcctr`	`BO,BI`	(LK = 0)	Branch Conditional to Count Register
`bcctrl`	`BO,BI`	(LK = 1)	Branch Conditional to Count Register then Link

use the CTR register instead of the LR register.

You can see from this description that the PowerPC does not closely adhere to all the principles of the RISC philosophy. Although it uses simple addressing modes and fixed-size instructions, some of its instructions are not simple. For example, the `lswx` and `stlwx` instructions mentioned on page 579 are not simple by any stretch of the imagination. For more details on the PowerPC processor, see the PowerPC manual [26]. Next we look at the architecture of the Intel Itanium processor.

14.5 Itanium Processor

Intel has moved from CISC designs of Pentium processors to RISC orientation for their 64-bit processors. Intel has finally decided to use the Itanium for their 64-bit processors. Like the PowerPC, the Itanium also uses the load/store architecture. Furthermore, the RISC features present in the PowerPC are also present. Because of these similarities, we briefly give the instruction set details of the Itanium.

In addition to the standard RISC features, the Itanium incorporates several advanced architectural features to improve performance. We discuss these features in this chapter as the Itanium provides a convenient platform. More specifically, we discuss instruction-level parallelism, register stacks, speculative execution, and predication to improve instruction reordering.

The Itanium's ISA is based on the EPIC (explicit parallel instruction computing) design philosophy. Of course, it also maintains backward compatibility to the IA-32 ISA. EPIC design features include the following:

- *Explicit Parallelism:* The ISA provides necessary support for the compiler to convey information on the instructions that can be executed in parallel. In traditional architectures, hardware extracts this instruction-level parallelism (ILP) within a fairly small window of instructions (or reorder buffer). In contrast, the Itanium allows the compiler to do the job. Since ILP extraction is done in software, a more detailed analysis can be done on a much larger window at compile time.

  The Itanium also provides hardware support to execute instructions in parallel by reading three instructions as a bundle. The compiler packs only instructions that have no dependencies into a bundle. Thus, the processor does not have to spend time in analyzing the instruction stream to extract ILP. We present details on this and other features of the Itanium later in this chapter.
- *Features to Enhance ILP:* Since the Itanium allows the compiler to detect and extract ILP, we can use more elaborate methods to improve ILP. Two such schemes are speculative execution and predication. Speculative execution allows high-latency load instructions to be advanced so that the latency can be masked. Branch handling is improved by using predication. In some instances, branch instructions can be completely eliminated. We present details on these two techniques later.
- *Resources for Parallel Execution:* It is imperative that to successfully exploit the ILP detected by the compiler, the processor should provide ample resources to execute instructions in parallel. The Itanium provides a large number of registers and functional units. It has 128 integer registers and another 128 floating-point registers. The large number of registers, for example, can be effectively used to make procedure calls and returns very efficient. Most procedure calls/returns need not access memory for parameter values and local and global variables.

To summarize, Itanium architecture improves performance by

- Increasing instruction-level parallelism by providing large register sets and a three-instruction wide word;
- Hiding memory latency by speculative loads;
- Improving branch handling by using predication; and
- Providing hardware support for efficient procedure calls and returns.

14.5.1 Architecture

The Itanium supports both integer and floating-point data types. Integer data types can be 1-, 2-, 4-, or 8-bytes wide. With a few exceptions, integer operations are done on 64-bit data. Furthermore, registers are always written as 64 bits. Thus, 1-, 2-, and 4-byte operands loaded from memory are zero-extended to 64 bits. Floating-point data types include IEEE single, double, and double-extended formats.

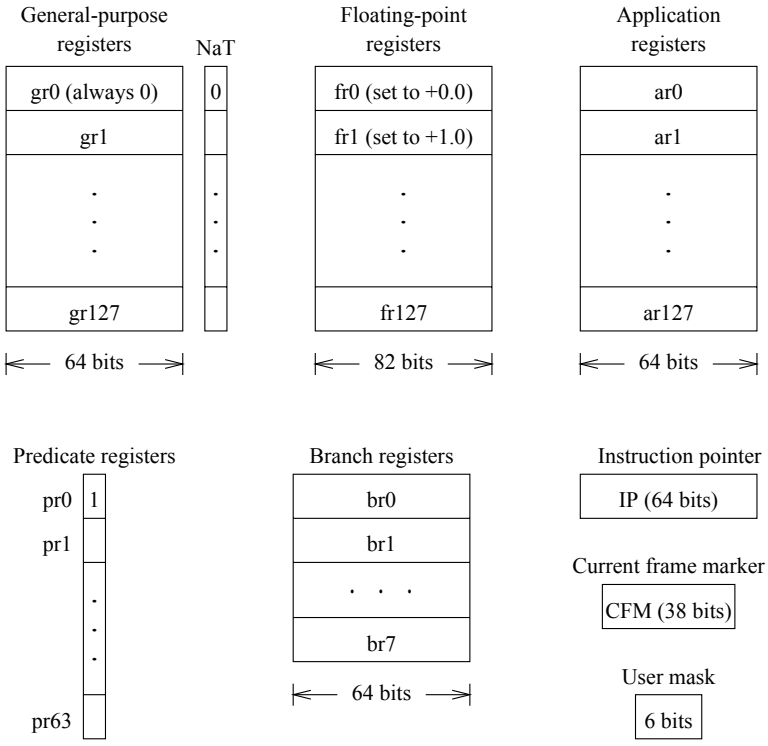


Figure 14.4 Itanium registers: The general-purpose register `gr0` and its `NaT` bit are hardwired to zero. Similarly, the first two floating registers are set to 0 and 1. The predicate register `pr0` is set to 1.

Registers

The Itanium has 128 general registers, which are labeled `gr0` through `gr127` as shown in Figure 14.4. Each register is 64-bits wide. In addition, a bit is associated with each register to indicate whether the register is free or contains something valid. This bit is called `NaT`, which stands for *Not-a-Thing*. We explain later how this bit is used in speculative loading.

The general registers are divided into *static* and *stacked* registers. Static registers are comprised of the first 32 registers: `gr0` through `gr31`. Of these, `gr0` is essentially a read-only register. This register always provides a zero value when used as a source operand. Writing to this register generates an "illegal operation" fault.

General registers `gr32` through `gr127` are classified as stacked registers. These registers are available for use by a program as a register stack frame.

The eight 64-bit branch registers, `br0` through `br7`, hold the target address for indirect branches. These registers are used to specify the target address for conditional branch, procedure call, and return instructions.

The user mask (UM) register is used to control memory access alignment, byte ordering (little-endian or big-endian), and user-configurable performance monitors. If the byte order bit is 0, little-endian order is used; otherwise, big-endian order is assumed. In IA-32 memory accesses, data access always uses little-endian order as does the Pentium and ignores this bit.

The uses of the remaining registers—predicate registers, applications registers, and current frame marker—are discussed later. The instruction pointer register plays the same role as the IP register in the Pentium.

Addressing Modes

The Itanium provides three addressing modes as does the PowerPC. As mentioned in Section 14.3, RISC processors provide only a few, simple addressing modes. Since the Itanium follows the load/store architecture, only the load and store instructions can access memory. These instructions use three general registers: `r1`, `r2`, and `r3`. Two of these registers, `r2` and `r3`, are used to compute the effective address. The third register `r1` either receives (in load) or supplies (in store) the data. Since these three addressing modes are very similar to the PowerPC addressing modes, we briefly describe them here:

- *Register Indirect Addressing:* In this mode, load and store instructions use the contents of `r3` as the effective address.

  Effective address = Contents of `r3`.

- *Register Indirect with Immediate Addressing:* In this addressing mode, the effective address is computed by adding the contents of `r3` and a signed 9-bit immediate value `imm9` specified in the instruction. The computed effective address is placed in `r3`.

  Effective address = Contents of `r3` + `imm9`,
  `r3` = Effective address.

- *Register Indirect with Index Addressing:* In this addressing mode, two general registers `r2` and `r3` are used to compute the effective address as in the PowerPC. As in the previous addressing mode, `r3` is updated with the computed effective address.

  Effective address = Contents of `r3` + Contents of `r2`,
  `r3` = Effective address.

These three addressing modes appear to be similar to the PowerPC modes; the update feature of the last two addressing modes gives more flexibility. For example, we can use the second addressing mode to access successive elements of an array by making `imm9` equal to the element size.

Procedure Calls

The Itanium provides hardware support for efficient procedure invocation and return. Unlike the Pentium procedure calls, an Itanium procedure call typically does not involve the stack. Instead, it uses a register stack for passing parameters, local variables, and the like. For this purpose, the 128 general register set is divided into two subsets:

- *Static Registers:* The first 32 registers, `gr0` though `gr31`, are called static registers. These registers are used to store the global variables and are accessible to all procedures.
- *Stacked Registers:* The upper 96 registers, `gr32` through `gr127`, are called stack registers. These registers, instead of the stack, are used for passing parameters, returning results, and storing local variables.

A *register stack frame* is the set of stacked registers visible to a given procedure. This stack frame is partitioned into local area and output area. The size of each area can be specified by each procedure by using the `alloc` instruction. The local area consists of input area and space for local variables of the procedure. The input area is used to receive parameters from the caller and the output area is used to pass parameters to the callee. When a procedure is called, the `alloc` instruction can be used to allocate the required number of registers for the local and output areas. As mentioned, local includes storage to receive arguments in addition to local variable storage. The Itanium aligns the caller's output area with the callee's local area so that passing of parameters does not involve actual copying of register values (see Figure 14.1 on page 577). Furthermore, the Itanium uses register renaming. That is, independent of the actual set of registers allocated to a procedure, the allocated register set is renamed such that the first register is always labeled as `gr32`.

The current frame marker (CFM) register maintains information on the current stack frame. It keeps two values: size of frame (SOF) and size of locals (SOL). As the names indicate, SOF gives the total stack frame size, and SOL specifies the local area size. The difference (SOF − SOL) is the output area size. The `alloc` statement takes three immediate values that specify the size of inputs, locals, and outputs. The SOF value is determined by adding these three values; SOL is given by the sum of the first two values (i.e., size of inputs and locals).

Another interesting feature is that a procedure's stack frame can be up to 90 registers. What happens if some of the registers are allocated to other procedures? Itanium uses a hardware mechanism called a register stack engine (RSE) to transparently manage registers. When allocation exceeds the available registers on the stack, it moves data from the registers to memory. The stored registers are restored when returning from the procedure. This mechanism is transparent, but the cost is not. If we use RSE to move data between registers and memory, the overhead is similar to what we see in processors like the Pentium that use the stack for activation records.

14.5.2 Itanium Instruction Set

Instruction Format

A typical Itanium instruction uses a three-operand format. The general syntax is

```
[(qp)] mnemonic[.comp] dests = srcs
```

The syntax is quite different from what we have seen before for the Pentium and PowerPC processors. The optional qualifying predicate `(qp)` specifies a predicate register that indicates whether the instruction should be executed. Recall that the Itanium has 64 1-bit predicate registers (see Figure 14.4). A instruction is executed only if the specified predicate register has a true (1) value; otherwise, the instruction is treated as a NOP (no operation). If a predicate register is not specified in an instruction, predicate register p0 is used, which is always true. Note that some instructions cannot be predicated.

The `mnemonic` field identifies an instruction and is similar to the mnemonic field in Pentium and PowerPC instructions. However, for some instructions, `mnemonic` identifies only a generic instruction such as comparison. Such instructions require more information to completely specify the operation. For example, we have to specify the type of comparison: equality, greater than, and so on. One or more completers `comp` can be used for this purpose. Completers indicate optional variations on the basic operation. `dests` is typically a register to receive the result. Most instructions require two source operands and `srcs` specifies these input operands. Some examples of Itanium instructions are given below:

Simple Instruction	`add r1 = r2,r3`
Predicated instruction	`(p4)add r1 = r2,r3`
Instruction with an immediate value	`add r1 = r2,r3,1`
Instruction with completers	`cmp.eq p3 = r2,r4`
	`cmp.gt p2,p3 = r3,r4`
	`br.cloop.sptk loop_back`

Each instruction is encoded using 41 bits as shown in Figure 14.5. This figure and Figure 14.5 show some sample Itanium instruction formats. In every instruction, the least significant 6 bits are used to specify one of the 64 predicate registers. The leftmost 4 bits specify a major opcode that identifies a major operation group such as comparison, floating-point operation, and so on. Opcode extension fields are used to specify subcases of the major operations. In the register format, source and destination register specification take a total of 21 bits. Each register specification needs 7 bits as Itanium has 128 registers (see Figure 14.5*a*).

The Itanium supports three types of immediate formats: 8-, 14-, and 22-bit immediate values can be specified. In the immediate format, the sign of the immediate value is always placed in the fifth leftmost bit (the `S` bit in Figure 14.5*b*). When a 22-bit immediate value is specified, the destination register must be one of the first four registers (`r0` through `r3`), as we have only two bits to specify this register.

Two sample compare instruction formats are presented in Figure 14.5*c*. As we show later, compare instructions can specify two predicate registers `p1` and `p2`. In a typical compare operation, the two source operands are compared, and the result is placed in `p1` and its complement in `p2`. The `c` bit extends the opcode to specify the complement compare relation. For example, if the encoding with `c` = 0 represents a "less than" comparison, `c` = 1 converts this to a "greater than or equal to" type of comparison.

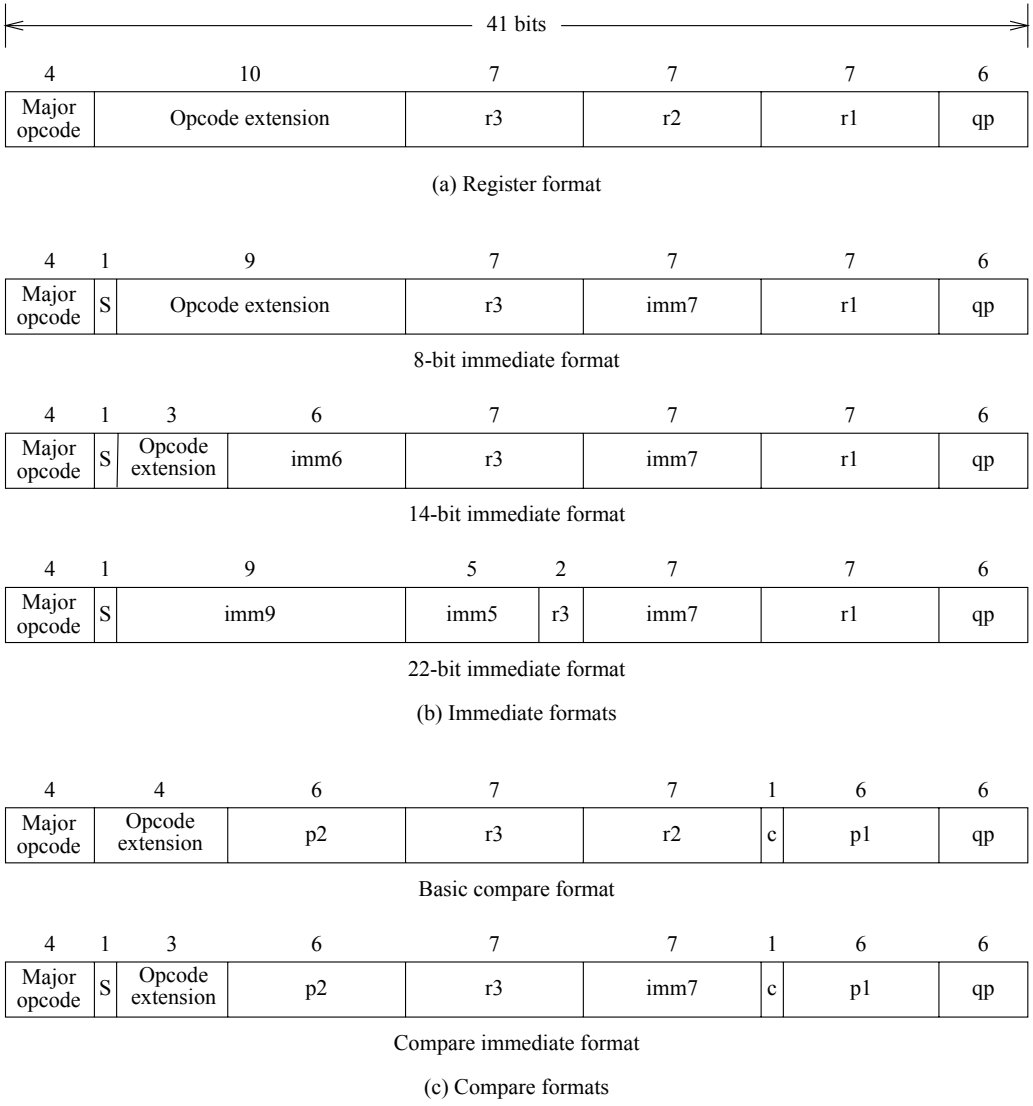
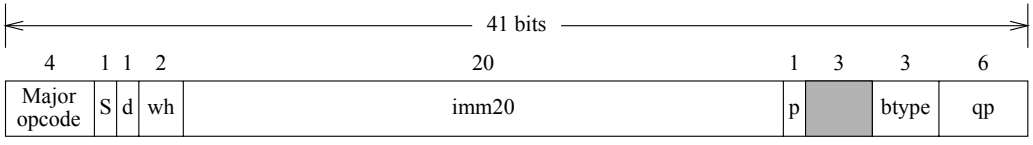
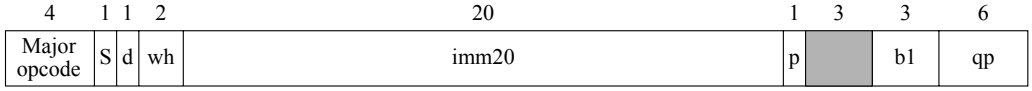


Figure 14.5 Sample Itanium instruction formats.

Figure 14.5*d* shows a sample format for branch and call instructions. Each instruction takes a 21-bit signed IP-relative displacement to the target. Note that the Itanium also supports indirect branch and call instructions. For both instructions, `d` and `wh` fields are opcode extensions to specify hints. The `d` bit specifies the branch cache deallocation hint, and the `wh` field gives the branch whether hint. These two hints are discussed later in this section (see page 604).

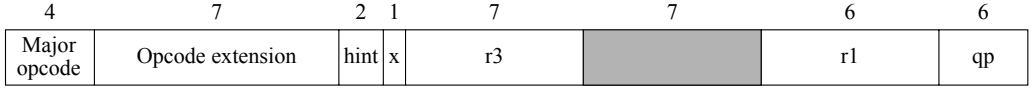


IP-relative branch format

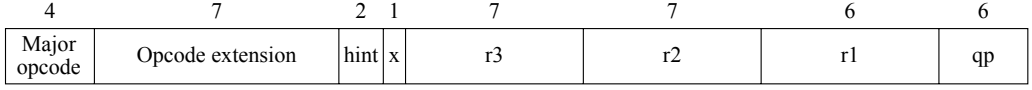


IP-relative call format

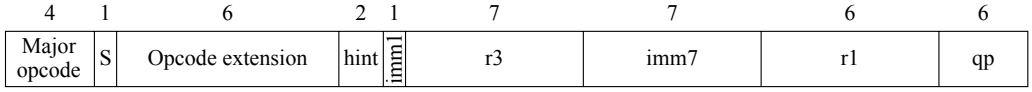
(d) Branch and call formats



Register indirect format



Register indirect with index format



Register indirect with immediate format

(e) Integer load formats

Figure 14.5 *Continued.*

In the branch instruction, the `btype` field indicates the type of branch instruction (e.g., branch on equality, less than, etc.). In the call instruction, the `b1` field specifies the register that should receive the return address.

The integer load instruction formats are shown in Figure 14.5*e*. The `x` bit is used for opcode extension. In the indexed addressing mode, the 9-bit immediate value is split into three pieces: `S`, `imm1`, and `imm7`. The `hint` field, which gives the memory reference hint, is discussed on page 600.

Instruction-Level Parallelism

The Itanium enables instruction-level parallelism by letting the compiler/assembler explicitly indicate parallelism by providing run-time support to execute instructions in parallel, and by

providing a large number of registers to avoid register contention. First we discuss the instruction groups, and then see how the hardware facilitates parallel execution of instructions by bundling nonconflicting instructions together.

Itanium instructions are bound into instruction groups. An instruction group is a set of instructions that do not have conflicting dependencies among them (read-after-write or write-after-write dependencies, as discussed on page 278), and may execute in parallel. The compiler or assembler can indicate instruction groups by using the `;;` notation. Let us look at a simple example to get an idea. Consider evaluating a logical expression consisting of four terms. For simplicity, assume that the results of these four logical terms are in registers `r10`, `r11`, `r12`, and `r13`. Then the logical expression in

```
if (r10 || r11 || r12 || r13) {
      /* if-block code */
}
```

can be evaluated using or-tree reduction as

```
or    r1 = r10,r11        /* Group 1 */
or    r2 = r12,r13;;
or    r3 = r1,r2;;        /* Group 2 */
other instructions        /* Group 3 */
```

The first group performs two parallel OR operations. Once these results are available, we can compute the final value of the logical expression. This final value in `r3` can be used by other instructions to test the condition. Since we have not discussed Itanium instructions, it does not make sense to explain these instructions at this point. We have some examples in a later section.

In any given clock cycle, the processor executes as many instructions from one instruction group as it can, according to its resources. An instruction group must contain at least one instruction; the number of instructions in an instruction group is not limited. Instruction groups are indicated in the code by cycle breaks (`;;`). An instruction group may also end dynamically during run-time by a taken branch.

An advantage of instruction groups is that they reduce the need to optimize the code for each new microarchitecture. Processors with additional resources will take advantage of the existing ILP in the instruction group.

By means of instruction groups, compilers package instructions that can be executed in parallel. It is the compiler's responsibility to make sure that instructions in a group do not have conflicting dependencies. Armed with this information, instructions in a group are bundled together as shown in Figure 14.6. Three instructions are collected into 128-bit, aligned containers called *bundles*. Each bundle contains three 41-bit instruction slots and a 5-bit template field.

The main purpose of the template field is to specify mapping of instruction slots to execution instruction types. Instructions are categorized into six instruction types: integer ALU, non-ALU integer, memory, floating-point, branch, and extended. A specific execution unit may execute each type of instruction. For example, floating-point instructions are executed by the F-unit,

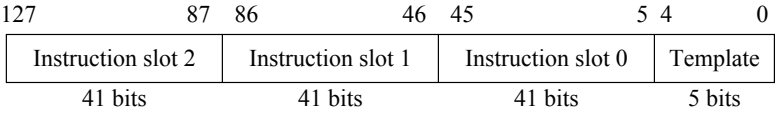


Figure 14.6 Itanium instruction bundle format.

branch instructions by the B-unit, and memory instructions such as load and store by the M-unit. The remaining three types of instructions are executed by the I-unit. All instructions, except extended instructions, occupy one instruction slot. Extended instructions, which use long immediate integers, occupy two instruction slots.

Data Transfer Instructions

The Itanium's load and store instructions are more complex than those in a typical RISC processor. The Itanium supports speculative loads to mask high latency associated with reading data from memory.

The basic load instruction takes one of the three forms shown below depending on the addressing mode used:

```
(qp) ldSZ.ldtype.ldhint  r1 = [r3]         /* No update form */
(qp) ldSZ.ldtype.ldhint  r1 = [r3],r2      /* Update form 1 */
(qp) ldSZ.ldtype.ldhint  r1 = [r3],imm9    /* Update form 2 */
```

The load instruction loads `SZ` bytes from memory, starting at the effective address. The `SZ` completer can be 1, 2, 4, or 8 to load 1, 2, 4, or 8 bytes. In the first load instruction, register `r3` provides the address. In the second instruction, contents of `r3` and `r2` are added to get the effective address. The third form uses a 9-bit signed immediate value, instead of register `r2`. In the last two forms, as explained on page 593, the computed effective address is stored in `r3`.

The `ldtype` completer can be used to specify special load operations. For normal loads, the completer is not specified. For example, the instruction

```
ld8     r5 = [r6]
```

loads eight bytes from the memory starting from the effective address in `r6`. As mentioned before, the Itanium supports speculative loads. Two example instructions are shown below:

```
ld8.a     r5 = [r6]        /* advanced load */
ld8.s     r5 = [r6]        /* speculative load */
```

We defer a discussion of these load instruction types to a later section that discusses the speculative execution model of Itanium.

The `ldhint` completer specifies the locality of the memory access. It can take one of the following three values:

ldhint	Interpretation
None	Temporal locality, level 1
nt1	No temporal locality, level 1
nta	No temporal locality, all levels

A prefetch hint is implied in the two "update" forms of load instructions. The address in r3 after the update acts as a hint to prefetch the indicated cache line. In the "no update" form of load, r3 is not updated and no prefetch hint is implied. Level 1 refers to the cache level. We have not covered temporal locality and cache memory details. Details on cache memory are in Chapter 17. For now, it is sufficient to view the ldhint completer as giving a hint to the processor as to whether a prefetch is beneficial.

The store instruction is simpler than the load instruction. There are two types of store instructions, corresponding to the two addressing modes, as shown below:

```
(qp) stSZ.sttype.sthint  r1 = [r3]         /* No update form */
(qp) stSZ.sttype.sthint  r1 = [r3],imm9    /* Update form */
```

The SZ completer can have four values as in the load instruction. The sttype can be none or rel. If the rel value is specified, an ordered store is performed. The sthint gives a prefetch hint as in the load instruction. However, it can be either none or nta. When no value is specified, temporal locality at level 1 is assumed. The nta has the same interpretation as in the load instruction.

The Itanium also has several move instructions to copy data into registers. We describe three of these instructions:

```
(qp) mov  r1 = r3
(qp) mov  r1 = imm22
(qp) movl r1 = imm64
```

These instructions move the second operand into the r1 register. The first two mov instructions are actually pseudoinstructions. That is, these instructions are implemented using other processor instructions. As we show in the next chapter, the MIPS processor also has several pseudoinstructions. The movl is the only instruction that requires two instruction slots within the same bundle.

Arithmetic Instructions

The Itanium provides only the basic integer arithmetic operations: addition, subtraction, and multiplication. There is no divide instruction, either for integers or floating-point numbers. Division is implemented in software. Let's start our discussion with the add instruction:

```
(qp) add r1 = r2,r3     /* register form */
(qp) add r1 = r2,r3,1   /* plus 1 form */
(qp) add r1 = imm,r3    /* immediate form */
```

In the *plus 1* form, the constant 1 is added as well. In the immediate form, `imm` can be a 14- or 22-bit signed value. If we use a 22-bit immediate value, `r3` can be one of the first four general registers GR0 through GR3 (i.e., only 2 bits are used to specify the second operand register as shown in Figure 14.5).

The immediate form is a pseudoinstruction that selects one of the two processor immediate `add` instructions,

```
(qp) add r1 = imm14,r3
(qp) add r1 = imm22,r3
```

depending on the size of the immediate operand size and value of `r3`.

The move instruction

```
(qp) mov r1 = r3
```

is implemented as

```
(qp) add  r1 = 0,r3
```

The move instruction

```
 (qp) mov r1 = imm22
```

is implemented as

```
(qp) add  r1 = imm22,r0
```

Remember that `r0` is hardwired to value zero.

The subtract instruction `sub` has the same format as the `add` instruction. The contents of register `r3` are subtracted from the contents of `r2`. In the *minus 1* form, the constant 1 is also subtracted. In the immediate form, `imm` is restricted to an 8-bit value.

The instruction `shladd` (shift left and add)

```
(qp) shladd r1 = r2,count,r3
```

is similar to the `add` instruction, except that the contents of `r2` are left-shifted by `count` bit positions before adding. The `count` operand is a 2-bit value, which restricts the shift to 1-, 2-, 3-, or 4-bit positions.

Integer multiply is done using the `xma` instruction and floating-point registers. This instruction multiplies two 64-bit integers and adds the product to another 64-bit value.

Logical Instructions

Logical operations AND, OR, and XOR are supported by three logical instructions. There is no NOT instruction. However, the Itanium has an *and-complement* (`andcm`) instruction that complements one of the operands before performing the bitwise-AND operation.

All instructions have the same format. We illustrate the format of these instructions for the `and` instruction:

```
(qp) and r1 = r2,r3
(qp) and r1 = imm8,r3
```

The other three operations use the mnemonics `or`, `xor`, and `andcm`. The and-complement instruction complements the contents of `r3` and ANDs it with the first operand (contents of `r2` or immediate value `imm8`).

Shift Instructions

Both left- and right-shift instructions are available. The shift instructions

```
(qp) shl r1 = r2,r3
(qp) shl r1 = r2,count
```

left-shift the contents of `r2` by the count value specified by the second operand. The count value can be specified in `r3` or given as a 6-bit immediate value. If the count value in `r3` is more than 63, the result is all zeros.

Right-shift instructions use a similar format. Since right-shift can be arithmetic or logical depending on whether the number is signed or unsigned, two versions are available. The register versions of the right-shift instructions are shown below:

```
(qp) shr     r1 = r2,r3    (signed right shift)
(qp) shr.u   r1 = r2,r3    (unsigned right shift)
```

In the second instruction, the completer `u` is used to indicate unsigned shift operation. We can also use a 6-bit immediate value for shift count as in the `shl` instruction.

Comparison Instructions

The compare instruction uses two completers as shown below:

```
(qp) cmp.crel.ctype p1,p2=r2,r3
(qp) cmp.crel.ctype p1,p2=imm8,r3
```

The two source operands are compared and the result is written to the two specified destination predicate registers. The type of comparison is specified by `crel`. We can specify one of 10 relations for signed and unsigned numbers. The relations "equal" (`eq`) and "not equal" (`neq`) are valid for both signed and unsigned numbers. For signed numbers, there are four relations to test for: "$<$" (`lt`), "$\leq$" (`le`), "$>$" (`gt`), and "$\geq$" (`ge`). The corresponding relations for testing unsigned numbers are `ltu`, `leu`, `gtu`, and `geu`. The relation is tested as "`r2 rel r3`".

The `ctype` completer specifies how the two predicate registers are to be updated. The normal type (default) writes the comparison result in the `p1` register and its complement in the `p2` register. This would allow us to select one of the two branches (we show an example on page 605). `ctype` allows specification of other types such as `and` and `or`. If `or` is specified, both `p1` and `p2` are set to 1 only if the comparison result is 1; otherwise, the two predicate registers are not altered. This is useful for implementing OR-type simultaneous execution. Similarly, if `and` is specified, both registers are set to 0 if the comparison result is 0 (useful for AND-type simultaneous execution).

Branch Instructions

As does the PowerPC, the Itanium uses branch instruction for traditional jump as well as procedure call and return. The generic branch is supplemented by a completer to specify the type of branch. The branch instruction supports both direct and indirect branching. All direct branches are IP relative (i.e., the target address is relative to the IP value as in the Pentium). Some sample branch instruction formats are shown below:

IP Relative Form:

```
(qp) br.btype.bwh.ph.dh   traget25       (Basic form)
(qp) br.btype.bwh.ph.dh   b1=traget25    (Call form)
     br.btype.bwh.ph.dh   traget25       (Counted loop form)
```

Indirect Form:

```
(qp) br.btype.bwh.ph.dh   b2             (Basic form)
(qp) br.btype.bwh.ph.dh   b1=b2          (Call form)
```

As can be seen, branch uses up to four completers. The `btype` specifies the type of branch. The other three completers provide hints and are discussed later.

For the basic branch, `btype` can be either `cond` or none. In this case, the branch is taken if the qualifying predicate is 1; otherwise, the branch is not taken. The IP-relative target address is given as a label in the assembly language. The assembler translates this into a signed 21-bit value that gives the difference between the target bundle and the bundle containing the branch instruction. Since the target pointer is to a bundle of 128 bits, the value (`target25−IP`) is shifted right by 4 bit positions to get a 21-bit value. Note that the format shown in Figure 14.5*d* (page 597) uses a 21-bit displacement value.

To invoke a procedure, we use the second form and specify `call` for `btype`. This turns the branch instruction into a condition call instruction. The procedure is invoked only if the qualifying predicate is true. As part of the call, it places the current frame marker and other relevant state information in the previous function state application register. The return link value is saved in the `b1` branch register for use by the return instruction.

There is also an unconditional (no qualifying predicate) counted loop version. In this branch instruction (the third one), `btype` is set to `cloop`. If the loop count (LC) application register `ar65` is not zero, it is decremented and the branch is taken.

We can use `ret` as the branch type to return from a procedure. It should use the indirect form and specify the branch register in which the `call` has placed the return pointer. In the indirect form, a branch register specifies the target address. The return restores the caller's stack frame and privilege level.

The last instruction can be used for an indirect procedure call. In this branch instruction, the `b2` branch register specifies the target address and the return address is placed in the `b1` branch register.

Let us look at some examples of branch instructions. The instruction

```
(p3) br  skip     or     (p3) br.cond  skip
```

transfers control to the instruction labeled `skip`, if the predicate register `p3` is 1.

The code sequence

```
      mov   lc = 100
loop_back:
        . . .
      br.cloop   loop_back
```

executes the loop body 100 times. A procedure call may look like

```
(p0)  br.call   br2 = sum
```

whereas the return from procedure `sum` uses the indirect form

```
(p0)  br.ret   br2
```

Since we are using predicate register 0, which is hardwired to 1, both the call and return become unconditional.

The `bwh` (branch whether hint) completer can be used to convey whether the branch is taken (see page 610). The `ph` (prefetch hint) completer gives a hint about sequential prefetch. It can take either `few` or `many`. If the value is `few` or none, few lines are prefetched; many lines are prefetched when `many` is specified. The two levels—`few` and `many`—are system defined. The final completer `dh` (deallocation hint) specifies whether the branch cache should be cleared. The value `clr` indicates deallocation of branch information.

14.5.3 Handling Branches

Pipelining works best when we have a linear sequence of instructions. Branches cause pipeline stalls, leading to performance problems. How do we minimize the adverse effects of branches? There are three techniques to handle this problem:

- *Branch Elimination:* The best solution is to avoid the problem in the first place. This argument may seem strange as programs contain lots of branch instructions. Although we cannot eliminate all branches, we can eliminate certain types of branches. This elimination cannot be done without support at the instruction-set level. We look at how the Itanium uses predication to eliminate some types of branches.
- *Branch Speedup:* If we cannot eliminate a branch, at least we can reduce the amount of delay associated with it. This technique involves reordering instructions so that instructions that are not dependent on the branch/condition can be executed while the branch instruction is processed. Speculative execution can be used to reduce branch delays. We describe the Itanium's speculative execution strategies in Section 14.5.5.

- *Branch Prediction:* If we can predict whether the branch will be taken, we can load the pipeline with the right sequence of instructions. Even if we predict correctly all the time, it would only convert a conditional branch into an unconditional branch. We still have the problems associated with unconditional branches. We described three types of branch prediction strategies in Section 8.4.2 (see page 283).

Since we covered branch prediction in Section 8.4.2, we discuss the first two techniques next.

14.5.4 Predication to Eliminate Branches

In the Itanium, branch elimination is achieved by a technique known as *predication*. The trick is to make execution of each instruction conditional. Thus, unlike the instructions we have seen so far, an instruction is not automatically executed when the control is transferred to it. Instead, it will be executed only if a condition is true. This requires us to associate a predicate with each instruction. If the associated predicate is true, the instruction is executed; otherwise, it is treated as a `nop` instruction. The Itanium architecture supports full predication to minimize branches. Most of the Itanium's instructions can be predicated.

To see how predication eliminates branches, let us look at the following example:

```
if (R1 == R2)                          cmp    r1,r2
    R3 = R3 + R1;                      je     equal
else                                   sub    r3,r1
    R3 = R3 - R1;                      jmp    next
                                equal:
                                       add    r3,r1
                                next:
```

The code on the left-hand side, expressed in C, is a simple if-then-else statement. The Pentium assembly language equivalent is shown on the right. As you can see, it introduces two branches: unconditional (`jmp`) and conditional (`je`). Using the Itanium's predication, we can express the same as

```
        cmp.eq p1,p2 = r1,r2
(p1)    add r3 = r3,r1
(p2)    sub r3 = r3,r1
```

The compare instruction sets two predicates after comparing the contents of `r1` and `r2` for equality. The result of this comparison is placed in `p1` and its complement in `p2`. Thus, if the contents of `r1` and `r2` are equal, `p1` is set to 1 (true) and `p2` to 0 (false). Since the `add` instruction is predicated on `p1`, it is executed only if `p1` is true. It should be clear that either the `add` or the `sub` instruction is executed, depending on the comparison result.

To illustrate the efficacy of predicated execution, we look at the following `switch` statement in C:

```
switch (r6)
{
    case 1:
            r2 = r3 + r4;
            break;
    case 2:
            r2 = r3 - r4;
            break;
    case 3:
            r2 = r3 + r5;
            break;
    case 4:
            r2 = r3 - r5;
            break;
}
```

For simplicity, we are using the register names in the `switch` statement. Translating this statement would normally involve a series of compare and branch instructions. Predication avoids this sequence as shown below:

```
     cmp.eq  p1,p0 = r6,1
     cmp.eq  p2,p0 = r6,2
     cmp.eq  p3,p0 = r6,3
     cmp.eq  p4,p0 = r6,4 ;;

(p1) add  r2 = r3,r4
(p2) sub  r2 = r3,r4
(p3) add  r2 = r3,r5
(p4) sub  r2 = r3,r5
```

In the first group of instructions, the four compare instructions set `p1/p2/p3/p4` if the corresponding comparison succeeds. If the processor has resources, all four instructions can be executed concurrently. Since `p0` is hardwired to 1, failure conditions are ignored in the above code. Depending on the compare instruction that succeeds, only one of the four arithmetic instructions in the second group is executed.

14.5.5 Speculative Execution

Speculative execution refers to the scenario where instructions are executed in the expectation that they will be needed in actual program execution. The main motivation, of course, is to improve performance. There are two main reasons to speculatively execute instructions: to keep the pipeline full and to mask memory access latency. We discuss two types of speculative execution supported by the Itanium: one type handles data dependencies, and the other deals with control dependencies. Both techniques are compiler optimizations that allow the compiler to reorder instructions. For example, we can speculatively move high-latency load instructions earlier so that the data are available when they are actually needed.

Data Speculation

Data speculation allows the compiler to schedule instructions across some types of ambiguous data dependencies. When two instructions access common resources (either registers or memory locations) in a conflicting mode, data dependency exists. A conflicting access is one in which one or both instructions alter the data. Depending on the type of conflicting access, we can define the following dependencies: read-after-write (RAW), write-after-read (WAR), write-after-write (WAW), and ambiguous data dependencies. We have discussed the first three types of dependencies in Section 8.3 on page 278. Ambiguous data dependency exists when pointers are used to access memory. Typically, in this case, dependencies between load and store instructions or store and store instructions cannot be resolved statically at compile time because we don't know the pointer values. Handling this type of data dependency requires run-time support.

The first three dependencies are not ambiguous in the sense that the dependency type can be statically determined at compile/assembly time. The compiler or programmer should insert stops (;;) so that the dependencies are properly maintained. The example

```
sub    r6=r7,r8 ;;
add    r9=r10,r6
```

exhibits a RAW data dependency on `r6`. The stop after the `sub` instruction would allow the `add` instruction to read the value written by the `sub` instruction.

If there is no data dependency, the compiler can reorder instructions to optimize the code. Let us look at the following example:

```
sub    r6=r7,r8 ;;     // cycle 1

sub    r9=r10,r6       // cycle 2
ld8    r4=[r5] ;;

add    r11=r12,r4      // cycle 4
```

Since there is a two-cycle latency to the first-level data cache, the `add` instruction is scheduled two cycles after scheduling the `ld8` instruction. A straightforward optimization involves moving the `ld8` instruction to cycle 1 as there are no data dependencies to prevent this reordering. By advancing the load instruction, we can schedule the `add` in cycle 3 as shown below:

```
ld8    r4=[r5]         // cycle 1
sub    r6=r7,r8 ;;

sub    r9=r10,r6 ;;    // cycle 2

add    r11=r12,r4      // cycle 3
```

However, when there is ambiguous data dependency, as in the following example, instruction reordering may not be possible:

```
sub    r6=r7,r8 ;;    // cycle 1

st8    [r9]=r6        // cycle 2
ld8    r4=[r5] ;;

add    r11=r12,r4 ;;  // cycle 4

st8    [r10]=r11      // cycle 5
```

In this code, ambiguous dependency exists between the first `st8` and `ld8` because `r9` and `r5` could be pointing to overlapped memory locations. Remember that each of these instructions accesses eight contiguous memory locations. This ambiguous dependency will not allow us to move the load instruction to cycle 1 as in the previous example.

The Itanium provides architectural support to move such load instructions. This is facilitated by advance load (`ld.a`) and check load (`ld.c`). The basic idea is that we initiate the load early and when it is time to actually execute the load, we will make a check to see if there is a dependency that invalidates our advance load data. If so, we reload; otherwise, we successfully advance the load instruction even when there is ambiguous dependency. The previous example with advance and check loads is shown below:

```
1:  ld8.a  r4=[r5]        // cycle 0 or earlier
        . . .
2:  sub    r6=r7,r8 ;;    // cycle 1

3:  st8    [r9]=r6       // cycle 2
4:  ld8.c  r4=[r5]
5:  add    r11=r12,r4 ;;

6:  st8    [r10]=r11      // cycle 3
```

We inserted an advance load (line 1) at cycle 0 or earlier so that `r4` would have the value ready for the `add` instruction on line 5 in cycle 2. However, we have to check to see if we can use this value. This check is done by the check load instruction `ld8.c` on line 4. If there is no dependency between the store on line 3 and load on line 4, we can safely use the prefetched value. This is the case if the pointers in `r9` and `r5` are different. On the other hand, if the load instruction is reading the value written by the store instruction, we have to reload the value. The check load instruction on line 4 automatically reloads the value in the case of a conflict.

In the last example, we advanced just the load instruction. However, we can improve performance further if we can also advance all (or some of) the statements that depend on the value read by the load instruction. In our example, it would be nice if we could advance the `add` instruction on line 5. This causes a problem if there is a dependency between the store and load instructions (on lines 3 and 4). In this case, we not only have to reexecute the load but also the `add` instruction. The advance check (`chk.a`) instruction provides the necessary support for such reexecution as shown in the following example:

```
        ld8.a  r4=[r5]      // cycle -1 or earlier
           . . .
        add    r11=r12,r4   // cycle 1
        sub    r6=r7,r8 ;;

        st8    [r9]=r6      // cycle 2
        chk.a  r4,recover
back:
        st8    [r10]=r11

recover:
        ld8    r4=[r5]      // reload
        add    r11=r12,r4   // reexecute add
        br     back         // jump back
```

When the advanced load fails, the check instruction transfers control to `recover` to reload and reexecute all the instructions that used the value provided by the advanced load.

How does the Itanium maintain the dependency information for use by the check instructions? It keeps a hardware structure called the advanced load address table (ALAT), indexed by the physical register number. When an advanced load (`ld.a`) instruction is executed, it records the load address. When a check is executed (either `ld.c` or `chk.a`), it checks ALAT for the address. The check instruction must specify the same register that the advanced load instruction used. If the address is present in ALAT, execution continues. Otherwise, a reload (in case of `ld.c`) or recovery code (in case of `chk.a`) is executed. An entry in ALAT can be removed, for example, by a subsequent store that overlaps the load address. To determine this overlap, the size of the load in bytes is also maintained.

Control Speculation

When we want to reduce latencies of long latency instructions such as load, we advance them earlier into the code. When there is a branch instruction, it blocks such a move because we do not know whether the branch will be taken until we execute the branch instruction. Let us look at the following code fragment:

```
             cmp.eq  p1,p0 = r10,10   // cycle 0
        (p1) br.cond   skip ;;        // cycle 0
             ld8   r1 = [r2] ;;       // cycle 1
             add   r3 = r1,r4         // cycle 3
skip:
             // other instructions
```

In the above code, we cannot advance the load instruction due to the branch instruction. Execution of the *then branch* takes four clock cycles. Note that the code implies that the integer compare instruction that sets the predicate register `p1` and the branch instruction that tests it are executed in the same cycle. This is the only exception; in general, there should be a stop

inserted between an instruction that is setting a predicate register and the subsequent instruction testing the same predicate.

Now the question is: How do we advance the load instruction past the branch instruction? We speculate in a manner similar to the way we handled data dependency. There is one additional problem: Since the execution may not take the branch, if the speculative execution causes exceptions, they should not be raised. Instead, exceptions should be deferred until we know that the instruction will indeed be executed. The not-a-thing bit is used for this purpose. If a speculative execution causes an exception, the `NaT` bit associated with that register is set. When a check is made at the point of actual instruction execution, if the deferred exception is not present, speculative execution was successful. Otherwise, the speculative execution should be redone. Let us rewrite the previous code with a speculative load.

```
            ld8.s   r1 = [r2] ;;     // cycle -2 or earlier

             // other instructions

            cmp.eq  p1,p0 = r10,10   // cycle 0
      (p1) br.cond   skip            // cycle 0
            chk.s   r1, recovery     // cycle 0
            add   r3 = r1,r4         // cycle 0
skip:
            // other instructions
recovery:
            ld8   r1 = [r2]
            br    skip
```

The load instruction is moved by at least two cycles so that the value is available in `r1` by the time it is needed by the `add` instruction. Since this is a speculative load, we use the `ld8.s` instruction. And, in place of the original load instruction, we insert a speculative check (`chk.s`) instruction with the same register `r1`. As in the data dependency example, if the speculative load is not successful (i.e., the `NaT` bit of `r1` is set), the instruction has to be reexecuted. In this case, the check instruction branches to the `recovery` code.

14.5.6 Branch Prediction

As mentioned in Chapter 8, most processors use branch prediction to handle the branch problem. We discussed three branch prediction strategies—fixed, static, and dynamic—in Section 8.4.2.

In the Itanium, branch hints can be explicitly provided in branch instructions. The `bwh` completer can take one of the following four values:

`spnt`	Static branch not taken
`sptk`	Static branch taken
`dpnt`	Dynamic branch not taken
`dpnt`	Dynamic branch taken

The SPARC processor also allows providing hints for the branch instructions (see Appendix H for specifics).

We have given a detailed overview of the Itanium processor. Certainly, we have left out several elements. If you are interested in learning more about this processor, visit the Intel Web site for recent information and Itanium manuals [21].

14.6 Summary

We have introduced important characteristics that differentiate RISC processors from their CISC counterparts. CISC processors such as the Pentium provide complex instructions and a large number of addressing modes compared to RISC processors. The rationale for this complexity is the desire to close the semantic gap that exists between high-level languages and machine languages. In the early days, effective usage of processor and memory resources was important. Complex instructions tend to minimize the memory requirements. However, implementing complex instructions in hardware caused problems until the advent of microcode. With microcoded implementations, designers were carried away and started making instruction sets very complex to reduce the previously mentioned semantic gap. The VAX 11/780 is a classic example of such complex processors.

Empirical data, however, suggested that compilers do not use these complex instructions; instead, they use simple instructions to synthesize complex instructions. Such observations led designers to take a fresh look at processor design philosophy. RISC principles, based on empirical studies on CISC processors, have been proposed as an alternative to CISC processors. Most of the current processor designs are based on these RISC principles.

We have presented details on two processors, the PowerPC and Itanium, that follow the RISC design philosophy to a varying degree. The Itanium is a more advanced processor in terms of the features provided. Compared to the PowerPC and Pentium, it supports explicit parallelism by providing ample resources such as registers and functional units. In addition, the Itanium supports sophisticated speculative loads and predication. It also uses traditional branch prediction strategies. The Itanium, even though it follows RISC principles, is not a "simple" processor. Some of its instructions truly belong to the CISC category.

Both the PowerPC and Itanium do not strictly follow all the RISC principles. In the next chapter, we present details on the MIPS processor, which closely follows the RISC philosophy.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Activation record
- Addressing modes
- Advanced load
- Branch elimination
- Branch hints
- Branch prediction
- Branch speedup
- Condition register

- Control speculation
- Data dependency
- Data speculation
- EPIC design
- Immediate index addressing mode
- Index addressing mode
- Indirect addressing mode
- Instruction-level parallelism
- Load/store architecture
- Microprogramming
- Predicated execution
- Procedure call in the Itanium
- Register renaming
- Register windows
- Stack frame
- Speculative execution
- Speculative load

14.7 Exercises

14–1 We have seen that CISC processors typically use variable-length instructions. Discuss the reasons for this.

14–2 A typical CISC processor supports several addressing modes. For example, the VAX 11/780 provides 22 addressing modes. What is the motivation for supporting a large number of addressing modes?

14–3 We have stated that RISC is a design philosophy. Discuss the main RISC characteristics.

14–4 Recent processors tend to follow the RISC design philosophy. For example, Intel moved from CISC to RISC designs for their 64-bit processors. Explain why.

14–5 The two RISC processors we have discussed in this chapter use the load/store architecture. The MIPS processor, discussed in the next chapter, also uses the load/store architecture. Why do RISC processors use the load/store architecture?

14–6 RISC processors provide a large number of registers compared to CISC processors. What is the rationale for this? How do these processors use these registers?

14–7 What is the purpose of the condition register in the PowerPC? Is there a similar register in the Pentium?

14–8 What is the purpose of the link, XER, and count registers in the PowerPC? Are there similar registers in the Pentium?

14–9 Discuss the addressing modes supported by the PowerPC.

14–10 The PowerPC provides only a few addressing modes. Compared to the addressing modes supported by the Pentium, which addressing modes are missing from the PowerPC list?

14–11 The PowerPC and Itanium appear to support the same types of addressing modes. What is the main difference between the addressing modes of these two processors? How does this difference benefit the Itanium processor?

14–12 Assume that two 64-bit integers are stored in memory at `number1` and `number2`. Suppose that we want to add these two numbers on a 32-bit PowerPC processor and store the result in memory at `result`. Write a PowerPC assembly language code fragment to perform this addition.

14–13 Discuss the differences between the multiply instructions of the Pentium and PowerPC.

14–14 In the PowerPC, branch instructions have BO and BI operands. What is the purpose of these two operands?

14–15 Explain the EPIC design philosophy of the Itanium processor.

14–16 Pentium and PowerPC processors provide 8 and 32 general-purpose registers, respectively. The Itanium, on the other hand, provides 128 general-purpose registers. Why did the designers of the Itanium decide to provide such a large number of registers?

14–17 The Itanium has 128 predicate registers, one for each general-purpose register. Each is a single-bit register. What is the purpose of this register set?

14–18 The Itanium provides eight 64-bit branch registers. Pentium and PowerPC processors do not have branch registers. What purpose do these registers serve in the Itanium?

14–19 The Itanium divides the 128 general-purpose registers into static and stacked registers. What is the main reason for this? How can this division be exploited to improve performance?

14–20 Explain how instruction-level parallelism is extracted in the Itanium. What are the advantages of this scheme over hardware-based approaches?

14–21 Instructions of the Itanium are 41-bits long. Three such instructions are packed together into 128-bit bundles. How does this bundling improve performance of the Itanium processor?

14–22 The Itanium instruction bundle contains a 5-bit template. What is the purpose of this template?

14–23 Branches cause performance problems. Discuss three ways to handle the branches.

14–24 Explain how the Itanium uses predication to eliminate certain types of branches. Use an example to illustrate your point.

14–25 What are data and control dependencies? Which is more difficult to handle?

14–26 Explain how the Itanium uses speculative execution to handle data and control dependencies.

Chapter 15

MIPS Assembly Language

Objectives

- To describe MIPS processor architecture;
- To present MIPS processor instruction set details;
- To illustrate how MIPS assembly language programs are written;
- To discuss MIPS stack implementation and procedures.

In the last chapter, we have briefly discussed two RISC processors, the PowerPC and Intel Itanium. In this chapter, we take a detailed look at the MIPS R2000 processor. There is also an opportunity to program in the MIPS assembly language. The MIPS simulator SPIM runs the programs written for the MIPS processor. Appendix G gives details on the SPIM simulator. It also gives information on how you can download and use the SPIM to run the examples discussed in this chapter.

We start this chapter with a description of the MIPS processor architecture. The following section discusses its instruction set in detail. The SPIM provides a few system calls to facilitate input/output from the assembly language programs. These calls are given in Section 15.3. As do the Pentium assemblers, the SPIM also provides several directives, which are described in Section 15.4. Some example MIPS assembly programs are given in Section 15.5.

Unlike the Pentium, simple procedures in MIPS processors do not use the stack. Section 15.6 explains how procedures are written in the MIPS assembly language. This section also gives examples to illustrate the principles involved in writing procedures. Although the stack is not needed to write simple procedures, nested or recursive procedures need to use the stack. Stack implementation is described in Section 15.7 with some examples. We conclude the chapter with a summary.

15.1 MIPS Processor Architecture

Our focus in this chapter is on the MIPS R2000 RISC processor. The main reason for selecting this specific MIPS processor is that the SPIM simulator is written for this processor. This is a 32-bit processor. Later processors (R4000 and above) are very similar to R2000 except that they are 64-bit processors. From a pedagogical perspective, the R2000 processor is sufficient to explain the RISC features.

MIPS processors follow the load/store architecture, which means that most instructions operate on registers. It has two instruction types to move data between registers and memory. As we show later, the R2000 provides several load and store instructions to transfer different sizes of data: byte, halfword, word, and doubleword.

Like most recent processors, the R2000 supports both little-endian and big-endian formats. Recall that the Pentium uses the little-endian format to store multibyte data, and the Itanium and PowerPC support both endian formats.

RISC processors typically have a large number of registers. For example, the Itanium has 128 registers, whereas the PowerPC provides 32 registers. We start our discussion with the registers of the R2000.

15.1.1 Registers

The R2000 provides 32 general-purpose registers, a program counter (PC), and two special-purpose registers. All registers are 32-bits wide as shown in Figure 15.1.

Unlike the Pentium, numbers are used to identify the general-purpose registers. In the assembly language, these registers are identified as `$0`, `$1`, . . . , `$31`. Two of the general-purpose registers—the first and the last—are reserved for a specific function:

- Register `$0` is hardwired to zero value. This register is often used as a source register when a zero value is needed. You can use this register as the destination register of an instruction if you want the result to be discarded.
- The last register `$31` is used as a link register by the jump and link (`jal`) instruction (discussed in Section 15.6 on page 643). This instruction is equivalent to the Pentium's `call` instruction. Register `$31` is used to store the return address of a procedure call. We discuss this issue in detail in Section 15.6.

The PC register serves the same purpose as the instruction pointer (IP) register in the Pentium. The two special-purpose registers—called HI and LO—are used to hold the results of integer multiply and divide instructions:

- In the integer multiply operation, HI and LO registers hold the 64-bit result, with the higher-order 32 bits in the HI and the lower-order 32 bits in the LO register.
- In integer divide operations, the 32-bit quotient is stored in the LO and the remainder in the HI register.

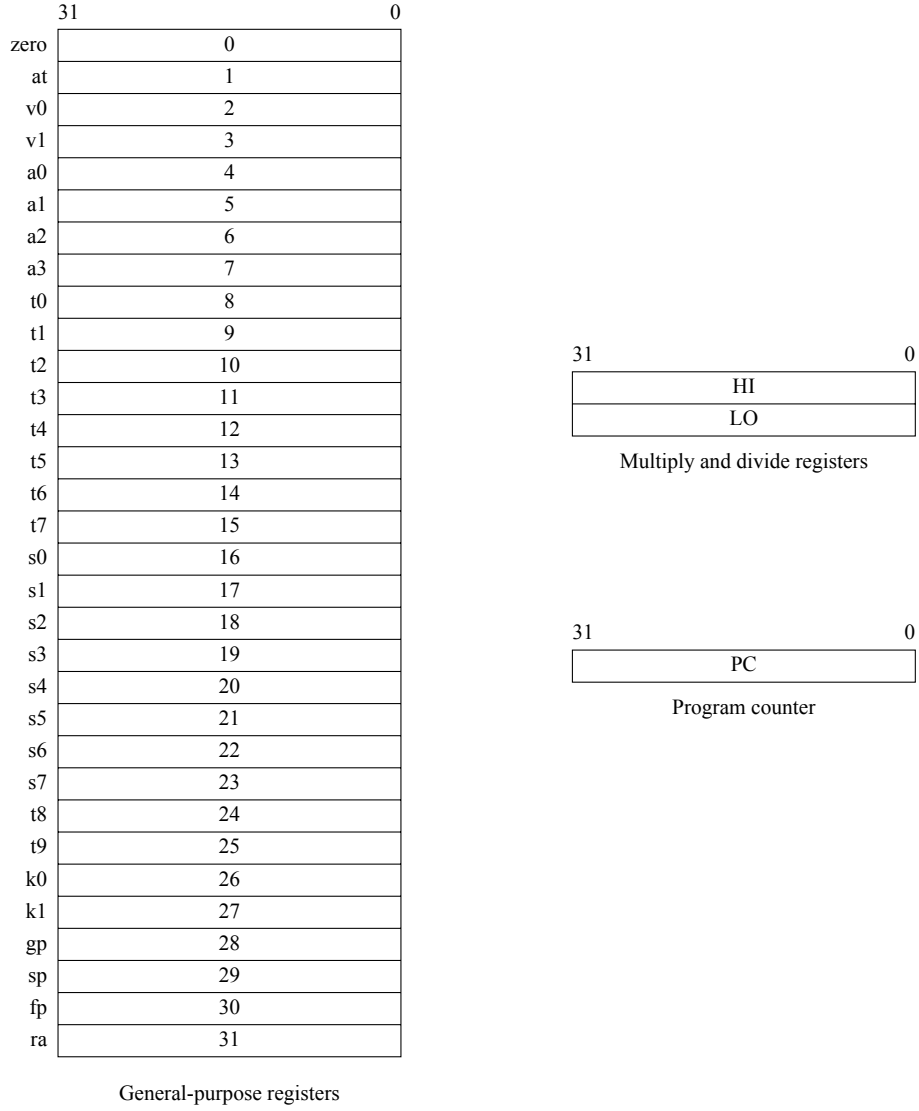


Figure 15.1 MIPS R2000 processor registers. All registers are 32-bits wide.

15.1.2 General-Purpose Register Usage Convention

Although there is no requirement from the processor hardware, the MIPS has established a convention on how these 30 registers should be used. Table 15.1 shows the suggested use of each register. Since these suggestions are not enforced by the hardware, we can use the general-purpose registers in an unconventional manner. However, such programs are not likely to work with other programs.

Table 15.1 MIPS registers and their conventional usage

Register name	Number	Intended usage
zero	0	Constant 0
at	1	Reserved for assembler
v0, v1	2, 3	Results of a procedure
a0, a1, a2, a3	4–7	Arguments 1–4
t0–t7	8–15	Temporary (not preserved across call)
s0–s7	16–23	Saved temporary (preserved across call)
t8, t9	24, 25	Temporary (not preserved across call)
k0, k1	26, 27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer (if needed); otherwise, a saved register `$s8`
ra	31	Return address (used by a procedure call)

Registers `$v0` and `$v1` are used to return results from a procedure. Registers `$a0` to `$a3` are used to pass the first four arguments to procedures. The remaining arguments are passed via the stack.

Registers `$t0` to `$t9` are temporary registers that need not be preserved across a procedure call. These registers are assumed to be saved by the caller. On the other hand, registers `$s0` to `$s7` are callee-saved registers that should be preserved across procedure calls.

Register `$sp` is the stack pointer and serves the same purpose as the Pentium's `sp` register. It points to the last location in use on the stack. The MIPS compiler does not use a frame pointer. As a result, the frame pointer register `$fp` is used as callee-saved register `$s8`. The `$ra` is used to store the return address in a procedure call. We discuss these registers in Section 15.6.

Register `$gp` points to the memory area that holds constants and global variables. The `$at` register is reserved for the assembler. The assembler often uses this register to translate pseudoinstructions. We show some examples of this later (see page 626).

15.1.3 Addressing Modes

The Pentium provides several addressing modes, which is a characteristic of CISC processors. Since the MIPS uses the load/store architecture, only the load and store instructions access memory. Thus, the addressing mode mainly refers to how these two instructions access memory. All other instructions use registers. The bare machine provides only a single memory addressing mode: `disp(Rx)`, where displacement `disp` is a signed, 16-bit immediate value.

Table 15.2 Addressing modes

Format	Address computed as
(Rx)	Contents of register Rx
imm	Immediate value imm
imm(Rx)	imm + contents of Rx
symbol	Address of symbol
symbol± imm	Address of symbol ±imm
symbol± imm(Rx)	Address of symbol ±(imm + contents of Rx)

The address is computed as disp + contents of base register Rx. Thus, the MIPS provides only the based/indexed addressing mode. In the MIPS, we can use any register as the base register.

The virtual machine supported by the assembler provides additional addressing modes for load and store instructions to help in assembly language programming. Table 15.2 shows the addressing modes supported by the virtual machine.

Note that most load and store instructions operate only on aligned data. The MIPS, however, provides some instructions for manipulating unaligned data. For more details on alignment of data and its impact on performance, see our discussion on page 683.

15.1.4 Memory Usage

The MIPS uses a conventional memory layout. A program's address space consists of three parts: code, data, and stack. The memory layout of these three components is shown in Figure 15.2. The text segment, which stores the instructions, is placed at the bottom of the user address space (at 4000000H).

The data segment is placed above the text segment and starts at 10000000H. The data segment is divided into static and dynamic areas. The dynamic area grows as memory is allocated to dynamic data structures.

The stack segment is placed at the end of the user address space at 7FFFFFFFH. It grows downward towards lower memory address. This placement of segments allows sharing of unused memory by both data and stack segments.

15.2 MIPS Instruction Set

The MIPS instruction set consists of *instructions* and *pseudoinstructions*. The MIPS processor supports only the instructions. Pseudoinstructions are provided by the assembler for convenience in programming. The assembler translates pseudoinstructions into a sequence of one or more processor instructions. We use a † to indicate the pseudoinstructions.

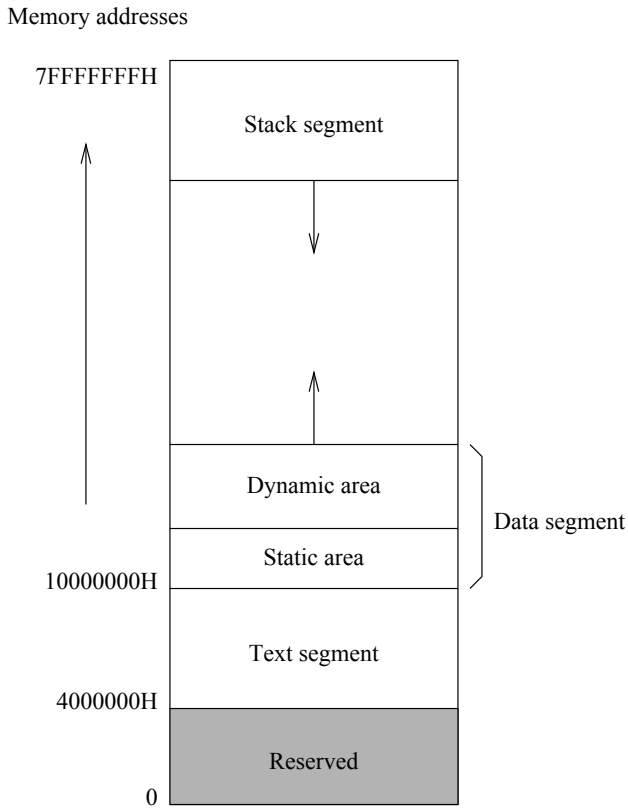


Figure 15.2 MIPS memory layout.

15.2.1 Instruction Format

The MIPS, being a RISC processor, uses a fixed-length instruction format. Each instruction is 32-bits long as shown in Figure 15.3. It uses only three different instruction formats, as opposed to the large number of formats used by the Pentium. The three formats are as follows:

- *Immediate (I-type)*: All load and store instructions use this instruction format. The immediate value is a signed 16-bit integer. In addition, arithmetic and logical instructions that use an immediate value also use this format. Branch instructions use a 16-bit signed offset relative to the program counter and are encoded in the I-type format.
- *Jump (J-type)*: Jump instructions that specify a 26-bit target address use this instruction format. These 26 bits are combined with the higher-order bits of the program counter to get the absolute address.
- *Register (R-type)*: Arithmetic and logical instructions use this instruction format. In addition, the jump instruction in which the target address is specified indirectly via a register also uses this instruction format.

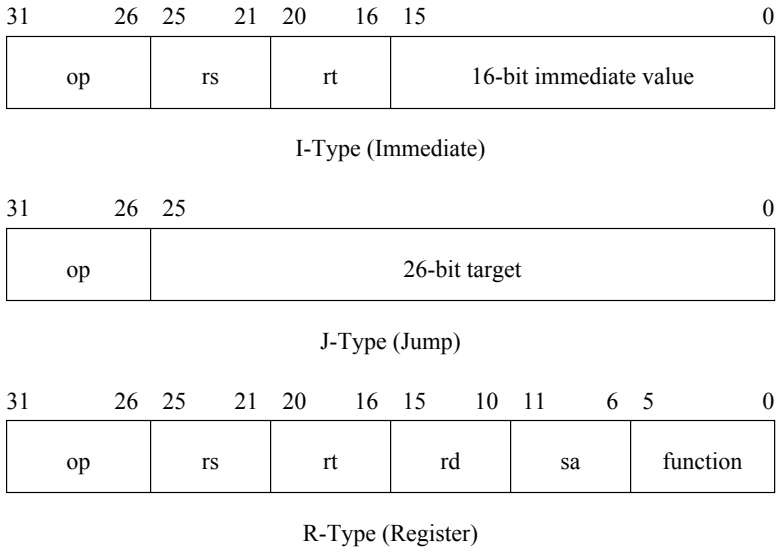


Figure 15.3 Three instruction formats of the MIPS R2000 processor.

The use of a limited number of instruction formats simplifies instruction decoding. However, three instruction formats and a single addressing mode mean that complicated operations and addressing modes will have to be synthesized by the compiler. If these operations and addressing modes are less frequently used, we may not pay much penalty. This is the motivation behind the RISC processors.

15.2.2 Data Transfer Instructions

The MIPS provides load and store instructions to move data between memory and registers. Load instructions move data from memory into registers, and the store instructions move the data in the opposite direction. Load and store instructions have a similar format. Therefore, we discuss the load instructions in more detail.

Moving Data

Several load and store instructions are available to move data of different sizes. The load byte (`lb`) instruction moves a byte of data from memory to a register. The format is

```
lb      Rdest,address
```

`lb` loads the least significant byte of `Rdest` with the byte at the specified memory address. The byte is treated as a signed number. Consequently, the sign bit is extended to the remaining three bytes of `Rdest`. To load an unsigned number, use load byte unsigned (`lbu`) instead of `lb`. In this case, the remaining three bytes are filled with zeros.

Table 15.3 Sample MIPS load instructions

Instruction	Description
`lb Rdest,address`	Load byte: Loads the byte at `address` in memory into the least significant byte of `Rdest`. The byte is treated as a signed number; sign extends to the remaining three bytes of `Rdest`.
`lbu Rdest,address`	Load byte unsigned: This instruction is similar to `lb` except that the byte is treated as an unsigned number. Upper three bytes of `Rdest` are filled with zeros.
`lh Rdest,address`	Load halfword: Loads the half-word (two bytes) at `address` in memory into the least significant two bytes of `Rdest`. The 16-bit data is treated as a signed number; sign extends to the remaining two bytes of `Rdest`.
`lhu Rdest,address`	Load halfword unsigned: Same as `lh` except that the 16-bit halfword is treated as an unsigned number.
`lw Rdest,address`	Load word: Loads the word (four bytes) at `address` in memory into `Rdest`.
Additional instructions available on 64-bit processors	
`lwu Rdest,address`	Load word unsigned: Loads the word (four bytes) at `address` in memory into the least significant four bytes of `Rdest`. The 32-bit data are treated as an unsigned number; sign extends to the remaining four bytes of `Rdest`.
`ld Rdest,address`	Load doubleword: Loads the doubleword (eight bytes) at `address` in memory into `Rdest`.
Assembler pseudoinstructions	
`la`† `Rdest,address`	Load address: Loads `address` into `Rdest`.
`li`† `Rdest,imm`	Load immediate: Loads the immediate value `imm` into `Rdest`.

Other load instructions facilitate movement of larger data items. These instructions are summarized in Table 15.3. Note that in 64-bit architectures, load word `lw` copies the 32-bit memory content into the least significant four bytes of `Rdest`. In addition, we can use the doubleword transfer instruction.

The assembler provides two pseudoinstructions to load an address or an immediate value into a register. For example,

```
la      $a0,marks
```

loads the address of the `marks` array into the `$a0` register. The `li` instruction is implemented as

```
ori     Rdest,$0,imm
```

The `ori` (OR immediate) instruction is discussed in Section 15.2.4.

Table 15.4 Sample MIPS store instructions

Instruction	Description
`sb Rsrc,address`	Store byte: Stores the least significant byte of `Rsrc` at the specified `address` in memory.
`sh Rsrc,address`	Store halfword: Stores the least significant two bytes (halfword) of `Rsrc` at the specified `address` in memory.
`sw Rsrc,address`	Store word: Stores the four-byte word from `Rsrc` at the specified `address` in memory.
Additional instructions available on 64-bit processors	
`sd Rsrc,address`	Store doubleword: Stores the eight-byte doubleword from `Rsrc` in memory at the specified `address`.

The store byte (`sb`) instruction

```
sb      Rsrc,address
```

stores the least significant byte of `Rsrc` at the specified memory address. Since the data transfer does not involve sign extension, there is no need for separate instructions to handle signed and unsigned byte transfers. Store instructions to handle 16-, 32-, and 64-bit data are also available as shown in Table 15.4.

To move data between registers, we can use the `move` pseudoinstruction. The format is

```
move†   Rdest,Rsrc
```

It copies the contents of `Rsrc` to `Rdest`. Four additional data movement instructions are available for transferring data between a general register and two special registers HI and LO. These instructions are described on page 625.

15.2.3 Arithmetic Instructions

MIPS supports the four basic arithmetic operations: addition, subtraction, multiplication, and division.

Addition Instructions

The basic addition instruction

```
add     Rdest,Rsrc1,Rsrc2
```

adds contents of `Rsrc1` and `Rsrc2` and stores the result in `Rdest`. The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated. We can use `addu` if

no overflow exception is needed. Except for this, there is no difference between `add` and `addu` instructions.

The second operand can be specified as an immediate 16-bit number. The format is

```
addi    Rdest,Rsrc1,imm
```

The 16-bit value is sign-extended to 32 bits and added to the contents of `Rsrc1`. As in the `add` instruction, an overflow exception is generated. As in `add`, we can use `addiu` if an overflow exception is not needed.

For convenience, assembler provides a pseudoinstruction that can take a register or an immediate value as the second source operand. The format is

```
add†    Rdest,Rsrc1,Src2
```

where `Src2` can be a 16-bit immediate value or a register. Use `addu`† for the no-overflow version.

Subtract Instructions

The subtract instruction

```
sub     Rdest,Rsrc1,Rsrc2
```

subtracts the contents of `Rsrc2` from `Rsrc1` (i.e., `Rsrc1` − `Rsrc2`). The result is stored in `Rdest`. The contents of the two source registers are treated as signed numbers and an integer overflow exception is generated. We use `subu` if this exception is not required.

There is no immediate version of the subtract instruction. It is not really needed as we can treat subtraction as an addition of a negative number. However, we can use the assembler pseudoinstruction to operate on immediate values. The format is

```
sub†    Rdest,Rsrc1,Src2
```

where `Src2` can be a 16-bit immediate value or a register.

To negate a value, we can use the assembler pseudoinstruction `neg` for signed numbers. The instruction

```
neg†    Rdest,Rsrc
```

negates the contents of `Rsrc` and stores the result in `Rdest`. An overflow exception is generated if the value is -2^{31}. Negation without the overflow exception (`negu`†) is also available.

As noted, `neg` is not a processor instruction; the SPIM assembler translates the negate instruction using `sub` as

```
sub    Rdest,$0,Rsrc
```

`abs` is another pseudoinstruction that is useful to get the absolute value. The format is

```
abs†    Rdest,Rsrc
```

This pseudoinstruction is implemented as

```
bgez    Rsrc,skip
sub     Rdest,$0,Rsrc
skip:
```

The `bgez` instruction actually uses an offset of 8 to affect the jump as shown below:

```
bgez    Rsrc,8
```

We discuss the branch instruction on page 630.

Multiply Instructions

Two multiply instructions are available: for signed numbers (`mult`) and for unsigned numbers (`multu`). The instruction

```
mult   Rsrc1,Rsrc2
```

multiplies contents of `Rsrc1` with the contents of `Rsrc2`. The numbers are treated as signed numbers. The 64-bit result is placed in two special registers LO and HI. The LO register receives the lower-order word and the higher-order word is placed in the HI register. No integer overflow exception is generated. The `multu` instruction has the same format but treats the source operands as unsigned numbers.

There are instructions to move data between these special LO/HI registers and general-purpose registers. The instruction `mfhi` (move from HI)

```
mfhi   Rdest
```

moves the contents of the HI register to the general register `Rdest`. The `mflo` instruction is used to move data from the LO register. For movement of data into these special registers, `mthi` (move to HI) or `mtlo` (move to LO) is used.

The assembler multiply pseudoinstruction can be used to place the result directly in a destination register. A limitation of the pseudoinstruction is that it stores only the 32-bit result, not the 64-bit value. Note that multiplication of two 32-bit numbers can produce a 64-bit result. We can use these pseudoinstructions if we know that the result can fit in 32 bits. The instruction

```
mul†    Rdest,Rsrc1,Src2
```

places the 32-bit result of the product of `Rsrc1` and `Src2` in `Rdest`. `Src2` can be a register or an immediate value. This instruction does not generate an overflow exception. If an overflow exception is required, the `mulo` instruction is used. Both these instructions treat the numbers as signed. To multiply two unsigned numbers, we can use the `mulou` instruction (multiply with overflow unsigned).

The `mul` pseudoinstruction is translated as

```
mult    Rsrc1,Src2
mflo    Rdest
```

when `Src2` is a register. If `Src2` is an immediate value, it uses an additional `ori` instruction. For example, the pseudoinstruction

```
mul    $a0,$a1, 32
```

is translated into

```
ori     $1,$0,32
mult    $5,$1
mflo    $4
```

Remember that `a0` maps to `$4`, `a1` to `$5`, and `at` to `$1`. This example shows how the assembler uses the `at` register to translate pseudoinstructions.

Divide Instructions

As with the multiply instructions, we can use `div` and `divu` instructions to divide signed and unsigned numbers, respectively. The instruction

```
div    Rsrc1,Rsrc2
```

divides the contents of `Rsrc1` by the contents of `Rsrc2` (i.e., `Rsrc1`/`Rsrc2`). The contents of both source registers are treated as signed numbers. The result of the division is placed in LO and HI registers. The LO register receives the quotient and the HI register receives the remainder. No integer overflow exception is generated.

The result of the operation is undefined if the divisor is zero. Thus, checks for a zero divisor should precede this instruction.

The assembler provides three-operand divide pseudoinstructions similar to the multiply instructions. The instruction

```
div†    Rdest,Rsrc1,Src2
```

places the quotient of two signed number divisions `Rsrc1/Src2` in `Rdest`. As in the other instructions, `Src2` can be a register or an immediate value. For unsigned numbers, the `divu`† pseudoinstruction is used. The quotient is rounded toward zero. Overflow is signaled when dividing -2^{31} by -1 as the quotient is greater than $2^{31} - 1$. The assembler generates the real `div` instruction if we use

```
div    $0,Rsrc1,Src2.
```

To get the remainder instead of the quotient, use

```
rem†    Rdest,Rsrc1,Src2
```

for signed numbers.

Table 15.5 MIPS logical instructions

Instruction		Description
`and`	`Rdest,Rsrc1,Rsrc2`	Bit-wise AND of `Rsrc1` and `Rsrc2` is stored in `Rdest`.
`andi`	`Rdest,Rsrc1,imm16`	Bit-wise AND of `Rsrc1` and 16-bit `imm16` is stored in `Rdest`. The 16-bit `imm16` is zero-extended.
`or`	`Rdest,Rsrc1,Rsrc2`	Bit-wise OR of `Rsrc1` and `Rsrc2` is stored in `Rdest`.
`ori`	`Rdest,Rsrc1,imm16`	Bit-wise OR of `Rsrc1` and 16-bit `imm16` is stored in `Rdest`. The 16-bit `imm16` is zero-extended.
`not`†	`Rdest,Rsrc`	Bit-wise NOT of `Rsrc` is stored in `Rdest`.
`xor`	`Rdest,Rsrc1,Rsrc2`	Bit-wise XOR of `Rsrc1` and `Rsrc2` is stored in `Rdest`.
`xori`	`Rdest,Rsrc1,imm16`	Bit-wise XOR of `Rsrc1` and 16-bit `imm16` is stored in `Rdest`. The 16-bit `imm16` is zero-extended.
`nor`	`Rdest,Rsrc1,Rsrc2`	Bit-wise NOR of `Rsrc1` and `Rsrc2` is stored in `Rdest`.

15.2.4 Logical Instructions

The MIPS supports the logical instructions `and`, `or`, `nor`, and `xor` (exclusive-OR). The missing `not` operation is supported by a pseudoinstruction. All operations, except `not`, take two source operands and a destination operand. The `not` instruction takes one source and one destination operand. As with most instructions, all operands must be registers. However, `and`, `or`, and `xor` instructions can take one immediate operand.

A summary of the logical instructions is given in Table 15.5. Assembler pseudoinstructions use the same mnemonics for the logical operations AND, OR, and XOR but allow the second source operand to be either a register or a 16-bit immediate value.

The `not` pseudoinstruction can be implemented by the `nor` instruction as

```
nor    Rdest,Rsrc,$0
```

15.2.5 Shift Instructions

Both left-shift and right-shift instructions are available to facilitate bit operations. The number of bit positions to be shifted (i.e., shift count) can be specified as an immediate 5-bit value or via a register. If a register is used, only the least significant 5 bits are used as the shift count.

The basic left-shift instruction `sll` (shift left logical)

```
sll    Rdest,Rsrc,count
```

shifts the contents of Rsrc left by count bit positions and stores the result in Rdest. When shifting left, vacated bits on the right are filled with zeros. These are called logical shifts. We show arithmetic shifts when dealing with right-shifts.

The sllv (shift left logical variable) instruction

```
sllv    Rdest,Rsrc1,Rsrc2
```

is similar to the sll instruction except that the shift count is in the Rsrc2 register.

There are two types of right shift operations: logical or arithmetic. This is because, when we shift right, we have the option of filling the vacated left bits by zeros (called logical shift) or copying the sign bit (arithmetic shift). The difference between the logical and arithmetic shifts is explained in detail on page 357.

The logical right-shift instructions—shift right logical (srl) and shift right logical variable (srlv)—have a format similar to their left-shift cousins. As mentioned, the vacated bits on the left are filled with zeros.

The arithmetic shift right instructions follow a similar format; however, shifted bit positions on the left are filled with the sign bit (i.e., sign extended). The shift instructions are summarized in Table 15.6.

15.2.6 Rotate Instructions

A problem with the shift instructions is that the shifted-out bits are lost. Rotate instructions allow us to capture these bits. The processor does not support rotate instructions. However, the assembler provides two rotate pseudoinstructions: rotate left (rol) and rotate right (ror).

In rotate left, the bits shifted out at the left (i.e., sign-bit side) are inserted on the right-hand side. In rotate right, bits falling off the right side are inserted on the sign-bit side.

Table 15.7 summarizes the two rotate instructions, which are pseudoinstructions. For example, the rotate instruction

```
ror†    $t2,$t2,1
```

is translated as

```
sll     $1,$10,31
srl     $10,$10,1
or      $10,$10,$1
```

15.2.7 Comparison Instructions

Several comparison pseudoinstructions are available. The instruction slt (set on less than)

```
slt†    Rdest,Rsrc1,Rsrc2
```

sets Rdest to one if the contents of Rsrc1 are less than the contents of Rsrc2; otherwise, Rdest is set to zero. This instruction treats the contents of Rsrc1 and Rsrc2 as signed

Table 15.6 MIPS shift instructions

Instruction		Description
`sll`	`Rdest,Rsrc,count`	Left-shifts `Rsrc` by `count` bit positions and stores the result in `Rdest`. Vacated bits are filled with zeros. `count` is an immediate value between 0 and 31. If `count` is outside this range, it uses `count MOD 32` as the number of bit positions to be shifted (i.e., takes only the least significant five bits of `count`).
`sllv`	`Rdest,Rsrc1,Rsrc2`	Similar to `sll` except that the count is taken from the least significant five bits of `Rsrc2`.
`srl`	`Rdest,Rsrc,count`	Right-shifts `Rsrc` by `count` bit positions and stores the result in `Rdest`. This is a logical right-shift (i.e., vacated bits are filled with zeros). `count` is an immediate value between 0 and 31.
`srlv`	`Rdest,Rsrc1,Rsrc2`	Similar to `srl` except that the count is taken from the least significant five bits of `Rsrc2`.
`sra`	`Rdest,Rsrc,count`	Right-shifts `Rsrc` by `count` bit positions and stores the result in `Rdest`. This is an arithmetic right-shift (i.e., vacated bits are filled with the sign bit). `count` is an immediate value between 0 and 31.
`srav`	`Rdest,Rsrc1,Rsrc2`	Similar to `sra` except that the count is taken from the least significant five bits of `Rsrc2`.

Table 15.7 MIPS rotate instructions

Instruction		Description
`rol`†	`Rdest,Rsrc,Src2`	Rotates contents of `Rsrc` left by `Src2` bit positions and stores the result in `Rdest`. `Src2` can be a register or an immediate value. Bits shifted out on the left are inserted on the right-hand side. `Src2` should be a value between 0 and 31. If this value is outside this range, only the least significant five bits of `Src2` are used as in the shift instructions.
`ror`†	`Rdest,Rsrc,Src2`	Rotates contents of `Rsrc` right by `Src2` bit positions and stores the result in `Rdest`. Bits shifted out on the right are inserted on the left-hand side. `Src2` operand is similar to that in the `rol` instruction.

numbers. To test for the "less than" relationship, `slt` subtracts contents of `Rsrc2` from the contents of `Rsrc1`.

The second operand can be a 16-bit immediate value. In this case, use `slti` (set on less than immediate) as shown below:

```
slti†     Rdest,Rsrc1,imm
```

For unsigned numbers, use `sltu` for the register version and `sltiu` for the immediate-operand version.

As a convenience, the assembler allows us to use `slt` and `sltu` for both register and immediate-operand versions. In addition, the assembler provides more comparison instructions. These pseudoinstructions can be used to test for equal, not equal, greater than, greater than or equal, and less than or equal relationships. Table 15.8 summarizes the comparison instructions provided by the assembler.

All comparison instructions in Table 15.8 are pseudoinstructions. For example, the instruction

```
seq    $a0,$a1,$a2
```

is translated as

```
      beq    $6,$5,skip1
      ori    $4,$0,0
      beq    $0,$0,skip2
skip1:
      ori    $4,$0,1
skip2:
      . . .
```

Note that `$a0`, `$a1`, and `$a2` represent registers `$4`, `$5`, and `$6`, respectively. Branch instructions are discussed next.

15.2.8 Branch and Jump Instructions

Conditional execution is implemented by jump and branch instructions. We first look at the jump instructions. The basic jump instruction

```
j       target
```

transfers control to the `target` address. There are other jump instructions that are useful in procedure calls. These instructions are discussed in Section 15.6.

Branch instructions provide a more flexible test and jump execution. The MIPS supports several branch instructions. We describe some of these instructions in this section.

The unconditional branch instruction

```
b†     target
```

Table 15.8 MIPS comparison instructions

Instruction	Description
`seq`† `Rdest,Rsrc1,Src2`	`Rdest` is set to one if contents of `Rsrc1` and `Src2` are equal; otherwise, `Rdest` is set to zero.
`sgt`† `Rdest,Rsrc1,Src2`	`Rdest` is set to one if contents of `Rsrc1` are greater than Src2; otherwise, `Rdest` is set to zero.
`sgtu`† `Rdest,Rsrc1,Src2`	Same as `sgt` except that the source operands are treated as unsigned numbers.
`sge`† `Rdest,Rsrc1,Src2`	`Rdest` is set to one if contents of `Rsrc1` are greater than or equal to Src2; otherwise, `Rdest` is set to zero.
`sgeu`† `Rdest,Rsrc1,Src2`	Same as `sge` except that the source operands are treated as unsigned numbers.
`slt`† `Rdest,Rsrc1,Src2`	`Rdest` is set to one if contents of `Rsrc1` are less than Src2; otherwise, `Rdest` is set to zero.
`sltu`† `Rdest,Rsrc1,Src2`	Same as `slt` except that the source operands are treated as unsigned numbers.
`sle`† `Rdest,Rsrc1,Src2`	`Rdest` is set to one if contents of `Rsrc1` are less than or equal to Src2; otherwise, `Rdest` is set to zero.
`sleu`† `Rdest,Rsrc1,Src2`	Same as `sle` except that the source operands are treated as unsigned numbers.
`sne`† `Rdest,Rsrc1,Src2`	`Rdest` is set to one if contents of `Rsrc1` and `Src2` are not equal; otherwise, `Rdest` is set to zero.

transfers control to `target` unconditionally. Semantically, it is very similar to the jump instruction. The main difference is that the `b` instruction uses a 16-bit relative address whereas the `j` instruction uses a 26-bit absolute address. Thus, the jump instruction has a larger range than the branch instruction. But the branch is more convenient because it uses relative address.

Next we look at the conditional branch instructions. The branch instruction

```
beq    Rsrc1,Rsrc2,target
```

compares the contents of `Rsrc1` and `Rsrc2` and transfers control to `target` if they are equal.

To compare with zero, we can use the `beqz` instruction. The format is

```
beqz   Rsrc,target
```

This instruction transfers control to `target` if the value of `Rsrc` is equal to zero.

As noted, `b` is a pseudoinstruction that implemented as

```
bgez    $0,target
```

where `target` is the relative offset.

Branch instructions to test "less than" and "greater than" are also supported. As an example, the instruction

```
bgt    Rsrc1,Rsrc2,target
```

branches to the `target` location when the contents of `Rsrc1` are greater than `Rsrc2`. When comparing, the contents of `Rsrc1` and `Rsrc2` are treated as signed numbers. For unsigned numbers, we have to use the `bgtu` instruction.

Branch instructions to test combinations such as "greater than or equal to" are also available. Table 15.9 summarizes some of the branch instructions provided by the MIPS assembler.

15.3 SPIM System Calls

The SPIM provides I/O support through the system call (`syscall`) instruction. Eight of these calls facilitate input and output of the four basic data types: string, integer, float, and double. A notable service missing in this list is the character input and output. For character I/O, we have to use the string system calls.

To invoke a service, the system call service code should be placed in the `$v0` register. Any required arguments are placed in the `$a0` and `$a1` registers (use `$f12` for floating-point values). Any value returned by a system call is placed in `$v0` (`$f0` for floating-point values).

All 10 system calls are summarized in Table 15.10. The first 3 calls are self-explanatory. The `print_string` system call takes a pointer to a NULL-terminated string and prints the string. The `read_int`, `read_float`, and `read_double` system calls read input up to and including newline. Characters following the number are ignored. The `read_string` call takes the pointer to a buffer where the input string is to be placed and the buffer size n in `$a1`. The buffer size should be expressed in bytes. It reads at most $n-1$ characters into the buffer and terminates the string by the NULL character. The `read_string` call has the same semantics as the `fgets` function in the C library.

The `sbrk` call returns a pointer to a block of memory containing n additional bytes. The final system call `exit` stops execution of a program.

Here is an example code fragment that prompts the user for a name and reads the name:

```
        .DATA
prompt:
        .ASCIIZ   "Enter your name: "
in_name:
        .SPACE    31
        .TEXT
            . . .
        la    $a0,prompt     ; prompt user
        li    $v0,4
        syscall
```

Table 15.9 MIPS branch instructions

Instruction	Description
`b`[†] `target`	Branches unconditionally to `target`.
`beq Rsrc1,Rsrc2,target`	Branches to `target` if the contents of `Rsrc1` and `Rsrc2` are equal.
`bne Rsrc1,Rsrc2,target`	Branches to `target` if the contents of `Rsrc1` and `Rsrc2` are not equal.
`blt Rsrc1,Rsrc2,target`	Branches to `target` if the value of `Rsrc1` is less than the value of `Rsrc2`. The source operands are considered as signed numbers.
`bltu Rsrc1,Rsrc2,target`	Same as `blt` except that the source operands are treated as unsigned numbers.
`bgt Rsrc1,Rsrc2,target`	Branches to `target` if the value of `Rsrc1` is greater than the value of `Rsrc2`. The source operands are treated as signed numbers.
`bgtu Rsrc1,Rsrc2,target`	Same as `bgt` except that the source operands are treated as unsigned numbers.
`ble Rsrc1,Rsrc2,target`	Branches to `target` if the value of `Rsrc1` is less than or equal to the value of `Rsrc2`. The source operands are treated as signed numbers.
`bleu Rsrc1,Rsrc2,target`	Same as `ble` except that the source operands are treated as unsigned numbers.
`bge Rsrc1,Rsrc2,target`	Branches to `target` if the value of `Rsrc1` is greater than or equal to the value of `Rsrc2`. The source operands are treated as signed numbers.
`bgeu Rsrc1,Rsrc2,target`	Same as `bge` except that the source operands are considered as unsigned numbers.
Comparison with zero	
`beqz Rsrc,target`	Branches to `target` if the value of `Rsrc` is equal to zero.
`bnez Rsrc,target`	Branches to `target` if the value of `Rsrc` is not equal to zero.
`bltz Rsrc,target`	Branches to `target` if the value of `Rsrc` is less than zero.
`bgtz Rsrc,target`	Branches to `target` if the value of `Rsrc` is greater than zero.
`blez Rsrc,target`	Branches to `target` if the value of `Rsrc` is less than or equal to zero.
`bgez Rsrc,target`	Branches to `target` if the value of `Rsrc` is greater than or equal to zero.

Table 15.10 SPIM assembler system calls

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_sting	4	$a0 = string address	
read_int	5		Integer in $v0
read_float	6		Float in $f0
read_double	7		Double in $f0
read_string	8	$a0 = buffer address $a1 = buffer size	
sbrk	9		Address in $v0
exit	10		

```
la    $a0,in_name    ; read name
li    $a1,31
li    $v0,8
syscall
```

15.4 SPIM Assembler Directives

The SPIM supports a subset of the assembler directives provided by the MIPS assembler. This section presents some of the most common SPIM directives. The SPIM reference manual [24] provides a complete list of directives supported by the simulator. All assembler directives begin with a period.

Segment Declaration

Two segments of an assembly program—code and data—can be declared by using `.TEXT` and `.DATA` directives. The statement

```
.TEXT  <address>
```

directs the assembler to map the following statements to the user text segment. The argument `address` is optional; if present, the statements are stored beginning at `address`. The SPIM allows only instructions or words (using `.WORD`) in the text segment.

The data directive has a similar format to `.TEXT` except that the statement following it must refer to data items, as shown in the last example code fragment.

String Directives

The SPIM provides two directives to allocate storage for strings: `.ASCII` and `.ASCIIZ`. The `.ASCII` directive can be used to allocate space for a string that is not terminated by the NULL character. The statement

```
.ASCII  string
```

allocates a number of bytes equal to the number of characters in `string`. For example,

```
.ASCII  "Toy Story"
```

allocates nine bytes of contiguous storage and initializes it to "Toy Story".

Strings are normally NULL-terminated as in C. For example, to display a string using `print_string` service, the string must be NULL-terminated. Using `.ASCIIZ` instead of `ASCII` stores the specified string in the NULL-terminated format. The `.ASCII` directive is useful for breaking a long string into multiple string statements as shown in the following example:

```
.ASCII    "Toy Story is a good computer-animated movie. \n"
.ASCII    "This reviewer recommends it to all kids \n"
.ASCIIZ   "and their parents."
```

An associated assembler directive

```
.SPACE    n
```

can be used to allocate `n` bytes of uninitialized space in the current segment.

Data Directives

The SPIM provides four directives to store both integers and floating-point numbers. The assembler directive

```
.HALF    h1,h2, . . .,hn
```

stores the n 16-bit numbers in successive memory halfwords. For 32-bit numbers, the `.WORD` directive is used. Although we refer to these 16- and 32-bit values as numbers, they can be any 16- and 32-bit quantities.

Floating-point values can be stored as single-precision or double-precision numbers. To store n single-precision floating-point numbers, use

```
.FLOAT    f1,f2, . . .,fn
```

To store double-precision numbers, use the `.DOUBLE` directive instead.

Miscellaneous Directives

We discuss two directives that deal with data alignment and global symbol declaration. The data directives `.HALF`, `.WORD`, `.FLOAT`, and `.DOUBLE` automatically align the data. We can explicitly control data alignment using the `.ALIGN` directive. The statement

```
.ALIGN    n
```

aligns the next datum on a 2^n byte boundary. Use

```
.ALIGN    0
```

to turn off the automatic alignment feature of the data directives until the next `.DATA` directive.

We discuss one last directive, `.GLOBL`. It declares a symbol as global so that it can be referenced from other files. We normally use this directive to declare `main` as a global symbol so that it can be referenced by the SPIM's trap file (see Appendix G for details on the trap file). In our programs, the code segment begins with this directive as shown below:

```
        .TEXT
        .GLOBL    main
main:
        . . .
```

15.5 Illustrative Examples

We use four example programs to illustrate the MIPS assembly language features. On purpose, we have selected the examples from Chapter 9 so that you can see the difference between the assembly languages of Pentium and MIPS processors. If you have gone through those examples in Section 9.9 starting on page 368, understanding the MIPS versions becomes easier, as the underlying algorithms are the same. These examples can be run on the MIPS simulator SPIM. Appendix G gives details about downloading and using the SPIM simulator.

Example 15.1 *Displays the ASCII value of the input character in binary representation.*
This is the MIPS version of the Pentium example discussed on page 369. It takes a character as input and displays its ASCII value in binary. Since the SPIM does not support character I/O, we use the string read system call to read the input character. We allocate two bytes of storage space to read a single character (`ch` on lines 21 and 22). For the same reason, we have to construct the output binary number as a character string. We use `ascii_string` on lines 23 and 24 for this purpose.

The conversion to binary can be done in several ways. The logic of the program follows the algorithm given in Example 9.6 on page 369. We use the `t2` register to hold the mask byte, which is initialized to 80H to test the most significant bit of the input character in `t0`. After testing the bit, the mask byte is shifted right by one bit position to test the next bit (line 58). We get the binary value after iterating eight times. We do not have a special counter to terminate the loop after eight iterations. Instead, we use the condition that mask byte will have zero when

shifted eight times. The statement on line 60 detects this condition and terminates the loop. Once the loop is exited, we just need to append a NULL character to the output string and display it.

Within the loop body, we use `and` and `beqz` to find if the tested bit is 0 or 1. We load characters 0 and 1 in registers `t4` and `t5` (lines 47 and 48) so that we can use `sb` on lines 53 and 56 to store the correct value. We have to resort to this because `sb` does not allow immediate values.

Program 15.1 Conversion of a ASCII value into binary representation

```
 1:  # Convert a character to ASCII                          BINCH.ASM
 2:  #
 3:  # Objective: To convert a character to its binary equivalent.
 4:  #            The character is read as a string.
 5:  #     Input: Requests a character from keyboard.
 6:  #    Output: Outputs the ASCII value.
 7:  #
 8:  #    t0 - holds the input character
 9:  #    t1 - points to output ASCII string
10:  #    t2 - holds the mask byte
11:  #
12:  ###################### Data segment ##########################
13:
14:        .data
15:  ch_prompt:
16:        .asciiz     "Please enter a character: \n"
17:  out_msg:
18:        .asciiz     "\nThe ASCII value is: "
19:  newline:
20:        .asciiz     "\n"
21:  ch:
22:        .space      2
23:  ascii_string:
24:        .space      9
25:
26:  ###################### Code segment ##########################
27:
28:        .text
29:        .globl main
30:  main:
31:        la    $a0,ch_prompt      # prompt user for a character
32:        li    $v0,4
33:        syscall
34:
35:        la    $a0,ch             # read the input character
```

```
36:         li    $a1,2
37:         li    $v0,8
38:         syscall
39:
40:         la    $a0,out_msg        # write output message
41:         li    $v0,4
42:         syscall
43:
44:         lb    $t0,ch             # t0 holds the character
45:         la    $t1,ascii_string   # t1 points to output string
46:         li    $t2,0x80           # t2 holds the mask byte
47:         li    $t4,'0'
48:         li    $t5,'1'
49:
50:   loop:
51:         and   $t3,$t0,$t2
52:         beqz  $t3,zero
53:         sb    $t5,($t1)           # store 1
54:         b     rotate
55:   zero:
56:         sb    $t4,($t1)           # store 0
57:   rotate:
58:         srl   $t2,$t2,1           # shift mask byte
59:         addu  $t1,$t1,1
60:         bnez  $t2,loop            # exit loop if mask byte is 0
61:
62:         sb    $0,($t1)            # append NULL
63:         la    $a0,ascii_string    # output ASCII value
64:         li    $v0,4
65:         syscall
66:
67:         la    $a0,newline         # output newline
68:         li    $v0,4
69:         syscall
```

Example 15.2 *Conversion of lowercase letters to uppercase.*

This example converts lowercase letters in a string to the corresponding uppercase letters. All other characters are not affected. We have done the Pentium version of this example on page 375, which presents the pseudocode that describes the program's logic.

The input string is limited to 30 characters as we allocate 31 bytes of space (see line 19). The program enforces this restriction as we use 31 as a parameter to the string input system call on line 31.

The loop terminates when a NULL character is encountered. Remember that the ASCII value for the NULL is zero. We use the `beqz` instruction on line 42 to detect the end of the string. We will go to line 45 only if the character is a lowercase letter. Since the SPIM assembler does not allow writing constants of the form `'A'-'a'`, we use −32 on line 45. The rest of the program is straightforward to follow.

Program 15.2 String conversion from lowercase to uppercase

```
 1:  # Uppercase conversion of characters    TOUPPER.ASM
 2:  #
 3:  # Objective: To convert lowercase letters to
 4:  #            corresponding uppercase letters.
 5:  #     Input: Requests a character string from keyboard.
 6:  #    Output: Prints the input string in uppercase.
 7:  #
 8:  # t0 - points to character string
 9:  # t1 - used for character conversion
10:  #
11:  ################### Data segment #####################
12:
13:        .data
14:  name_prompt:
15:        .asciiz     "Please type your name: \n"
16:  out_msg:
17:        .asciiz     "Your name in capitals is: "
18:  in_name:
19:        .space      31
20:
21:  ################### Code segment #####################
22:
23:        .text
24:        .globl main
25:  main:
26:        la    $a0,name_prompt  # prompt user for input
27:        li    $v0,4
28:        syscall
29:
30:        la    $a0,in_name      # read user input string
31:        li    $a1,31
32:        li    $v0,8
33:        syscall
34:
35:        la    $a0,out_msg      # write output message
36:        li    $v0,4
37:        syscall
```

```
38:
39:         la    $t0,in_name
40:   loop:
41:         lb    $t1,($t0)
42:         beqz  $t1,exit_loop    # if NULL, we are done
43:         blt   $t1,'a',no_change
44:         bgt   $t1,'z',no_change
45:         addu  $t1,$t1,-32      # convert to uppercase
46:                                # 'A'-'a' = -32
47:   no_change:
48:         sb    $t1,($t0)
49:         addu  $t0,$t0,1        # increment pointer
50:         j     loop
51:   exit_loop:
52:         la    $a0,in_name
53:         li    $v0,4
54:         syscall
```

Example 15.3 *Addition of individual digits of an integer—String version.*

We have done the Pentium version of this example on page 377. Here, we present two versions of this program. In the first version (this example), we read the input integer as a string. In the next example, we read the number as an integer. Both versions take an integer input and print the sum of the individual digits. For example, giving 12,345 as input produces 15 as the output.

We use `t0` to point to the digit that is to be processed. The running total is maintained in `t2`. We convert each digit in the input number into its decimal equivalent by stripping off the upper four bits. For example, character 5 is represented in ASCII as `00110101` when expressed in binary. To convert this to number 5, we force the upper four bits to zero. This is what the loop body (lines 42 to 50) does in the following program.

The loop iterates until it encounters either a NULL character (line 45) or a newline character (line 44). The reason for using two conditions, rather than just NULL-testing, is that the string input system call actually copies the newline character when the ENTER key is pressed. Thus, when the user enters less than 11 digits, a newline character is present in the string. On the other hand, when an 11-digit number is entered, we do not see the newline character. In this case, we have to use the NULL character to terminate the loop.

Program 15.3 Addition of individual digits: String version

```
1:  # Add individual digits of a number          ADDIGITS.ASM
2:  #
3:  # Objective: To add individual digits of an integer.
4:  #            The number is read as a string.
```

```
#     Input: Requests a number from keyboard.
#    Output: Outputs the sum.
#
#    t0 - points to character string (i.e., input number)
#    t1 - holds a digit for processing
#    t2 - maintains the running total
#
###################### Data segment ############################

        .data
number_prompt:
        .asciiz     "Please enter a number (<11 digits): \n"
out_msg:
        .asciiz     "The sum of individual digits is: "
number:
        .space      12

###################### Code segment ############################

        .text
        .globl main
main:
        la    $a0,number_prompt  # prompt user for input
        li    $v0,4
        syscall

        la    $a0,number         # read the input number
        li    $a1,12
        li    $v0,8
        syscall

        la    $a0,out_msg        # write output message
        li    $v0,4
        syscall

        la    $t0,number         # pointer to number
        li    $t2,0              # init sum to zero
loop:
        lb    $t1,($t0)
        beq   $t1,0xA,exit_loop  # if CR, we are done, or
        beqz  $t1,exit_loop      # if NULL, we are done
        and   $t1,$t1,0x0F       # strip off upper 4 bits
        addu  $t2,$t2,$t1        # add to running total

        addu  $t0,$t0,1          # increment pointer
```

```
50:         j     loop
51:   exit_loop:
52:         move  $a0,$t2            # output sum
53:         li    $v0,1
54:         syscall
```

Example 15.4 *Addition of individual digits of an integer—Number version.*
In this program, we read the input as an integer using the `read_int` system call (lines 30 and 31). Since the `read_int` call accepts signed integers, we convert any negative value to a positive integer by using the `abs` instruction on line 33.

To separate individual digits, we divide the number by 10. The remainder of this division gives us the rightmost digit. We repeat this process on the quotient of the division until the quotient is zero. For example, dividing the number 12,345 by 10 gives us 5 as the remainder and 1,234 as the quotient. Now dividing the quotient by 10 gives us 4 as the remainder and 123 as the quotient and so on. For this division, we use the unsigned divide instruction `divu` (line 42). This instruction places the remainder and quotient in HI and LO special registers. Special move instructions `mflo` and `mfhi` are used to copy these two values into `t0` and `t3` registers (lines 44 and 45). The loop terminates if the quotient (in `t0`) is zero.

Program 15.4 Conversion to upper case

```
 1:  # Add individual digits of a number     ADDIGITS2.ASM
 2:  #
 3:  # Objective: To add individual digits of an integer.
 4:  #            To demonstrate DIV instruction.
 5:  #     Input: Requests a number from keyboard.
 6:  #    Output: Outputs the sum.
 7:  #
 8:  #    t0 - holds the quotient
 9:  #    t1 - holds constant 10
10:  #    t2 - maintains the running sum
11:  #    t3 - holds the remainder
12:  #
13:  ################### Data segment #####################
14:
15:         .data
16:   number_prompt:
17:         .asciiz  "Please enter an integer: \n"
18:   out_msg:
19:         .asciiz  "The sum of individual digits is: "
20:
21:  ################### Code segment #####################
22:
```

```
23:         .text
24:         .globl main
25:   main:
26:         la    $a0,number_prompt  # prompt user for input
27:         li    $v0,4
28:         syscall
29:
30:         li    $v0,5              # read the input number
31:         syscall                  # input number in $v0
32:         move  $t0,$v0
33:         abs   $t0,$t0            # get absolute value
34:
35:         la    $a0,out_msg        # write output message
36:         li    $v0,4
37:         syscall
38:
39:         li    $t1,10             # $t1 holds divisor 10
40:         li    $t2,0              # init sum to zero
41:   loop:
42:         divu  $t0,$t1            # $t0/$t1
43:         # leaves quotient in LO and remainder in HI
44:         mflo  $t0                # move quotient to $t0
45:         mfhi  $t3                # move remainder to $t3
46:         addu  $t2,$t2,$t3        # add to running total
47:         beqz  $t0,exit_loop      # exit loop if quotient is 0
48:         j     loop
49:   exit_loop:
50:         move  $a0,$t2            # output sum
51:         li    $v0,1
52:         syscall
```

15.6 Procedures

The MIPS provides two instructions to support procedures: `jal` and `jr`. These correspond to the `call` and `ret` instructions of the Pentium. The `jal` (jump and link) instruction

```
jal    proc_name
```

transfers control to `proc_name` just as a jump instruction does. Since we need the return address, it also stores the address of the instruction following `jal` in the `ra` register.

To return from a procedure, use

```
jr     $ra
```

which reads the return address from the `ra` register and transfers control to this address.

In the Pentium, procedures require the stack. The `call` instruction places the return address on the stack, and the `ret` instruction retrieves it from the stack to return from a procedure. Thus, two memory accesses are involved to execute a procedure. In contrast, procedures in the MIPS can be implemented without using the stack. It uses the `ra` register for this purpose, which makes procedure invocation and return faster than in the Pentium. However, this advantage is lost when we use recursive or nested procedures. This point becomes clear later when we discuss recursive procedure examples in Section 15.7.1.

Parameter passing can be done via the registers or the stack. It is a good time to review our discussion of parameter passing mechanisms in Chapter 9. In the Pentium, register-based parameter passing is fairly restrictive due to the small number of registers available. However, the large number of registers in the MIPS makes this method attractive. We now use two examples to illustrate how procedures are written in the MIPS assembly language.

Example 15.5 *Finds minimum and maximum of three numbers.*
This is a simple program to explain the basics of procedures in the MIPS assembly language. The main program requests three integers and passes them to two procedures: `find_min` and `find_max`. Each procedure returns a value: minimum or maximum. Registers are used for parameter passing as well as to return the result. Registers `a1`, `a2`, and `a3` are used to pass the three integers. Each procedure returns its result in `v0`.

To invoke a procedure, we use `jal` as shown on lines 42 and 45. The body of these procedures is simple and straightforward to understand. When the procedure is done, a `jr` instruction returns control to the main program (see lines 83 and 97).

Program 15.5 A simple procedure example

```
 1:  # Find min and max of three numbers         MIN_MAX.ASM
 2:  #
 3:  # Objective: Finds min and max of three integers.
 4:  #            To demonstrate register-based parameter passing.
 5:  #     Input: Requests three numbers from keyboard.
 6:  #    Output: Outputs the minimum and maximum.
 7:  #
 8:  #    a1, a2, a3 - three numbers are passed via these registers
 9:  #
10:  ######################## Data segment ############################
11:        .data
12:  prompt:
13:        .asciiz     "Please enter three numbers: \n"
14:  min_msg:
15:        .asciiz     "The minimum is: "
16:  max_msg:
17:        .asciiz     "\nThe maximum is: "
18:  newline:
19:        .asciiz     "\n"
```

```
20:
21:  ###################### Code segment ##########################
22:
23:        .text
24:        .globl main
25:  main:
26:        la    $a0,prompt          # prompt user for input
27:        li    $v0,4
28:        syscall
29:
30:        li    $v0,5               # read the first number into $a1
31:        syscall
32:        move  $a1,$v0
33:
34:        li    $v0,5               # read the second number into $a2
35:        syscall
36:        move  $a2,$v0
37:
38:        li    $v0,5               # read the third number into $a3
39:        syscall
40:        move  $a3,$v0
41:
42:        jal   find_min
43:        move  $s0,$v0
44:
45:        jal   find_max
46:        move  $s1,$v0
47:
48:        la    $a0,min_msg         # write minimum message
49:        li    $v0,4
50:        syscall
51:
52:        move  $a0,$s0             # output minimum
53:        li    $v0,1
54:        syscall
55:
56:        la    $a0,max_msg         # write maximum message
57:        li    $v0,4
58:        syscall
59:
60:        move  $a0,$s1             # output maximum
61:        li    $v0,1
62:        syscall
63:
64:        la    $a0,newline         # write newline
```

```
65:         li    $v0,4
66:         syscall
67:
68:         li    $v0,10              # exit
69:         syscall
70:
71:   #-------------------------------------------------------------
72:   # FIND_MIN receives three integers in $a0, $a1, and $a2 and
73:   # returns the minimum of the three in $v0
74:   #-------------------------------------------------------------
75:   find_min:
76:         move  $v0,$a1
77:         ble   $v0,$a2,min_skip_a2
78:         move  $v0,$a2
79:   min_skip_a2:
80:         ble   $v0,$a3,min_skip_a3
81:         move  $v0,$a3
82:   min_skip_a3:
83:         jr    $ra
84:
85:   #-------------------------------------------------------------
86:   # FIND_MAX receives three integers in $a0, $a1, and $a2 and
87:   # returns the maximum of the three in $v0
88:   #-------------------------------------------------------------
89:   find_max:
90:         move  $v0,$a1
91:         bge   $v0,$a2,max_skip_a2
92:         move  $v0,$a2
93:   max_skip_a2:
94:         bge   $v0,$a3,max_skip_a3
95:         move  $v0,$a3
96:   max_skip_a3:
97:         jr    $ra
```

Example 15.6 *Finds string length.*

The previous example used the call-by-value mechanism to pass parameters via the registers. In this example, we use the call-by-reference method to pass a string pointer via `a0` (line 36). The procedure finds the length of the string and returns it via the `v0` register.

The string length procedure scans the string until it encounters either a newline or a NULL character (for the same reasons discussed in Example 15.3). These two termination conditions are detected on lines 63 and 64.

Program 15.6 String length example

```
 1:  # Finds string length                              STR_LEN.ASM
 2:  #
 3:  # Objective: Finds length of a string.
 4:  #            To demonstrate register-based pointer passing.
 5:  #     Input: Requests a string from keyboard.
 6:  #    Output: Outputs the string length.
 7:  #
 8:  #    a0 - string pointer
 9:  #    v0 - procedure returns string length
10:  #
11:  ###################### Data segment ##########################
12:        .data
13:  prompt:
14:        .asciiz     "Please enter a string: \n"
15:  out_msg:
16:        .asciiz     "\nString length is: "
17:  newline:
18:        .asciiz     "\n"
19:  in_string:
20:        .space      31
21:
22:  ###################### Code segment ##########################
23:
24:        .text
25:        .globl main
26:  main:
27:        la    $a0,prompt         # prompt user for input
28:        li    $v0,4
29:        syscall
30:
31:        la    $a0,in_string      # read input string
32:        li    $a1,31             # buffer length in $a1
33:        li    $v0,8
34:        syscall
35:
36:        la    $a0,in_string      # call string length proc.
37:        jal   string_len
38:        move  $t0,$v0            # string length in $v0
39:
40:        la    $a0,out_msg        # write output message
41:        li    $v0,4
42:        syscall
43:
44:        move  $a0,$t0            # output string length
```

```
45:         li    $v0,1
46:         syscall
47:
48:         la    $a0,newline         # write newline
49:         li    $v0,4
50:         syscall
51:
52:         li    $v0,10              # exit
53:         syscall
54:
55:   #-------------------------------------------------------------
56:   # STRING_LEN receives a pointer to a string in $a0 and
57:   # returns the string length in $v0
58:   #-------------------------------------------------------------
59:   string_len:
60:         li    $v0,0               # init $v0 (string length)
61:   loop:
62:         lb    $t0,($a0)
63:         beq   $t0,0xA,done        # if CR
64:         beqz  $t0,done            # or NULL, we are done
65:         addu  $a0,$a0,1
66:         addu  $v0,$v0,1
67:         b     loop
68:   done:
69:         jr    $ra
```

15.7 Stack Implementation

The MIPS does not explicitly support stack operations. In contrast, recall that the Pentium provides instructions such as `push` and `pop` to facilitate stack operations. In addition, there is a special stack pointer register `sp` that keeps the top-of-stack information. In the MIPS, a register plays the role of the stack pointer. We have to manipulate this register to implement the stack.

The MIPS stack implementation has some similarities to the Pentium implementation. For example, the stack grows downward (i.e., as we push items onto the stack, the address decreases). Thus, when reserving space on the stack for pushing values, we have to decrease the `sp` value. For example, to push registers `a0` and `ra`, we have to reserve eight bytes of stack space and use `sw` to push the values as shown below:

```
sub     $sp,$sp,8       # reserve 8 bytes of stack
sw      $a0,0($sp)      # save registers
sw      $ra,4($sp)
```

This sequence is typically used at the beginning of a procedure to save registers. To restore these registers before returning from the procedure, we can use the following sequence:

```
        lw      $a0,0($sp)      # restore the two registers
        lw      $ra,4($sp)
        addu    $sp,$sp,8       # clear 8 bytes of stack
```

15.7.1 Illustrative Examples

We give three examples that use the stack for parameter passing. Of the three, the last two examples show how recursion can be implemented in the MIPS assembly language.

Example 15.7 *Passing variable parameters via the stack.*
We use the variable parameter example discussed on page 417 to illustrate how the stack can be used to pass parameters. The procedure `sum` receives a variable number of integers via the stack. The parameter count is passed via `a0`. The main program reads a sequence of integers from the input. Entering zero terminates the input. Each number read from the input is directly placed on the stack (lines 36 and 37). Since `sp` always points to the last item pushed onto the stack, we can pass this value to the procedure. Thus, a simple procedure call (line 41) is sufficient to pass the parameter count and the actual values.

The procedure sum reads the numbers from the stack. As it reads, it decreases the stack size (i.e., `sp` increases). The loop in the `sum` procedure terminates when `a0` is zero (line 66). When the loop is exited, the stack is also cleared of all the arguments.

Compared to the Pentium version, the MIPS allows a more flexible access to parameters. In the Pentium, because the return address is pushed onto the stack, we had to use the `BP` register to access the parameters. In addition, we could not remove the numbers from the stack. The stack had to be cleared in the main program by manipulating the `SP` register.

Program 15.7 Passing variable number of parameters to a procedure

```
 1:  # Sum of variable number of integers         VAR_PARA.ASM
 2:  #
 3:  # Objective: Finds sum of variable number of integers.
 4:  #            Stack is used to pass variable number of integers.
 5:  #            To demonstrate stack-based parameter passing.
 6:  #     Input: Requests integers from the user;
 7:  #            terminated by entering a zero.
 8:  #    Output: Outputs the sum of input numbers.
 9:  #
10:  #    a0 - number of integers passed via the stack
11:  #
12:  ######################## Data segment ############################
13:        .data
14:  prompt:
15:        .ascii     "Please enter integers. \n"
16:        .asciiz    "Entering zero terminates the input. \n"
17:  sum_msg:
```

```
18:         .asciiz     "The sum is: "
19:   newline:
20:         .asciiz     "\n"
21:
22:   ###################### Code segment ##########################
23:
24:         .text
25:         .globl main
26:   main:
27:         la    $a0,prompt          # prompt user for input
28:         li    $v0,4
29:         syscall
30:
31:         li    $a0,0
32:   read_more:
33:         li    $v0,5               # read a number
34:         syscall
35:         beqz  $v0,exit_read
36:         subu  $sp,$sp,4           # reserve 4 bytes on stack
37:         sw    $v0,($sp)           # store the number on stack
38:         addu  $a0,$a0,1
39:         b     read_more
40:   exit_read:
41:         jal   sum                 # sum is returned in $v0
42:         move  $s0,$v0
43:
44:         la    $a0,sum_msg         # write output message
45:         li    $v0,4
46:         syscall
47:
48:         move  $a0,$s0             # output sum
49:         li    $v0,1
50:         syscall
51:
52:         la    $a0,newline         # write newline
53:         li    $v0,4
54:         syscall
55:
56:         li    $v0,10              # exit
57:         syscall
58:
59:   #-------------------------------------------------------------
60:   # SUM receives the number of integers passed in $a0 and the
61:   # actual numbers via the stack. It returns the sum in $v0.
62:   #-------------------------------------------------------------
```

```
63:  sum:
64:         li     $v0,0
65:  sum_loop:
66:         beqz   $a0,done
67:         lw     $t0,($sp)
68:         addu   $sp,$sp,4
69:         addu   $v0,$v0,$t0
70:         subu   $a0,$a0,1
71:         b      sum_loop
72:  done:
73:         jr     $ra
```

Recursion

The first two examples (Examples 15.5 and 15.6) showed how simple procedures can be written in the MIPS assembly language. We could write these procedures without using the stack. The last example used the stack to pass a variable number of parameters. For most normal procedures, we do not have to use the stack. The availability of a large number of registers allows us to use register-based parameter passing. However, when we write recursive procedures, we have to use the stack.

We introduced principles of recursion in Section 11.5 (see page 455). In that section, we presented two example Pentium programs: factorial and quicksort. Now, we do those examples in the MIPS assembly language to illustrate how recursion is implemented in the MIPS.

Example 15.8 *A recursion example—Factorial.*
Recall that the factorial function is defined as

$$\begin{aligned} 0! &= 1, \\ N! &= N * (N-1)!. \end{aligned}$$

Program 15.8 requests an integer N from the input and prints $N!$. This value is passed on to the factorial procedure (`fact`) via the `a0` register. First we have to determine the state information that needs to be saved (i.e., our activation record). In all procedures, we need to store the return address. In the Pentium, this is automatically done by the `call` instruction. In addition, in our factorial example, we need to keep track of the current value in `a0`. However, we don't have to save `a0` on the stack as we can restore its value by adding 1, as shown on line 76. Thus we save just the return address (line 67) and restore it on line 80. The body of the procedure can be divided into two parts: recursion termination and recursive call. Since 1! is also 1, we use this to terminate recursion (lines 69 to 71).

If the value is more than 1, a recursive call is made with $(N-1)$ (lines 74 and 75). After the call is returned, `a0` is incremented to make it N before multiplying it with the values returned for $(N-1)!$ in `v0` (lines 76 and 77).

Program 15.8 Computing factorial: An example recursive function

```
 1:  # Finds factorial of a number                    FACTORIAL.ASM
 2:  #
 3:  # Objective: Computes factorial of an integer.
 4:  #            To demonstrate recursive procedures.
 5:  #     Input: Requests an integer N from keyboard.
 6:  #    Output: Outputs N!
 7:  #
 8:  #    a0 - used to pass N
 9:  #    v0 - used to return result
10:  #
11:  ###################### Data segment ##########################
12:        .data
13:  prompt:
14:        .asciiz     "Please enter a positive integer: \n"
15:  out_msg:
16:        .asciiz     "The factorial is: "
17:  error_msg:
18:        .asciiz     "Sorry! Not a positive number.\nTry again.\n "
19:  newline:
20:        .asciiz     "\n"
21:
22:  ###################### Code segment ##########################
23:
24:        .text
25:        .globl main
26:  main:
27:        la    $a0,prompt         # prompt user for input
28:        li    $v0,4
29:        syscall
30:
31:  try_again:
32:        li    $v0,5              # read the input number into $a0
33:        syscall
34:        move  $a0,$v0
35:
36:        bgez  $a0,num_OK
37:        la    $a0,error_msg      # write error message
38:        li    $v0,4
39:        syscall
40:        b     try_again
41:
42:  num_OK:
43:        jal   fact
44:        move  $s0,$v0
```

```
45:
46:         la    $a0,out_msg        # write output message
47:         li    $v0,4
48:         syscall
49:
50:         move  $a0,$s0            # output factorial
51:         li    $v0,1
52:         syscall
53:
54:         la    $a0,newline        # write newline
55:         li    $v0,4
56:         syscall
57:
58:         li    $v0,10             # exit
59:         syscall
60:
61:   #-------------------------------------------------------------
62:   # FACT receives N in $a0 and returns the result in $v0
63:   # It uses recursion to find N!
64:   #-------------------------------------------------------------
65:   fact:
66:         subu  $sp,$sp,4          # allocate stack space
67:         sw    $ra,0($sp)         # save return address
68:
69:         bgt   $a0,1,one_up       # recursion termination
70:         li    $v0,1
71:         b     return
72:
73:   one_up:
74:         subu  $a0,$a0,1          # recurse with (N-1)
75:         jal   fact
76:         addu  $a0,$a0,1
77:         mulou $v0,$a0,$v0        # $v0 := $a0*$v0
78:
79:   return:
80:         lw    $ra,0($sp)         # restore return address
81:         addu  $sp,$sp,4          # clear stack space
82:         jr    $ra
```

Example 15.9 *A recursion example—Quicksort.*

As a second example, we implement the quicksort algorithm using recursion. A detailed description of the quicksort algorithm is given on page 458. Program 15.9 gives an implementation of the quicksort algorithm in the MIPS assembly language. The corresponding Pentium

assembly language implementation is given on page 460. One main difference between these two programs is the addressing modes used to access array elements. Since the MIPS does not support based-indexed addressing, the `qsort` procedure receives two pointers (as opposed to array indexes). Furthermore, the Pentium's `xchg` instruction comes in handy to exchange values between two registers.

The main program reads integers from input until terminated by a zero. We store the zero in the array, as we will use it as the sentinel to output the sorted array (see lines 55 and 56). Lines 43 to 46 prepare the two arguments for the `qsort` procedure.

The `qsort` recursive procedure stores `a3` in addition to the `a1`, `a2`, and `ra` registers. This is because we store the end-of-subarray pointer in `a3`, which is required for the second recursive call (line 121). As pointed out, due to lack of addressing mode support to access arrays, we have to use byte pointers to access individual elements. This means updating the index involves adding or subtracting 4 (see lines 94, 99, and 116). The rest of the procedure follows the quicksort algorithm described on page 459. You may find it interesting to compare this program with the Pentium version presented in Example 11.6 to see the similarities and differences between the two assembly languages in implementing recursion.

Program 15.9 Quicksort: Another example recursive program

```
 1:  # Sorting numbers using quicksort              QUICKSORT.ASM
 2:  #
 3:  # Objective: Sorts an array of integers using quicksort.
 4:  #            Uses recursion.
 5:  #     Input: Requests integers from the user;
 6:  #            terminated by entering a zero.
 7:  #    Output: Outputs the sorted integer array.
 8:  #
 9:  #    a0 - start of array
10:  #    a1 - beginning of (sub)array
11:  #    a2 - end of (sub)array
12:  #
13:  ###################### Data segment ##########################
14:        .data
15:  prompt:
16:        .ascii     "Please enter integers. \n"
17:        .asciiz    "Entering zero terminates the input. \n"
18:  output_msg:
19:        .asciiz     "The sorted array is: \n"
20:  newline:
21:        .asciiz     "\n"
22:  array:
23:        .word       200
24:
25:  ###################### Code segment ##########################
26:
```

```
27:         .text
28:         .globl main
29:   main:
30:         la    $a0,prompt          # prompt user for input
31:         li    $v0,4
32:         syscall
33:
34:         la    $t0,array
35:   read_more:
36:         li    $v0,5               # read a number
37:         syscall
38:         sw    $v0,($t0)           # store it in the array
39:         beqz  $v0,exit_read
40:         addu  $t0,$t0,4
41:         b     read_more
42:   exit_read:
43:         # prepare arguments for procedure call
44:         la    $a1,array           # a1 = lo pointer
45:         move  $a2,$t0
46:         subu  $a2,$a2,4           # a2 = hi pointer
47:         jal   qsort
48:
49:         la    $a0,output_msg      # write output message
50:         li    $v0,4
51:         syscall
52:
53:         la    $t0,array
54:   write_more:
55:         lw    $a0,($t0)           # output sorted array
56:         beqz  $a0,exit_write
57:         li    $v0,1
58:         syscall
59:         la    $a0,newline         # write newline message
60:         li    $v0,4
61:         syscall
62:         addu  $t0,$t0,4
63:         b     write_more
64:   exit_write:
65:
66:         li    $v0,10              # exit
67:         syscall
68:
69:   #-------------------------------------------------------------
70:   # QSORT receives pointer to the start of (sub)array in a1 and
71:   # end of (sub)array in a2.
```

```
 72:  #-------------------------------------------------------------
 73:  qsort:
 74:        subu  $sp,$sp,16         # save registers
 75:        sw    $a1,0($sp)
 76:        sw    $a2,4($sp)
 77:        sw    $a3,8($sp)
 78:        sw    $ra,12($sp)
 79:
 80:        ble   $a2,$a1,done       # end recursion if hi <= lo
 81:
 82:        move  $t0,$a1
 83:        move  $t1,$a2
 84:
 85:        lw    $t5,($t1)          # t5 = xsep
 86:
 87:  lo_loop:                             #
 88:        lw    $t2,($t0)                #
 89:        bge   $t2,$t5,lo_loop_done     # LO while loop
 90:        addu  $t0,$t0,4                #
 91:        b     lo_loop                  #
 92:  lo_loop_done:
 93:
 94:        subu  $t1,$t1,4          # hi = hi-1
 95:  hi_loop:
 96:        ble   $t1,$t0,sep_done         #
 97:        lw    $t3,($t1)                #
 98:        blt   $t3,$t5,hi_loop_done     # HI while loop
 99:        subu  $t1,$t1,4                #
100:        b     hi_loop                  #
101:  hi_loop_done:
102:
103:        sw     $t2,($t1)               #
104:        sw     $t3,($t0)               # x[i]<=>x[j]
105:        b      lo_loop                 #
106:
107:  sep_done:
108:        move  $t1,$a2                  #
109:        lw    $t4,($t0)                #
110:        lw    $t5,($t1)                # x[i] <=>x[hi]
111:        sw    $t5,($t0)                #
112:        sw    $t4,($t1)                #
113:
114:        move  $a3,$a2            # save HI for the second call
115:        move  $a2,$t0                  #
116:        subu  $a2,$a2,4                # set hi as i-1
```

```
117:        jal   qsort
118:
119:        move  $a1,$a2                 #
120:        addu  $a1,$a1,8               # set lo as i+1
121:        move  $a2,$a3
122:        jal   qsort
123:  done:
124:        lw    $a1,0($sp)          # restore registers
125:        lw    $a2,4($sp)
126:        lw    $a3,8($sp)
127:        lw    $ra,12($sp)
128:        addu  $sp,$sp,16
129:
130:        jr    $ra
```

15.8 Summary

We have discussed the MIPS assembly language in detail. The MIPS is a RISC processor that uses the load/store architecture. Thus, only the load and store instructions can access memory. All other instructions expect their operands in registers. As a result, RISC processors provide many more registers than CISC processors such as the Pentium. The MIPS R2000 processor provides 32 general-purpose registers. In addition, two special registers, HI and LO, are used to store the output of multiply and divide instructions. In addition, a program counter serves the purpose of instruction pointer. All these registers are 32-bits wide.

The R2000 does not provide as many addressing modes as the Pentium does. As discussed in the last chapter, this is one of the differences between CISC and RISC processors. In fact, the MIPS processor supports only one addressing mode. However, the assembler augments this by a few other addressing modes.

The MIPS processor provides a reasonable number of instructions. The assembler supplements these instructions by several useful pseudoinstructions. Unlike the Pentium, all instructions take 32 bits to encode. The MIPS R2000 uses only three different types of instruction formats.

MIPS processor hardware does not impose many restrictions on how the registers are used. Except for a couple of registers, the programmer is fairly free to use these registers as she wishes. However, certain conventions have been developed to make programs portable.

We have used several examples to illustrate the features of the MIPS assembly language. We have done most of these examples in the Pentium assembly language. The idea in redoing the same example set was to bring out the differences between the two assembly languages. In particular, the quicksort example brings out some of the limitations of the MIPS assembly language.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Absolute address
- Activation record
- Addressing modes in MIPS
- Based-addressing mode
- Call-by-reference
- Call-by-value
- Indexed addressing mode
- Instruction format
- Load instructions
- Merge sort
- PC-relative address
- Procedure call
- Quicksort
- Recursion in MIPS
- SPIM assembler directives
- SPIM segments
- SPIM system calls

15.9 Exercises

15–1 In the Itanium, the general-purpose register `gr0` is hardwired to zero. As we have seen in this chapter, the MIPS also has a register (`$zero`) that is hardwired to zero. What is the reason for this?

15–2 What is the purpose of HI and LO registers?

15–3 What is the difference between registers `t0` to `t9` and `S0` to `S7`?

15–4 What is a typical use of registers `v0` and `v1`?

15–5 What is a typical use of registers `a0` to `a3`?

15–6 Describe the addressing modes supported by the MIPS processor.

15–7 What is the difference between a pseudoinstruction and an instruction?

15–8 What is the difference between `.ASCII` and `.ASCIIZ` assembler directives?

15–9 The Pentium provides instructions and registers to implement the stack. In MIPS processors, there is no such support for stack implementation. Describe how the stack is implemented in the MIPS.

15–10 Write a simple assembly language program to see how the branch (`b`) and jump (`j`) instructions are translated. Use the SPIM to do the translation. You can use the "Text Segment" display window of the SPIM to check the translation (see Appendix G for details on the SPIM windows).

15–11 Use the SPIM simulator to find how the following pseudoinstructions are translated:

```
(a) rol     $t1,$t1,3

(b) li      $t0,5
    rol     $t1,$t1,$t0

(c) mul     $t0,$v0,9

(d) div     $t0,$t0,5
```

```
(e) rem    $t0,$t0,5              (f) sle    $a0,$a1,$a2

(g) sge    $a0,$a1,$a2            (h) sgeu   $a0,$a1,$a2

(i) slt    $a0,$a1,$a2            (j) sltu   $a0,$a1,$a2

(k) move   $a0,$t0
```

15–12 Discuss the differences between the Pentium and the MIPS in passing a variable number of parameters to procedures.

15–13 In the Pentium, the register-based parameter passing mechanism is not pragmatic. Why does it make sense to use this method in MIPS processors?

15–14 Why are simple procedure calls and returns faster in the MIPS than in the Pentium?

15–15 Discuss the differences between the Pentium and the MIPS in accessing multidimensional arrays.

15.10 Programming Exercises

15–P1 Modify the `addigits.asm` program such that it accepts a string from the keyboard consisting of digit and nondigit characters. The program should display the sum of the digits present in the input string. All nondigit characters should be ignored. For example, if the input string is

```
ABC1?5wy76:~2
```

the output of the program should be

```
sum of individual digits is: 21
```

15–P2 Write an assembly language program to encrypt digits as shown below:

Input digit:	0 1 2 3 4 5 6 7 8 9;
Encrypted digit:	4 6 9 5 0 3 1 8 7 2.

Your program should accept a string consisting of digit and nondigit characters. The encrypted string should be displayed in which only the digits are affected. Then the user should be queried as to whether he wants to terminate the program. If the response is either "y" or "Y", you should terminate the program; otherwise, you should request another input string from the keyboard.

The encryption scheme given here has the property that when you encrypt an already encrypted string, you get back the original string. Use this property to verify your program.

15–P3 Write a program that reads an input number (given in decimal) between 0 and 65,535 and displays the hexadecimal equivalent. You can read the input using the `read_int` system call. As in the last exercise, you should query the user for program termination.

15–P4 Modify the above program to display the octal equivalent instead of the hexadecimal equivalent of the input number.

15–P5 Write a procedure to perform string reversal. The procedure `reverse` receives a pointer to a character string (terminated by a NULL character) and reverses the string. For example, if the original string is

```
slap
```

the reversed string should read

```
pals
```

The `main` procedure should request the string from the user. It should also display the reversed string as the output of the program.

15–P6 Write a procedure `locate` to locate a character in a given string. The procedure receives a pointer to a NULL-terminated character string and the character to be located. When the first occurrence of the character is located, its position is returned to `main`. If no match is found, a negative value is returned. The `main` procedure requests a character string and a character to be located and displays the position of the first occurrence of the character returned by the `locate` procedure. If there is no match, a message should be displayed to that effect.

15–P7 Write a procedure that receives a string (i.e., a string pointer is passed to the procedure) and removes all leading blank characters in the string. For example, if the input string is (␣ indicates a blank character)

␣␣␣␣␣Read␣␣my␣lips.

it will be modified by removing all leading blanks as

Read␣␣my␣lips.

15–P8 Write a procedure that receives a string (i.e., a string pointer is passed to the procedure) and removes all leading and duplicate blank characters in the string. For example, if the input string is (␣ indicates a blank character)

␣␣␣␣␣Read␣␣␣my␣␣␣␣␣lips.

it will be modified by removing all leading and duplicate blanks as

Read␣my␣lips.

15–P9 Redo the Programming Exercise 10–P10 on page 434.

15–P10 Write a complete assembly language program to read two matrices **A** and **B** and display the result matrix **C**, which is the sum of **A** and **B**. Note that the elements of **C** can be obtained as

$$\mathbf{C}[i, j] = \mathbf{A}[i, j] + \mathbf{B}[i, j].$$

Your program should consist of a main procedure that calls the `read_matrix` procedure twice to read data for **A** and **B**. It should then call the `matrix_add` procedure,

which receives pointers to **A**, **B**, **C**, and the size of the matrices. Note that both **A** and **B** should have the same size. The `main` procedure calls another procedure to display **C**.

15–P11 Write a procedure to perform matrix multiplication of matrices **A** and **B**. The procedure should receive pointers to the two input matrices **A** of size $l \times m$, **B** of size $m \times n$, the product matrix **C**, and values l, m, and n. Also, the data for the two matrices should be obtained from the user. Devise a suitable user interface to input these numbers.

15–P12 Redo the Programming Exercise 11–P11 on page 467.

15–P13 We have discussed merge sort in Programming Exercise 12–P19 (page 548). Write a MIPS assembly language program to implement the merge sort.

15–P14 Write a procedure `str_ncpy` to mimic the `strncpy` function provided by the C library. The function `str_ncpy` receives two strings, `string1` and `string2`, and a positive integer `num`. Of course, the procedure receives only the string pointers but not the actual strings. It should copy at most the first `num` characters of `string2` to `string1`.

15–P15 You have written a recursive procedure to compute Fibonacci numbers in Programming Exercise 11-P14 (page 468). Redo the exercise in the MIPS assembly language. Debug and test your program using the SPIM.

15–P16 In Exercise 11-P15 (page 469), we have presented details about the Ackermann's function. Write a MIPS assembly language procedure to implement this function using recursion. You should also write a simple main procedure to test your Ackermann's procedure. Debug and test your program using the SPIM.

15–P17 We have presented details about the Towers of Hanoi puzzle in Exercise 11-P16 on page 469. Write a recursive procedure in the MIPS assembly language to solve this puzzle. Also, write a simple main program to test your program.

Chapter 16

Memory System Design

Objectives

- To introduce memory design techniques using flip-flops and latches;
- To describe how memory units can be interfaced to the system bus;
- To present a method to design larger memory modules using standard memory chips;
- To discuss various ways of mapping memory modules to the memory address space;
- To explain the reasons for the adverse impact of unaligned data on performance;
- To give details on how interleaved memories are constructed.

In Chapter 4, we have seen how flip-flops and latches can be used to store a bit. This chapter builds on this foundation and explains how we can use these basic devices and build larger memory blocks and modules. We start off with a simple design process that can be used to design memory blocks using flip-flops and latches. In the following section, we present various ways of connecting the memory modules to the system bus. In particular, we describe tristate buffers that are extensively used as interface devices for connection to the system bus. The other techniques use either an open collector device or a multiplexer, both of which are useful only in certain special cases.

We also discuss how larger memories can be built using narrower memory chips. The design process is fairly intuitive. The basic technique involves using a two-dimensional array of memory chips. A characteristic of these designs is the use of chip select. Chip select input can be used to select or deselect a chip or a memory module. Chip select allows us to connect multiple devices to the system bus. Appropriate chip select signal generation facilitates communication among the entities connected to the system bus.

Chip select logic is also useful in mapping memory modules to memory address space. We present details about two ways of mapping a memory module to the address space. In Section 16.7, we explain the reasons why data alignment leads to improved performance. Before ending the chapter, we describe the structure of interleaved memories. We end the chapter with a summary.

16.1 Introduction

Flip-flops and latches provide the basic capability to store a bit of data. In Chapter 4, we have seen several types of flip-flops and latches. These devices can be replicated to build larger memory units. For example, we can place 16 JK flip-flops together to store a 16-bit word. All 16 flip-flops would have their clock inputs tied together to form a single common clock to write a 16-bit word.

There are two types of memories: read/write memory and read-only memory. As the name suggests, a read/write memory allows reading as well as writing. This type of memory is referred to as random access memory (RAM). Read-only memory, on the other hand, allows only reading of its contents. These are referred to as ROMs.

Several types of ROMs are available; each type requires a special kind of writing procedure. Writing into a ROM is called programming the ROM. ROMs can be factory programmed at the time of manufacture. These are suitable when the required quantity is large to amortize the overhead. There are user-programmable ROMs (PROMs). For example, the user can selectively blow electric fuses by sending a high current to the program contents of a PROM. There are also erasable PROMs (EPROMs) that can be programmed several times. Exposing them to an ultraviolet light for a few minutes can erase the contents of the entire EPROM. Electrically erasable PROMs (EEPROMs), on the other hand, allow the user to selectively erase contents. Note that, even though only read/write memories are called random access memories, both read/write memories and read-only memories are random access memories.

ROMs are typically used to store the boot program, as ROMs are nonvolatile memories. That is, ROMs do not require power to retain their contents. That's why they are useful to store the program that your processor can read before loading the operating system. The bulk of the system memory consists of RAM.

RAM can be grouped into two basic types: static and dynamic. Static RAM (SRAM) uses a latch or flip-flop as the basic cell to store a bit. Dynamic RAM (DRAM) uses a different technique. DRAM uses a tiny capacitor to store a bit. Since capacitors leak charge, DRAMs need periodic refreshing to retain their contents. Furthermore, reads are destructive as reading destroys the contents. This characteristic of DRAMs requires write-after-read to restore the contents after a read cycle. The main reason for the popularity of DRAMs is their price advantage over SRAMs. There are also additional advantages of using DRAMs including low power consumption, less heat, and high-density packaging. SRAMs, on the other hand, are expensive but faster than DRAMs. DRAMs are used for main memory, and SRAMs are used for cache memory.

Despite these technical differences, SRAMs and DRAMs as well as ROMs can be treated as simple building blocks to construct larger memory units. To illustrate the concepts, we focus on static RAMs. The techniques we discuss here can be easily extended to other types of memories.

16.2 A Simple Memory Block

In the digital logic chapters, we have discussed several basic building blocks such as D flip-flops, multiplexers, and decoders. We now describe how these digital circuits can be used to build a simple memory block.

16.2.1 Memory Design with D Flip-Flops

We begin our discussion with how one can build memories using D flip-flops. Recall that we use flip-flops for edge-triggered devices and latches for level-sensitive devices. The principle of constructing memory out of D flip-flops is simple. We use a two-dimensional array of D flip-flops with each row storing a word. The number of rows is equal to the number of words the memory should store. We refer to this as "horizontal" expansion to increase the word width and "vertical" expansion to increase the number of words.

In general, the number of columns and the number of rows is a power of two. We use the notation M $\times$ N memory to represent a memory that can store M words, where each word is N-bits long.

Figure 16.1 shows a 4 $\times$ 3 memory built with 12 D flip-flops organized as a 4 $\times$ 3 array. Since all flip-flops in a row are storing a word of data, each row of flip-flops has their clock signals tied together to form a single clock signal for each row. All flip-flops in a column receive input from the same input data line. For example, the rightmost column D inputs are connected to the input data DI0.

This memory requires two address lines to select one of the four words. The two address lines are decoded to select a specific row by using a 2-to-4 decoder. The low-active write signal ($\overline{\mathrm{WR}}$) is gated through an AND gate as shown in Figure 16.1. Depending on the address, only one of the four decoder output lines will be high, permitting the $\overline{\mathrm{WR}}$ signal to clock the selected row to write the 3-bit data present on DI0 to DI2 lines. Note that the decoder along with the four AND gates forms a demultiplexer that routes the $\overline{\mathrm{WR}}$ signal to the row selected by the address lines A1 and A0.

The design we have done so far allows us to write a 3-bit datum into the selected row. To complete the design, we have to find a way to read data from this memory. As each bit of data is supplied by one of the four D flip-flops in a column, we have to find a way to connect these four outputs to a single data out line. A natural choice for the job is a 4-to-1 multiplexer. The MUX selection inputs are connected to the address lines to allow appropriate data on the output lines DO0 through DO2. The final design is shown in Figure 16.1.

16.2.2 Problems with the Design

The memory design example tells us how we can put bit-level devices together to form a higher-level two-dimensional memory. This design, however, has several problems. To understand the kind of problems created by this design, let us look at how the memory unit is connected to the system bus. In Chapter 1, we presented a simplified view of a typical computer system (see Figure 1.5 on page 13). This figure clearly shows how the system bus connects the three main components: processor, memory, and I/O devices.

Focusing on the interactions between the processor and memory unit, we see that the data bus is bidirectional. That is, data input and output lines are not separate. Our memory design shown in Figure 16.1 uses separate data in and out lines. This is the first problem with the design. To be able to connect to the data bus, we should be able to connect the corresponding DI and DO lines (i.e., DI0 and DO0 should be tied together, DI1 and DO1 should be tied together, etc.). However, our design will not allow us to do this.

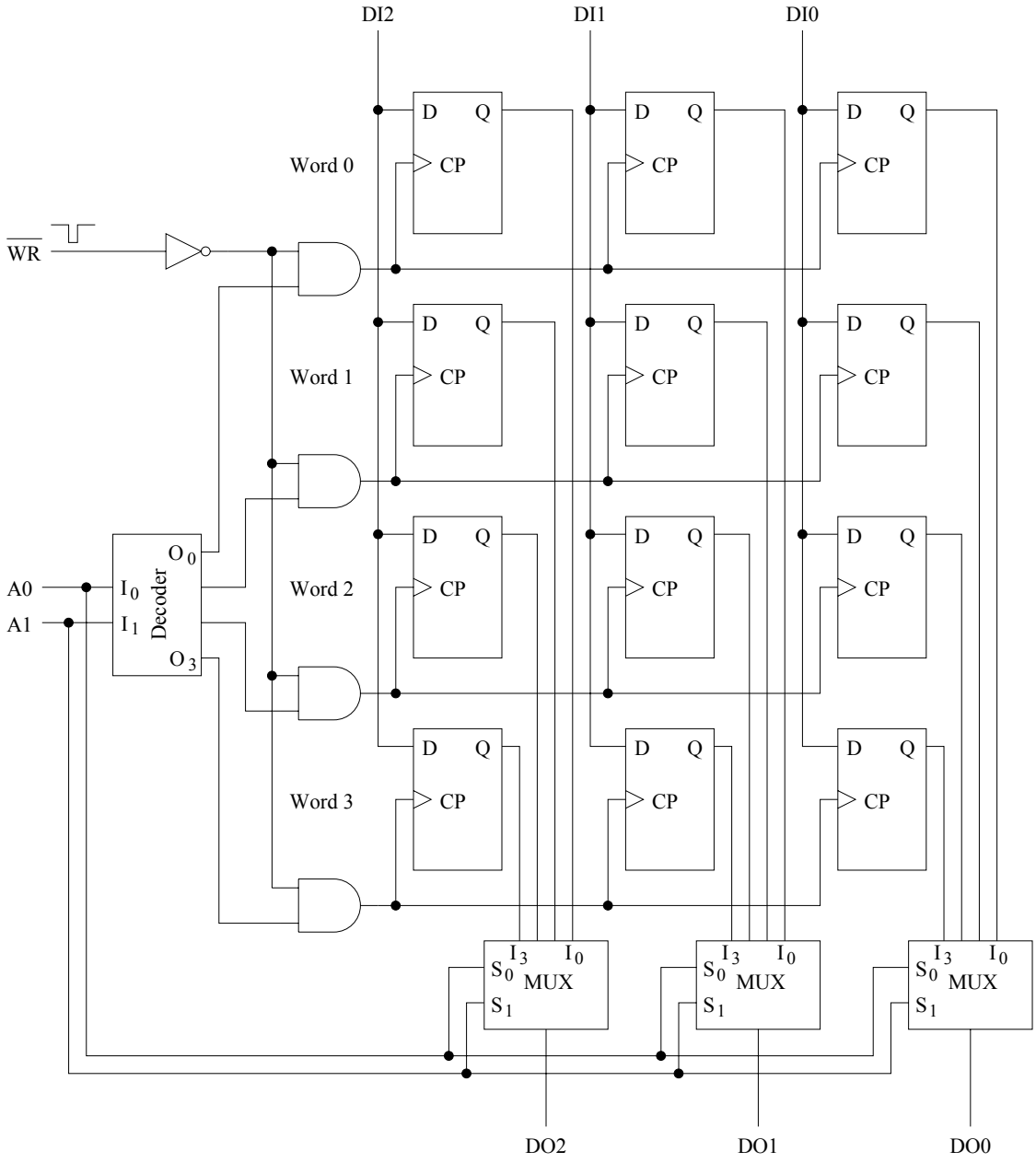


Figure 16.1 A simple 4 × 3 memory design using D flip-flops.

Another problem with this design is that we will not be able to use it as a building block to construct larger memory blocks. For example, we cannot build an 8×3 memory using two of these units. Such a hierarchical design needs a chip select input that either selects or deselects the whole unit.

We return to this design example in Section 16.4 that solves these two problems. Before revising our design, we look at some background information on some of the techniques we can use to connect multiple outputs to a single line as required for our data bus connections.

16.3 Techniques to Connect to a Bus

Data bus connections impose several requirements, some of which we have discussed in the last section. First and foremost, the data bus connections should support bidirectional data transfer. This eliminates those designs that require separate data in and out lines as in our previous design (see Figure 16.1). Furthermore, since several devices can be connected to the data bus, only the selected device should place data on the data bus. All other devices attached to the data bus should disconnect themselves (or at least act as though they were not connected to the data bus). We describe several techniques that satisfy one or both of these requirements.

16.3.1 Using Multiplexers

Based on our discussion of digital logic circuits, we know that we can use multiplexers to send multiple outputs onto a single line. In fact, we have used three 4-to-1 multiplexers in our memory design example in Figure 16.1. Multiplexers, however, do not satisfy the two requirements we have mentioned. You cannot use multiplexers to connect DI and DO inputs. Similarly, they do not provide an efficient way to implement the chip select function.

The next two subsections present two other techniques that are useful in deriving an implementation that satisfies these requirements.

16.3.2 Using Open Collector Outputs

A technique that is commonly used to connect several outputs uses "open collector" outputs. To give you some insight, Figure 16.2*a* shows how a normal gate output is designed (this is a reproduction of the NOT gate implementation shown in Figure 2.6 on page 47). A problem with this design is that the outputs of several gates cannot be connected. To see why, assume that three such normal gate outputs are tied together. In a normal gate, when the transistor is on, it can expect Vcc/R amperes of current through it. If one of the transistors is on and the other two are off, this one transistor would have to carry current that is three times its capacity (i.e., 3Vcc/R amperes). You can generalize this to a larger number of connections and see the corresponding increase in the required current-carrying capacity of each transistor. Remember that passing excessive current causes the transistor to burn out.

The idea of open collector design is to take the resistor out of the chip and provide open collector output to the chip pins (see Figure 16.2*b*). Open collector output can be in state 0 or 1 just as with a normal output. In addition, an open collector output can also be in a high

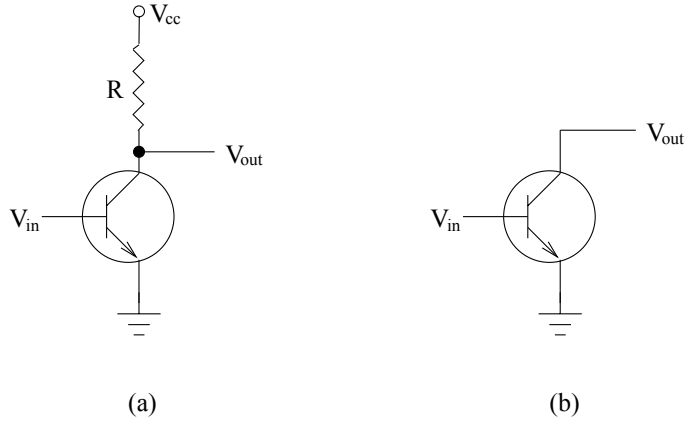
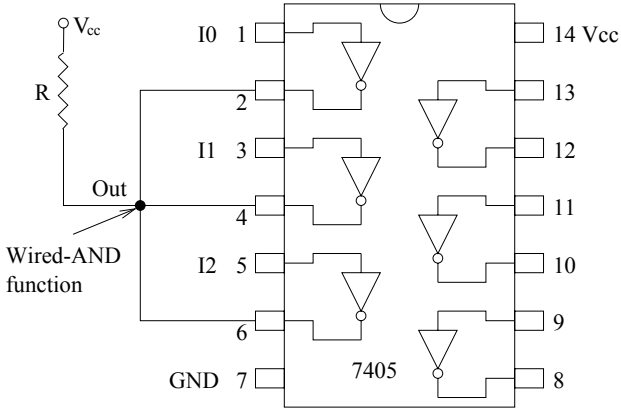


Figure 16.2 A simplified NOT gate implementation with normal and open collector outputs.



(a) Connection diagram

I2	I1	I0	Out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(b) Truth table

Figure 16.3 Open collector inverter chip: (a) 7405 connection diagram and an example circuit; (b) truth table for the example circuit in (a).

impedance (Z) state, in which the transistor is off. With the open collector outputs, we can connect several outputs without worrying about the capacity of these transistors. This is due to the fact that there is a single resistor that is connected to these common outputs as shown in Figure 16.3*a*. Thus we limit the current flow to Vcc/R amperes, independent of the number of connections.

Figure 16.3*a* shows details of the 7405 inverter chip that provides open collector outputs. The chip uses the same connection layout as the 7404 chip we have seen before (see page 50). This figure also shows how we can connect three outputs by "pulling" the **Out** point to Vcc through a resistor R. As noted in the diagram, by connecting these three open collector outputs, we are implementing the logical AND function. This is often referred to as the *wired-AND* function. The truth table in Figure 16.3*b* shows that this circuit implements a three-input NOR function.

Several open collector chips are commercially available. We present details about an example chip in Figure 16.4. This chip is a 4 $\times$ 4 register that uses D latches to store the data. Internally, the design is similar to that shown in Figure 16.1 except that the output goes through an open collector gate. The chip uses two read address lines (RA0 and RA1) and two write address lines (WA0 and WA1) along with two separate read ($\overline{\text{RE}}$) and write enable signals ($\overline{\text{WE}}$) as shown in Figure 16.4*b*. The use of separate address lines and enable signals for read and write operations gives the chip the ability to read and write simultaneously. This figure also shows the function table for read and write operations. We can use this chip to build larger memories by connecting outputs of several chips. For example, we can design an 8 $\times$ 4 memory by using two 74170 chips. The outputs of these two chips can be connected as we did in the open collector inverter chip example.

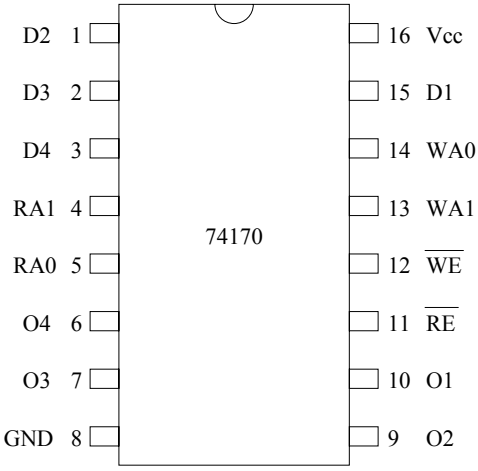
Open collector outputs are also used when signal drivers are needed. BCD to seven-segment decoder/driver chips are prime examples of this type of use of open collector outputs. An example of such a chip is the 74249, which is essentially an open collector version of the 7449 seven-segment decoder chip discussed in Chapter 2 on page 68.

A Problem: Can we use open collector outputs to solve our problem? We can certainly use open collector outputs to tie several outputs together. There is one major problem that makes them unsuitable for our memory design example. That is, the output is in high impedance (Z) state only if the output is supposed to be high. This forces us to apply appropriate inputs to get into this state. For example, if we are using the 7405 chip, we have to make sure that the inputs are low so that the outputs of deselected devices will not interfere with the output of a selected device. Thus, open collector devices are not suitable in the design of memory modules. We present the most commonly used solution next.

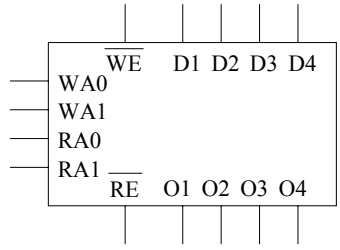
16.3.3 Using Tristate Buffers

With the exception of the open collector devices, all other logic circuits we have discussed have two possible states: 0 or 1. The devices we discuss now are called tristate devices as they can be in three states: 0, 1, or Z state. The states are similar to those associated with the open collector devices. There is one big difference between the two: tristate devices use a separate control signal so that the output can be in a high impedance state, independent of the data input. This particular feature makes them suitable for bus connections.

Figure 16.5*a* shows the logic symbol for a tristate buffer. When the enable input (E) is low, the buffer acts as an open circuit (i.e., output is in the high impedance state Z) as shown in



(a) Connection diagram



(b) Logic symbol

$\overline{WE}$	WA1	WA0	D inputs to
0	0	0	Word 0
0	0	1	Word 1
0	1	0	Word 2
0	1	1	Word 3
1	X	X	None

(c) Write function table

$\overline{RE}$	RA1	RA0	Output from
0	0	0	Word 0
0	0	1	Word 1
0	1	0	Word 2
0	1	1	Word 3
1	X	X	None (Z)

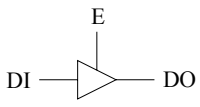
(d) Read function table

Figure 16.4 An example open collector register chip (X = don't care input, and Z = high impedance state).

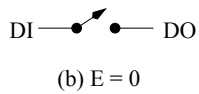
Figure 16.5*b*; otherwise, it acts as a short circuit (Figure 16.5*c*). The enable input must be high in order to pass the input data to output, as shown in the truth table (see Figure 16.5*d*).

Two example tristate buffer chips are shown in Figure 16.6. The 74367 chip provides six tristate buffers. These six buffers are grouped into four and two buffers, each with a separate enable input. Both enable inputs are low-active, as represented by $\overline{E1}$ and $\overline{E2}$. The 74368 chip is similar to the 74367 chip except that it provides tristate inverters instead of buffers. In the next section, we use the tristate buffers to modify our memory design example of Figure 16.1.

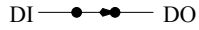
The 74373 is an example chip that provides tristate outputs. It uses eight D latches internally and feeds each latch $\overline{Q}$ through an inverting tristate buffer. The output enable ($\overline{OE}$) is a low-active input that controls the output of inverting tristate buffers. Thus, when $\overline{OE} = 1$, the outputs O0 through O7 float (i.e., the outputs are in the high impedance state). The data can be written



(a)



(b) E = 0

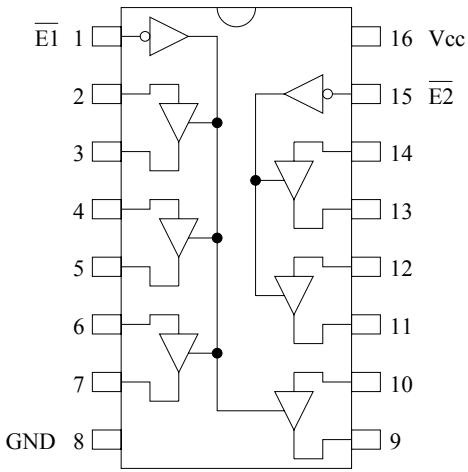


(c) E = 1

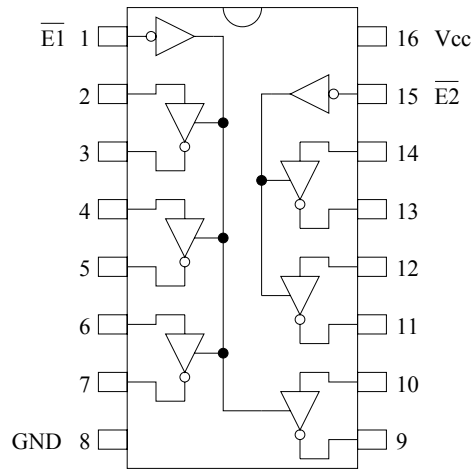
Inputs		Output
E	DI	DO
1	0	0
1	1	1
0	X	Z

(d)

Figure 16.5 Tristate buffer: (a) logic symbol; (b) it acts as an open circuit when the enable input is inactive (E = 0); (c) it acts as a closed circuit when the enable input is active (E = 1); (d) truth table (X = don't care input, and Z = high impedance state).



74367



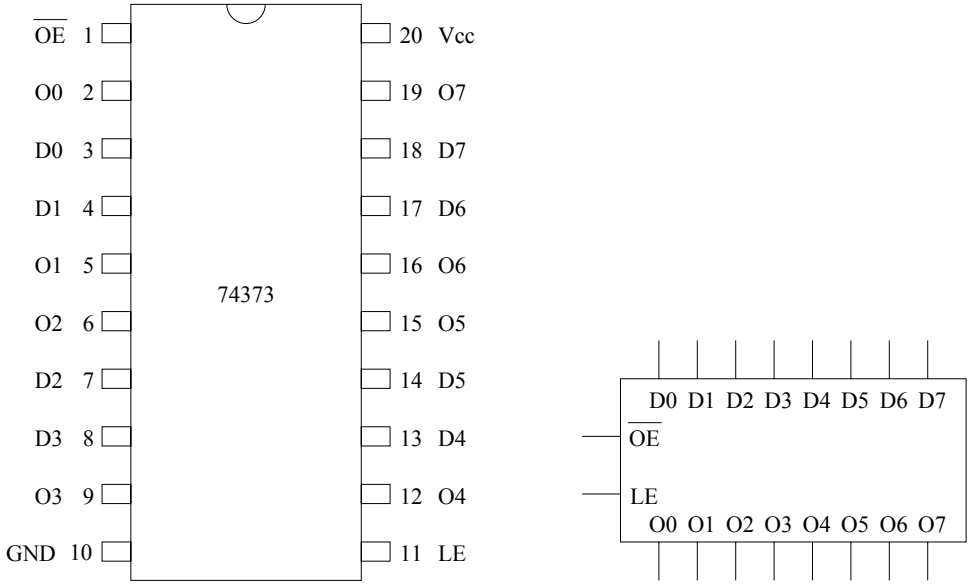
74368

Figure 16.6 Two example tristate buffer chips.

into the register by making the latch enable (LE) input high. Because this chip uses latches, the output follows the input as long as the LE input is high and the $\overline{OE}$ is low. In Section 16.5.1, we discuss how we can use this chip to build larger memories.

16.4 Building a Memory Block

We are now in a position to revise our previous memory design (shown in Figure 16.1) to remedy the two main problems we have identified before. Our revision is a minor one in the sense that we need to pass the outputs of the multiplexers through tristate buffers as shown in



(a) Connection diagram (b) Logic symbol

Figure 16.7 An example 8-bit tristate register.

Figure 16.8. The enable input signal for these output tristate buffers is generated by ANDing the chip select and read signals. The two inverters are used to provide low-active chip select ($\overline{\text{CS}}$) and memory read ($\overline{\text{RD}}$) inputs to the memory block.

With the use of these tristate buffers, we can now tie the corresponding data in and out signal lines together to satisfy the data bus connection requirements. Furthermore, we can completely disconnect the outputs of this memory block by making $\overline{\text{CS}}$ high.

We can represent our design using the logic symbol shown in Figure 16.9. Our design uses separate read and write signals. These two signals are part of the control bus (see Figure 1.5). It is also possible to have a single line to serve as a read and write line. For example, a 0 on this line can be interpreted as write and a 1 as read. Such signals are represented as the $\overline{\text{WR}}$/RD line, indicating low-active write and high-active read (see Exercise 16–5).

16.5 Building Larger Memories

Now that we know how to build memory blocks using devices that can store a single bit, we move on to building larger memory units using these memory blocks. We explain the design process by using some examples. The first example deals with designing an independent memory unit that is not hard-wired to a specific address in the memory address space. In a sense it is a larger version of the design shown in Figure 16.8. The second example shows how these

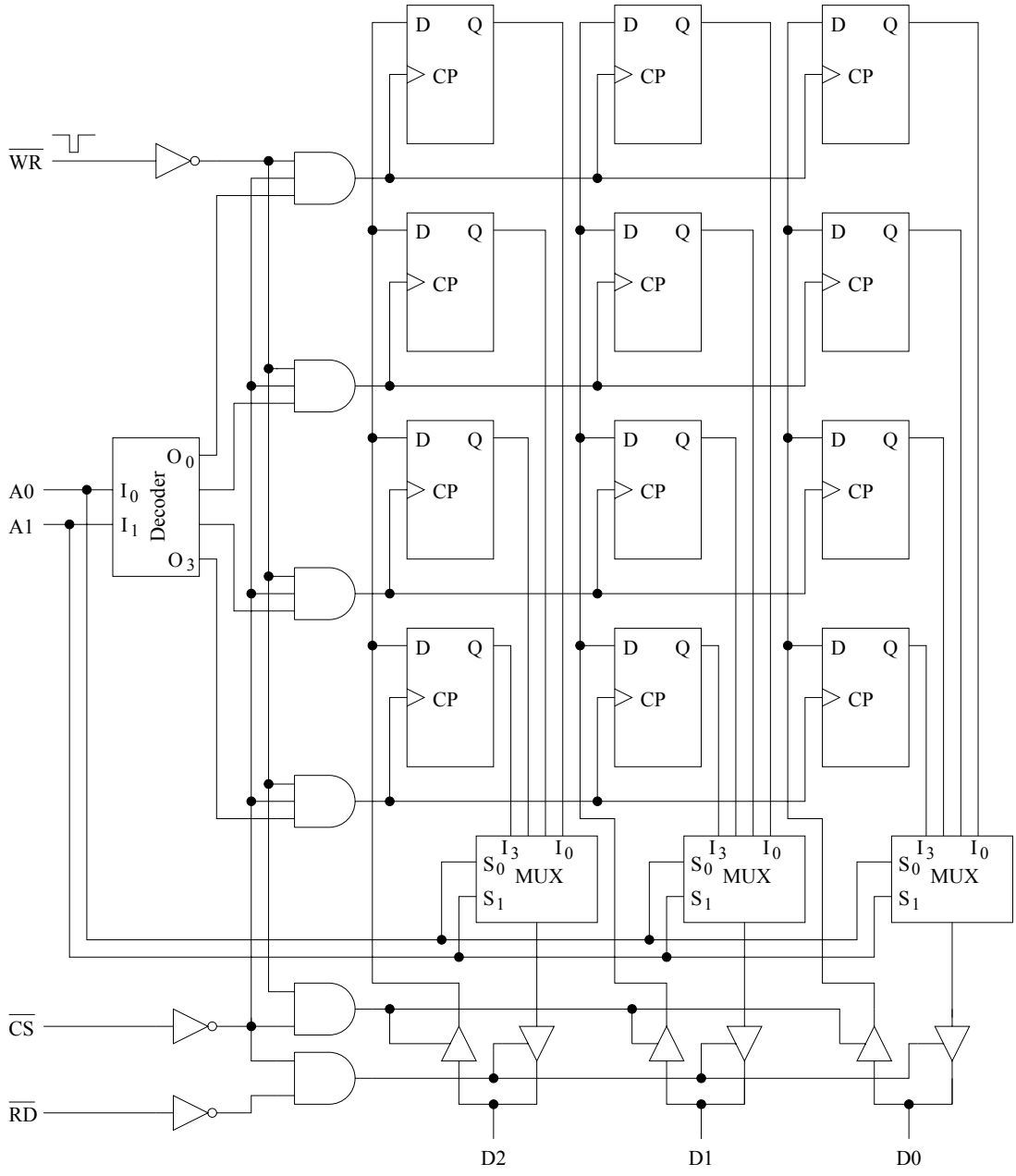


Figure 16.8 A 4 $\times$ 3 memory design using D flip-flops.

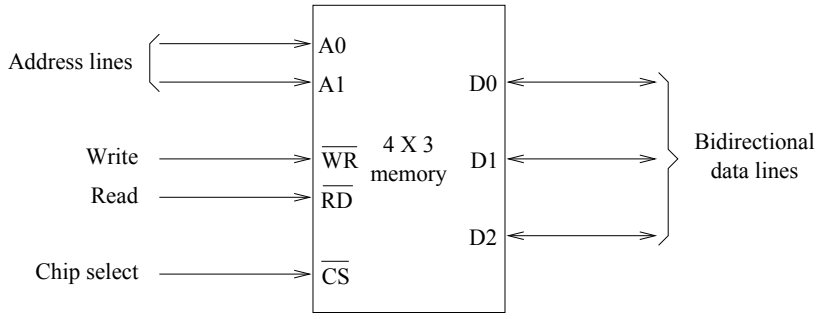


Figure 16.9 Block diagram representation of a 4 × 3 memory.

independent memory units can be mapped into the address space. A more detailed discussion is, however, deferred to the next section that describes full mapping as well as partial mapping of memory units into the address space.

16.5.1 Designing Independent Memory Modules

In this example, we use the 74373 register chip as the basic building block to construct a 2 × 16 memory module. Since each 74373 chip can store eight bits of data, we use four such chips organized as a 2 × 2 array (see Figure 16.10). In each row, the 74373 chip control signals LE and $\overline{\text{OE}}$ are tied together to form a 16-bit word. For example, in the top row, chip#1 receives the upper half of the 16-bit data (D8 to D15) and provides the upper half of the word to the data bus. The other half of the word is taken care of by chip#2.

A word on the notation: To simplify the schematic diagram, it is common practice to use arrows to identify a group of related signals. For example, instead of showing 16 separate lines for the data bus, we use an arrow (a double-headed arrow as the data bus is a bidirectional bus) and identify the group of signals it represents as D0 to D15. Similarly, the input connections of chip#1 are the upper half of the data bus D8 to D15.

By operating two chips in tandem, we double the word length. This technique can be used to build memories with larger word sizes using narrower units or chips. For example, to build a 64-bit word memory, we put eight 74373 chips in each row, with each chip supplying 8 bits of data. This is referred to as "horizontal" expansion, which takes care of the word length.

Next we have to increase the number of words. In our case, we just have to add one more word. Adding words is straightforward as well. To increase the number of words in memory, we simply add more rows, each row being a replication of the first row. This is referred to as "vertical" expansion. In our example, we create a second row exactly like the first one. All we have to do now is to make sure that only one row is selected at any point in time. This is done by the row select logic shown in Figure 16.10. Since our goal is to design an independent memory unit, we have to provide a chip select input. We use the LE input to write data into the selected row. The low-active 1-to-2 decoder does the row selection. In order to predicate both read and

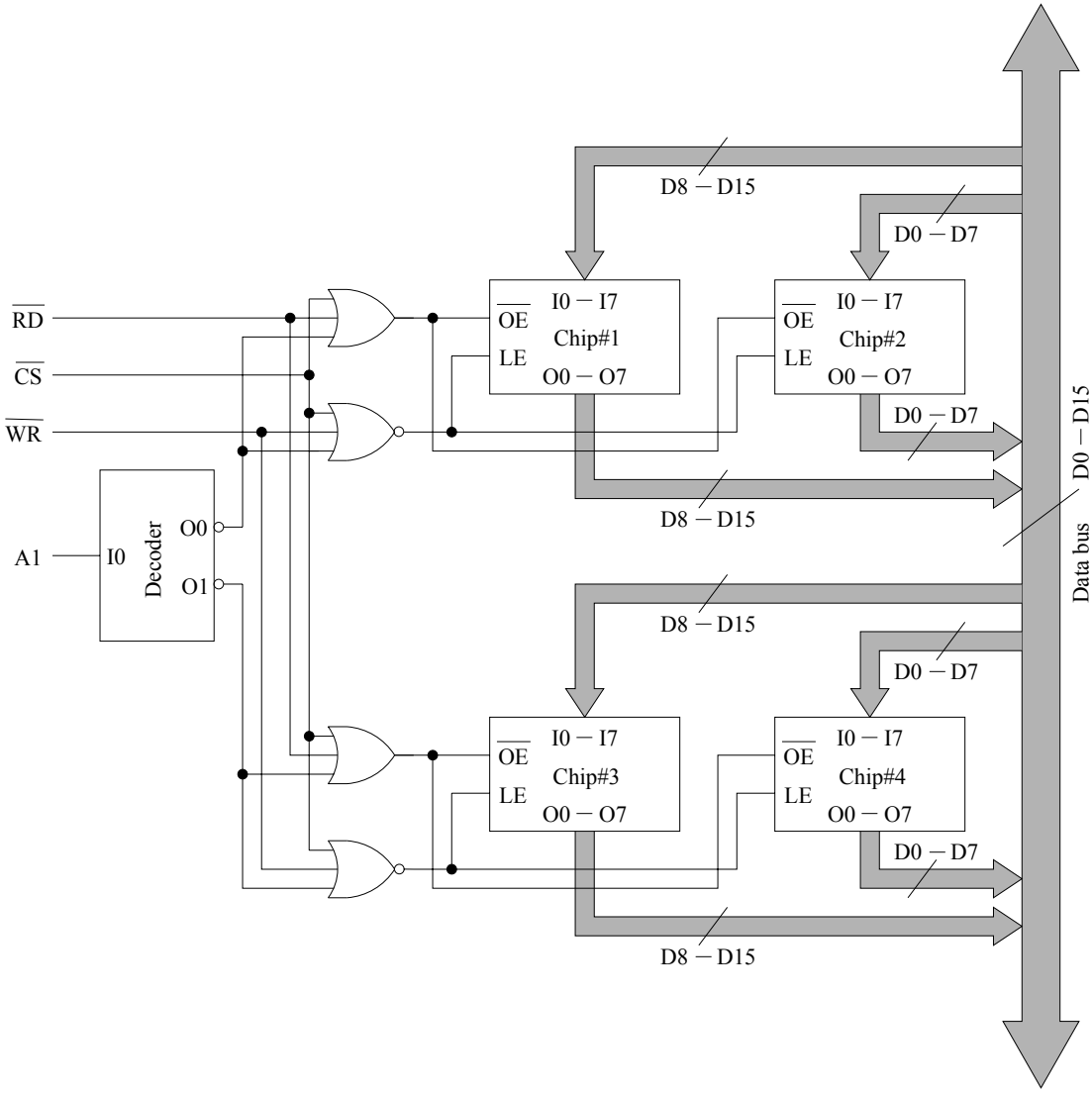


Figure 16.10 A 2 $\times$ 16 memory module using 74373 chips.

write on the active chip select signal, $\overline{\text{CS}}$ is used to gate the read and write signals. The $\overline{\text{OE}}$ line is used to output the data of the selected row.

As you can see from this design, the read operation on a specific row is enabled by the presence of active CS (i.e., $\overline{\text{CS}}$ = 0), selection of the row by the address decoder, and the presence of the $\overline{\text{RD}}$ signal. Similarly, the write operation is also predicated on the presence of chip select and address decoder output.

16.5.2 Designing Larger Memories Using Memory Chips

Before discussing the design procedure, we briefly present details about commercially available memory chips.

Memory Chips

Several commercial memory chips are available to build larger memories. Here we look at two example chips—a SRAM and a DRAM—from Micron Technology.

The SRAM we discuss is an 8-Mb chip that comes in three configurations: 512 K $\times$ 18, 256 K $\times$ 32, or 256 K $\times$ 36. Notice that, in the first and last configurations, word length is not a multiple of 8. These additional bits are useful for error detection/correction. These chips have an access time of 3.5 ns. The 512 K $\times$ 18 chip requires 19 address lines, whereas the 256 K $\times$ 32/36 versions require 18 address lines.

An example DRAM (it is a synchronous DRAM) is the 256-Mb capacity chip that comes in word lengths of 4, 8, or 16 bits. That is, this memory chip comes in three configurations: 64 M $\times$ 4, 32 M $\times$ 8, or 16 M $\times$ 16. The cycle time for this chip is about 7 ns.

In the days when the data bus widths were small (8 or 16), DRAM chips were available in 1-bit widths. Current chips use a word width of more than 1 as it becomes impractical to string 64 1-bit chips to get 64-bit word memories for processors such as the Pentium.

From the details of these 2 example memory chips, we see that the bit capacity of a memory chip can be organized into several configurations. If we focus on the DRAM chip, for example, what are the pros and cons of the various configurations? The advantage of wider memory chips (i.e., chips with larger word size) is that we require fewer of them to build a larger memory. As an example, consider building memory for your Pentium-based PC. Even though the Pentium is a 32-bit processor, it uses a 64-bit wide data bus. Suppose that you want to build a 16 M $\times$ 64 memory. We can build this memory by using four 16 M $\times$ 16 chips, all in a single row. How do we build such a memory using, for example, the 32 M $\times$ 8 version of the chip? Because our word size is 64, we have to use 8 such chips in order to provide 64-bit wide data. That means we get 32 M $\times$ 64 memory as the minimum instead of the required 16 M $\times$ 64. The problem becomes even more serious if we were to use the 64 M $\times$ 4 version chip. We have to use 16 such chips, and we end up with a 64 M $\times$ 64 memory. This example illustrates the tradeoff between using "wider" memories versus "deeper" memories.

Larger Memory Design

Before proceeding with the design of a memory unit, we need to know if the memory address space (MAS) supported by the processor is byte addressable or not. In a byte-addressable space, each address identifies a byte. All popular processors—the Pentium, PowerPC, SPARC, and MIPS—support byte-addressable space. Therefore, in our design examples, we assume byte-addressable space.

We now discuss how one can use memory chips, such as the ones discussed before, to build system memory. The procedure is similar to the intuitive steps followed in the previous design examples.

First we have to decide on the configuration of the memory chip, assuming that we are using the DRAM chip described before. As described in the last section, independent of the configuration, the total bit capacity of a chip remains the same. That means the number of chips required remains the same. For example, if we want build a 64 M $\times$ 32 memory, we need eight chips. We can use eight 64 M $\times$ 4 in a single row, eight 32 M $\times$ 8 in 2 $\times$ 4 array, or 16 M $\times$ 16 in 4 $\times$ 2 array. Although we have several alternatives for this example, there may be situations where the choice is limited. For example, if we are designing a 16 M $\times$ 32 memory, we have no choice but to use the 16 M $\times$ 16 chips.

Once we have decided on the memory chip configuration, it is straightforward to determine the number of chips and the organization of the memory unit. Let us assume that we are using D $\times$ W chips to build an M $\times$ N memory. Of course, we want to make sure that D $\leq$ M and W $\leq$ N.

$$\text{Number of chips required} = \frac{\text{M} \times \text{N}}{\text{D} \times \text{W}},$$

$$\text{Number of rows} = \frac{\text{M}}{\text{D}},$$

$$\text{Number of columns} = \frac{\text{N}}{\text{W}}.$$

The read and write lines of all memory chips should be connected to form a single read and write signal. These signals are connected to the control bus memory read and write lines. For simplicity, we omit these connections in our design diagrams.

Data bus connections are straightforward. Each chip in a row supplies a subset of data bits. In our design, the right chip supplies D0 to D15, and the left chip supplies the remaining 16 data bits (see Figure 16.11).

For each row, connect all chip select inputs as shown in Figure 16.11. Generating appropriate chip select signals is the crucial part of the design process. To complete the design, partition the address lines into three groups as shown in Figure 16.12.

The least significant Z address bits, where Z = $\log_2$(N/8), are not connected to the memory unit. This is because each address going into the memory unit will select an N-bit value. Since we are using byte-addressable memory address space, we can leave the Z least significant bits that identify a byte out of N/8 bytes. In our example, N = 32, which gives us Z = 2. Therefore, the address lines A0 and A1 are not connected to the memory unit.

The next Y address bits, where Y = $\log_2$D, are connected to the address inputs of all the chips. Since we are using 16 M chips, Y = 24. Thus, address lines A2 to A25 are connected to all the chips as shown in Figure 16.11.

The remaining most significant address bits X are used to generate the chip select signals. This group of address bits plays an important role in mapping the memory to a part of the memory address space. We discuss this mapping in detail in the next section. The design shown in Figure 16.11 uses address lines A26 and A27 to generate four chip select signals, one for each row of chips. We are using a low-active 2-to-4 decoder to generate the $\overline{\text{CS}}$ signals.

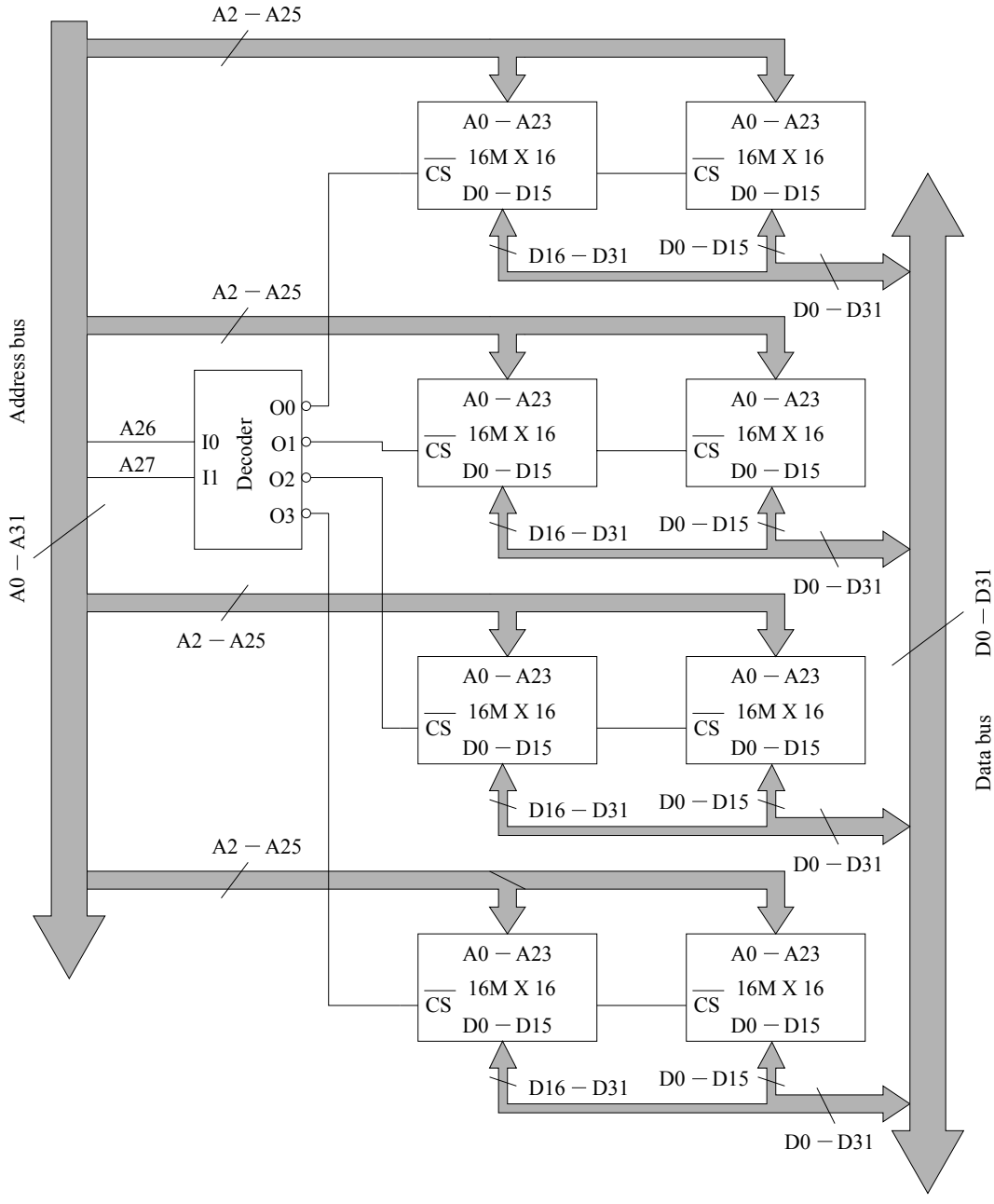


Figure 16.11 Design of a 64 M × 32 memory using 16 M × 16 memory chips.

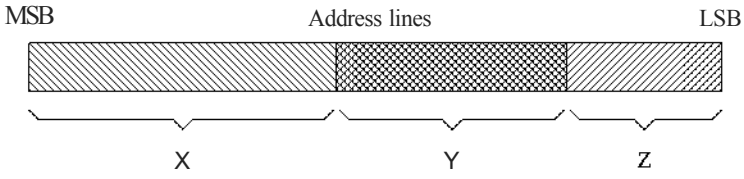


Figure 16.12 Address line partition.

The top row of chips in Figure 16.11 is mapped to the first 64-MB address space (i.e., from addresses 0 to 2^{26} - 1). The second row is mapped to the next 64-MB address space, and so on. After reading the next section, you will realize that this is a partial mapping.

16.6 Mapping Memory

Memory mapping refers to the placement of a memory unit in the memory address space (MAS). For example, the Pentium supports 4 GB of address space (i.e., it uses 32 bits for addressing a byte in memory). If your system has 128 MB of memory, it can be mapped to one of several address subspaces. This section describes how this mapping is done.

16.6.1 Full Mapping

Full mapping refers to a one-to-one mapping function between the memory address and the address in MAS. This means, for each address value in MAS that has a memory location mapped, there is one and only one memory location responding to the address.

Full mapping is done by completely decoding the higher-order X bits of memory (see Figure 16.12) to generate the chip select signals. Two example mappings of 16 M x 32 memory modules are shown in Figure 16.13. Both these mappings are full mappings as all higher-order X bits participate in generating the $\overline{CS}$ signal.

Logically we can divide the 32 address lines into two groups. One group, consisting of address lines Y and Z, locates a byte in the selected 16 M x 32 memory module. The remaining higher-order bits (i.e., the X group) are used to generate the $\overline{CS}$ signal. Given this delineation, it is simple to find the mapping.

We illustrate the technique by using the two examples shown in Figure 16.13. Since the memory modules have a low-active chip select input, a given module is selected if its $\overline{CS}$ input is 0. For Module A, the NAND gate output is low when A26 and A29 are low and the remaining four address lines are high. Thus, this memory module responds to memory read/write activity whenever the higher-order six address bits are 110110. From this, we can get the address locations mapped to this module as D8000000H to DBFFFFFFH. For convenience, we have expressed the addresses in the hexadecimal system (as indicated by the suffix letter H). The address D8000000H is mapped to the first location and the address DBFFFFFFH to the last location of Module A. For addresses that are outside this range, the $\overline{CS}$ input to Module A is high and, therefore, it is deselected.

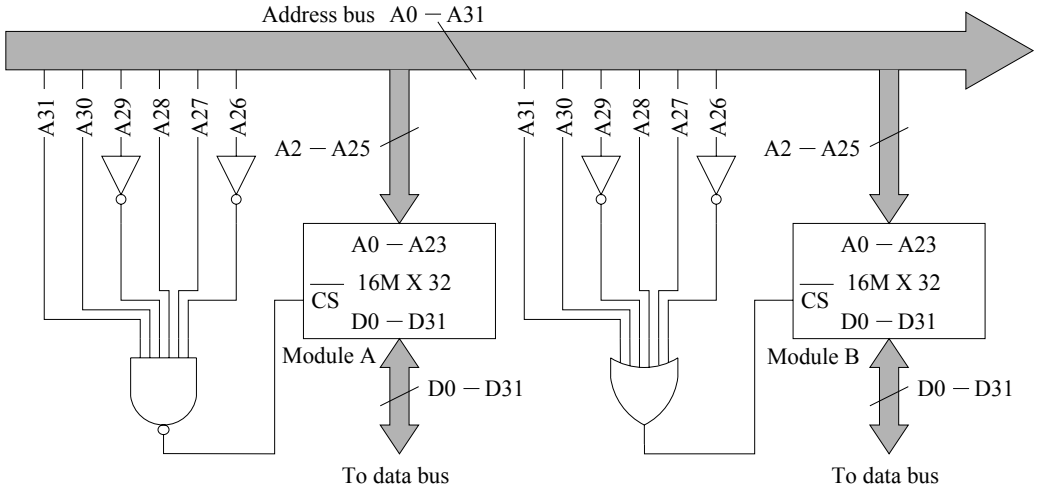


Figure 16.13 Full address mapping.

For Module B, the same inputs are used except that the NAND gate is replaced by an OR gate. Thus, the output of this OR gate is low when the higher-order six address bits are 001001. From this, we can see that mapping for Module B is 24000000H to 27FFFFFH. As these two ranges are mutually exclusive, we can keep both mappings without causing conflict problems.

16.6.2 Partial Mapping

Full mapping is useful in mapping a memory module; however, often the complexity associated with generating the $\overline{\text{CS}}$ signal is not necessary. For example, we needed a 6-input NAND or OR gate to map the two memory modules in Figure 16.13. Partial mapping reduces this complexity by mapping each memory location to more than one address in MAS. We can obtain simplified $\overline{\text{CS}}$ logic if the number of addresses a location is mapped to is a power of 2.

Let us look at the mapping of Module A in Figure 16.14 to clarify some of these points. The $\overline{\text{CS}}$ logic is the same except that we are not connecting the A26 address line to the NAND gate. Because A26 is not participating in generating the signal, it becomes a don't care input. In this mapping, Module A is selected when the higher-order six address bits are 110110 or 110111. Thus, Module A is mapped to the address space D8000000H to DBFFFFFFH and DC000000H to DFFFFFFFH. That is, the first location in Module A responds to addresses D8000000H and DC000000H. Since we have left out one address bit A26, two (i.e., 2^1) addresses are mapped to a memory location. In general, if we leave out k address bits from the chip select logic, we map 2^k addresses to each memory location. For example, in our memory design of Figure 16.11, four address lines (A28 to A31) are not used. Thus, $2^4 = 16$ addresses are mapped to each memory location.

We leave it as an exercise to verify that each location in Module B is mapped to eight addresses as there are three address lines that are not used to generate the $\overline{\text{CS}}$ signal.

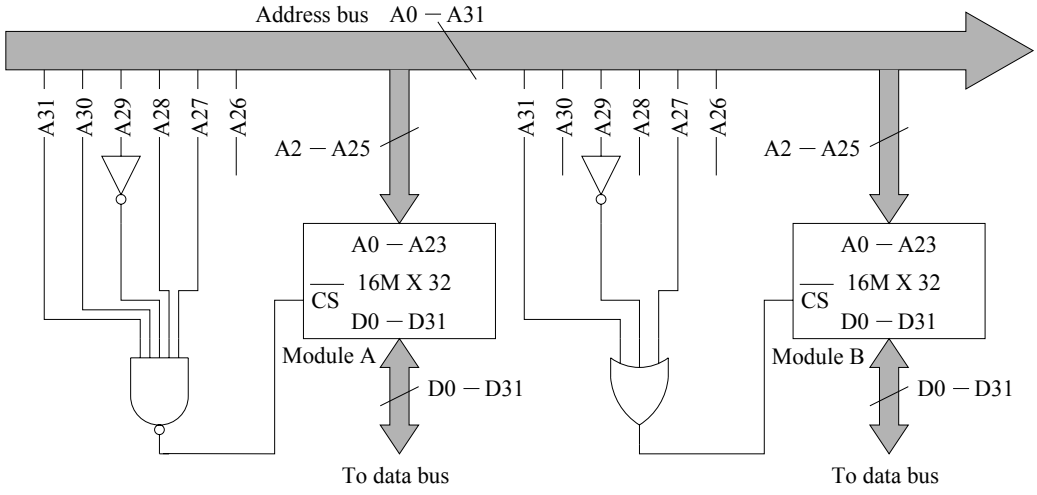


Figure 16.14 Partial address mapping.

16.7 Alignment of Data

We can use our memory example to understand why data alignment improves the performance of computer systems. Suppose we want to read 32-bit data from the memory shown in Figure 16.11. If the address of these 32-bit data is a multiple of four (i.e., address lines A0 and A1 are 0), the 32-bit data are stored in a single row of memory. Thus the processor can get the 32-bit data in one read cycle. If this condition is not satisfied, then the 32-bit data item is spread over two rows. Thus the processor needs to read two 32-bits of data and extract the required 32-bit data. This scenario is clearly demonstrated in Figure 16.15.

In Figure 16.15, the 32-bit data item stored at address 8 (shown by hashed lines) is aligned. Due to this alignment, the processor can read this data item in one read cycle. On the other hand, the data item stored at address 17 (shown shaded) is unaligned. Reading this data item requires two read cycles: one to read the 32 bits at address 16 and the other to read the 32 bits at address 20. The processor can internally assemble the required 32-bit data item from the 64-bit data read from the memory.

You can easily extend this discussion to the Pentium 64-bit data bus. It should be clear to you that aligned data improve system performance.

- *2-Byte Data*: A 16-bit data item is aligned if it is stored at an even address (i.e., addresses that are multiples of two). This means that the least significant bit of the address must be 0.
- *4-Byte Data*: A 32-bit data item is aligned if it is stored at an address that is a multiple of four. This implies that the least significant two bits of the address must be 0 as discussed in the last example.

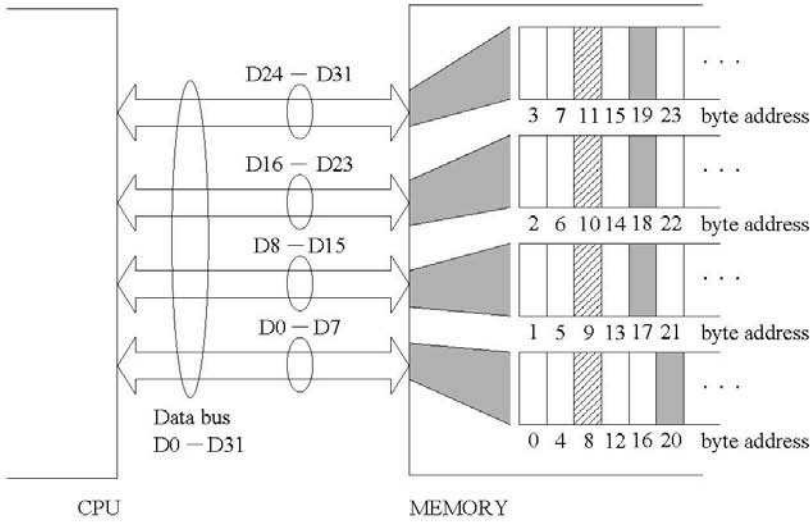


Figure 16.15 Byte-addressable memory interface to the 32-bit data bus.

- *8-Byte Data*: A 64-bit data item is aligned if it is stored at an address that is a multiple of eight. This means that the least significant three bits of the address must be 0. This alignment is important for Pentium processors, as they have a 64-bit wide data bus. On 80486 processors, since their data bus is 32-bits wide, a 64-bit data item is read in two bus cycles and alignment at 4-byte boundaries is sufficient.

The Intel 80X86 family of processors allows aligned and unaligned data items. Of course, unaligned data cause performance degradation. Alignment constraints of this type are referred to as *soft alignment* constraints. Because of the performance penalty associated with unaligned data, some processors, such as the Motorola 68000 and Intel i860, do not allow unaligned data. This alignment constraint is referred to as the *hard alignment* constraint.

16.8 Interleaved Memories

In our memory designs thus far, we have mapped a block of contiguous memory addresses to a memory module. If we want a larger memory, we simply add more memory modules. This modular design is useful in incrementally expanding the memory. This design, however, has one major disadvantage. Since sequential addresses are mapped to the same memory module, we have to wait for the memory access time for each word we read from the memory module. As an example, let's assume that memory access time is four clock cycles. Using our design, we need $8 \times 4 = 32$ cycles to read eight sequential words. Interleaved memories help reduce this access time.

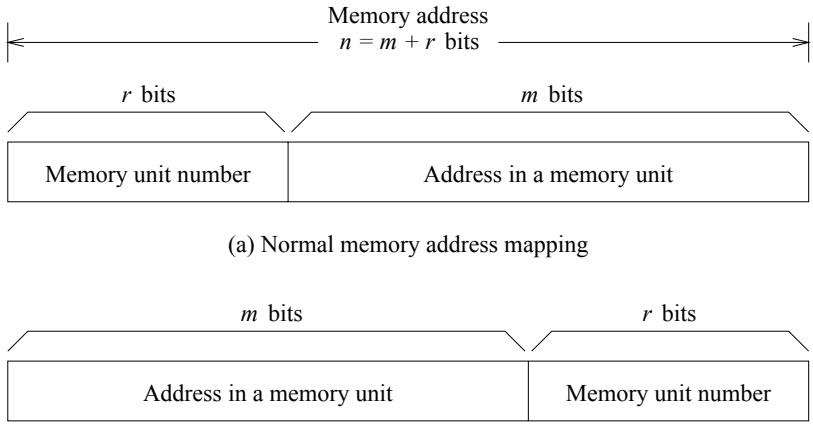


Figure 16.16 Interleaved memories use the lower-order bits to select a memory bank.

16.8.1 The Concept

As we mentioned in Chapter 8, high-performance computers typically use interleaved memories to improve access performance. Essentially, these memories allow overlapped access to hide the memory latency. The key idea is to design the memory system with multiple banks (similar to our memory modules) and access all banks simultaneously so that access time can be overlapped. We explain this in detail next.

In the memory designs we described so far, we divided the n-bit memory address bits into two parts: the higher-order r bits are used to identify a memory module and the lower-order m bits are used to specify a location in the memory module. As shown in Figure 16.16a, $n = r + m$. This technique is sometimes called *high-order interleaving*.

To provide overlapped access to sequential addresses, we have to resort to *low-order interleaving*, as shown in Figure 16.16b. In this design, the lower-order r bits are used to identify a memory module and the higher-order m bits are used to specify a location. We normally use the term interleaved memory to mean low-order interleaving. In these designs, the memory modules are referred to as *memory banks*.

In interleaved memories, memory addresses are mapped on a round-robin basis. Thus, in a B-bank design, address 0 is mapped to bank 0, address 1 to bank 1, and so on. A memory address $addr$ is mapped to memory bank $b = addr$ MOD B. Some example mappings are shown in Table 16.1.

We can implement interleaved memories using two possible designs: synchronized access organization or independent access organization. These two organizations are described next.

Table 16.1 Example mappings

Memory bank number	Memory addresses mapped
Bank 0	$0, B, 2B, \ldots$
Bank 1	$1, B+1, 2B+1, \ldots$
Bank 2	$2, B+2, 2B+2, \ldots$
. . .	. . .
Bank $B-1$	$B-1, 2B-1, 3B-1, \ldots$

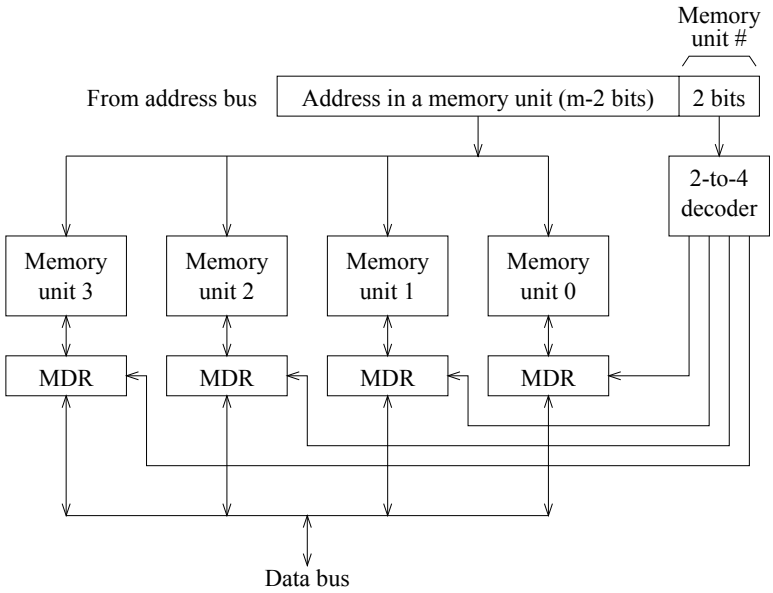
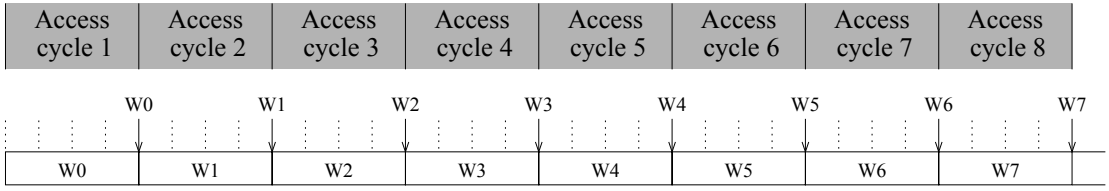


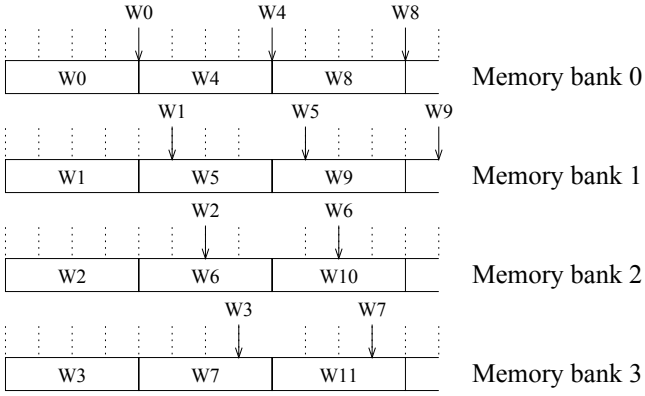
Figure 16.17 Interleaved memory design with synchronized access organization.

16.8.2 Synchronized Access Organization

In this organization, the upper m bits are presented to all memory banks simultaneously. We illustrate the working of this design by means of an example. Figure 16.17 shows this design for a four-bank interleaved memory. As before, it is assumed that the memory access takes four clock cycles. Once the address is presented, all four memory banks initiate their memory access cycles. Thus, in four clock cycles, we will have four data words from the four banks. These data words are latched into four tristate memory data registers (MDRs). We can transfer these four words in four clocks by selecting the appropriate MDR from the lower-order 2-bits



(a) Noninterleaved memory access



(b) Interleaved memory access

Figure 16.18 Interleaved memory allows pipelined access to memory.

using a 2-to-4 decoder (see Figure 16.17). Simultaneously, we can present the next address to the memory banks to initiate the next memory access cycle. Thus, by the time we transfer the four words from the first access cycle, we will have the next four words from the second access cycle. This process is shown in Figure 16.18.

As shown in Figure 16.18*a*, noninterleaved memory access takes four clocks for each access. Interleaved memory, on the other hand, transfers the first word (W0) after four clocks (see Figure 16.18*b*). After that, one word is transferred every clock cycle. Thus, to transfer eight words, we need 12 clock cycles (as opposed to 32 clocks in a noninterleaved design).

16.8.3 Independent Access Organization

A drawback with the synchronized design is that it does not efficiently support access to nonsequential access patterns. The independent access organization allows pipelined access even for arbitrary addresses. In order to support this kind of access, we have to provide each memory bank with a memory address register (MAR) to latch the address that the bank should use (see Figure 16.19). In our example, we can load four different addresses for the four banks. In this design, we do not need the MDR registers to latch the data. Instead, the data are read directly from the memory bank.

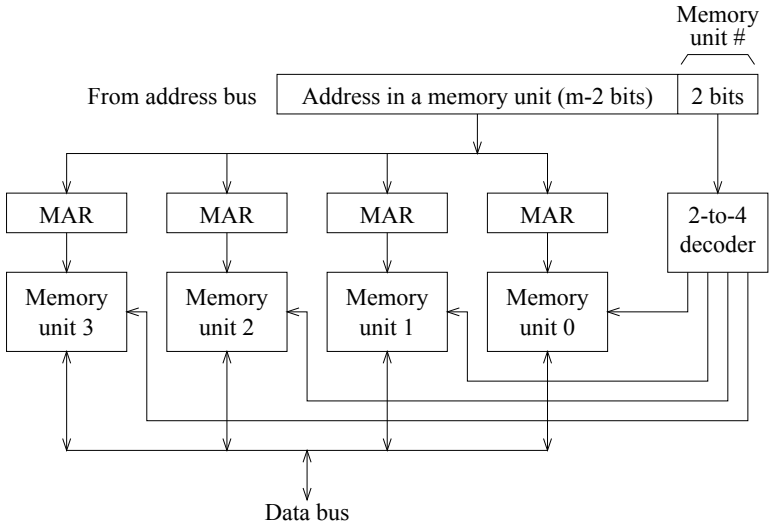


Figure 16.19 Interleaved memory design with independent access organization.

Independent design still provides the same kind of pipelined access as the synchronized access design. To see this, let's trace the first few data transfers from addresses 100, 81, 54, 31, 108, and 121. We can use (address MOD 4) to determine the memory bank number. The first address 100 is sent to bank 0. Once this address is latched in the MAR of bank 0, this bank can initiate the memory access cycle. During the next clock cycle, address 81 is latched into the MAR of bank 1. The next two addresses are sent to banks 2 and 3. After latching the address in the MAR of bank 3, data from bank 0 are available. So the next clock cycle is used to transfer the data from bank 0. We can now load the new address (108) in the MAR register of bank 0. Similarly, during the next clock, data from bank 1 are transferred, and a new address (121) is latched into the MAR of bank 1. This process repeats to provide pipelined access for arbitrary addresses. Of course, this organization also works for the sequential addresses.

16.8.4 Number of Banks

In our examples, we assumed four banks with a memory access time of four cycles. This gives us a data transfer rate of one word per cycle (after the initial startup delay). In general, how many memory banks do we need to provide one word per clock cycle transfer rate if the memory access time is M cycles? From our discussion, it should be clear that the number of banks B should be at least M.

We should caution the reader that interleaved memory does not always provide pipelined access to data. It depends on the memory access pattern. Next we give two examples representing best- and worst-case scenarios.

Example 16.1 *Suppose we have eight memory banks with an access time of six clock cycles. Find the time needed to read 16 words of a vector with stride 1.*

Note that a stride of 1 corresponds to sequential access, assuming that each address supplies a word (see page 308 for information on vector stride). This means we will be accessing the memory sequentially. Thus, whichever design we use, we will get pipelined access. This is the best-case scenario. Since the number of memory banks is at least six, we can get 1 word per cycle data transfer rate. Thus, to complete the transfer of 16 words, we have to wait 6 clocks for the first word and 16 additional clocks to transfer the 16 words. Therefore, we need $6 + 16 = 22$ clock cycles. □

Example 16.2 *Consider the memory system in the previous example. Find the time needed to read 16 words of a vector with stride 8.*

Since the stride is equal to the number of memory banks, we know that all 16 word addresses are mapped to the same bank. In this case, interleaving is not useful at all. We have to access each word sequentially from the same bank. Thus, we need $16 \times 6 = 96$ clock cycles. This is the worst-case scenario. □

16.8.5 Drawbacks

Why do we see interleaved memories only in high-performance computers? Why not in our PCs? One reason is that interleaved memories involve complex design. We have seen some of this complexity in our examples. We need extra registers (MAR or MDR) to latch addresses or data. What we have not shown in these examples is the control circuit needed to transfer data and to load addresses.

Another major reason is that interleaved memories do not have good fault-tolerance properties. That is, if one bank fails, we lose the entire memory. On the other hand, in traditional designs, failure of a module disables only a specific area of the memory address space mapped to the failed module. Furthermore, interleaved memories do not lend themselves well to incremental memory expansion.

16.9 Summary

We have discussed the basic memory design issues. We have shown how flip-flops and latches can be used to build memory blocks. Interfacing a memory unit to the system bus typically requires tristate buffers. Open collector devices and multiplexers are useful only in some special cases.

Building larger memories requires both horizontal and vertical expansion. Horizontal expansion is used to expand the word size, and vertical expansion provides an increased number of words. We have shown how one can design memory modules using standard memory chips. In all these designs, chip select plays an important role in allowing multiple entities to be attached to the system bus.

Chip select logic also plays an important role in mapping memory modules into the address space. Two basic mapping functions are used: full mapping and partial mapping. Full mapping provides a one-to-one mapping between memory locations and addresses. Partial mapping maps each memory location to a number of addresses equal to a power of 2. The main advantage of partial mapping is that it simplifies the chip select logic.

We have discussed the importance of data alignment. Unaligned data can lead to performance degradation. We have discussed the reasons for improvement in performance due to alignment of data.

In the last section, we presented details on interleaved memories, which are typically used in vector processors to facilitate pipelined access to memory.

We have provided only the basic information in designing memory modules. One aspect that we have not looked at is the interface details to handle slow memories. It is important to look at this issue as the speed gap between memories and processors keeps increasing. If a memory unit cannot respond at the rate required by the processor, the memory unit should be able to ask the processor to wait. This is typically done by the memory signaling the processor that it has not completed the operation (memory read or write). For example, the Pentium has a pin labeled READY that any device can pull down to make the processor wait. Thus, slow memories can use the READY input to slow the processor down to the speed of memory. We have discussed these issues in Chapter 5.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Building larger memories
- Chip select
- Data alignment
- Data alignment—hard alignment
- Data alignment—soft alignment
- Interleaved memories
- Interleaved memories—independent access organization
- Interleaved memories—synchronized access organization
- Memory address space
- Memory design with D flip-flops
- Memory mapping—full mapping
- Memory mapping—partial mapping
- Nonvolatile memories
- Open collector outputs
- Tristate buffers

16.10 Exercises

16–1 What is the purpose of the multiplexers in Figure 16.1?

16–2 What is the reason for not being able to connect the data in and out lines in Figure 16.1?

16–3 Suppose that we want to extend the memory block in Figure 16.1 to 4×8. Do you need to change the decoder? How many more multiplexers do you need?

16–4 Suppose that we want to extend the memory block in Figure 16.1 to 16×8. What kind of decoder do you use in your design? What kind of multiplexers do we need?

16–5 Figure 16.8 uses separate read and write lines. Modify this design to use a single line for both read and write operations.

16–6 What are the advantages of tristate buffers over open collector buffers?

16–7 Figure 16.10 shows a 2×16 memory module using four 74373 chips. Suppose that we want to build a 4×16 memory using the same chips. Extend the design in Figure 16.10 to derive the new design.

16–8 We have partitioned the address lines into three groups: X, Y, and Z (see Figure 16.12). Explain the purpose of each group of address bits.

16–9 What are the advantages and disadvantages of full mapping over partial mapping?

16–10 Figure 16.11 shows a 64 M $\times$ 32 memory design. We have stated that this memory block is partially mapped. Give the number of addresses to which each memory location of this block is mapped. What are the addresses of the first location?

16–11 Suppose that we have replaced the 6-input NAND gate in Figure 16.13 by a 6-input AND gate. Give the address mapping of Module A. It is sufficient to give the mapping of the first and last location.

16–12 Suppose that we have replaced the 6-input OR gate in Figure 16.13 by a 6-input NOR gate. Give the address mapping of Module B. It is sufficient to give the mapping of the first and last location.

16–13 We have stated that each address of Module B in Figure 16.14 is mapped to eight addresses. Give the eight addresses to which the first location of Module B is mapped. What are the eight addresses to which the last location is mapped?

16–14 Assume a hypothetical processor with a memory address space of 256 bytes. Design a 16×8 memory system that is mapped to locations 160 to 175 (decimal) using 4×4 memory chips.

16–15 Modify your memory design of the last exercise to partially map to locations 160 to 175 (decimal) and 224 to 239 (decimal). You just need to show the changes to your chip select logic.

16–16 Consider a hypothetical system that supports an address space of 128 bytes. Its data bus width is 8 bits. Design a 16×8 memory block using 4×4 memory chips and map it to address locations 48 to 63. You need to show all the details including the chip select logic needed to map the memory. You can use block diagram notation for memory chips. Note that you can use only 2-input AND gates and inverters to implement your chip select logic.

16–17 How would you change your design in Exercise 16–16 if the mapping were to locations 80 to 95 instead? You need to show only the changes. There is no need to reproduce the part of the circuit that is not changing. Note that you can use only 2-input AND gates and inverters to implement your chip select logic.

16–18 This exercise also refers to the memory block you designed in Exercise 16–16. Show how a single block of 16×8 memory can be mapped to locations 80 to 95 and 112 to 127. In other words, the first location of the 16×8 memory block should respond to addresses 80 and 112, the second location to addresses 81 and 113, and so on. Note that you can use only 2-input AND gates and inverters to implement your chip select logic.

16–19 Consider a hypothetical machine that can address 256 bytes of memory. It uses a 16-bit wide data bus. Suppose that only 32×8 memory chips are available. Using these memory chips, design a 64×16 memory system starting at address 128 (the address is expressed in decimal). You need to show how the address, data, and chip select lines of the chips are connected to the system address and data buses. You don't have to show the read/write connections to the control bus. Be sure to show the necessary chip select logic using only 2-input gates (AND, OR) and inverters. Assume that the chip select is low-active.

16–20 We have mentioned that aligned data improve performance. Discuss the reasons for this improvement.

16–21 Suppose that the data bus width is 32 bits. From the performance viewpoint, does it make sense to talk about 64-bit aligned data? Explain.

16–22 What are the advantages and disadvantages of interleaved memories?

16–23 What are the advantages and disadvantages of synchronized and independent access interleaved memory designs?

16–24 Let B be the number of banks and M be the memory access time in cycles. Explain why we need to have $B \geq M$ to support a one-word-per-cycle data-transfer rate.

16–25 Suppose we have an interleaved memory with eight banks. As in our examples on page 689, the access time is six clock cycles. Find the time needed to access data at the following 10 addresses: 152, 153, 154, 155, 156, 157, 158, 159, 160, and 161.

16–26 Suppose we have an interleaved memory as in the last exercise. Find the time needed to access data at the following 10 addresses: 152, 320, 868, 176, 184, 800, 880, 640, 560, and 160.

16–27 In the last exercise, does it matter whether you use the synchronized access design or the independent access design?

Chapter 17

Cache Memory

Objectives

- To introduce basic concepts of cache memory;
- To explain how and why cache memory works;
- To describe the various policies used in designing a cache memory system;
- To discuss performance implications of cache parameters and policies;
- To look at some practical cache implementations.

Cache memory successfully bridges the speed gap between the processor and memory. The cache is a small amount of fast memory that sits between the processor and memory in the memory hierarchy. We first describe how the cache memory works (Section 17.2). The main reason cache memory improves overall system performance is because programs exhibit locality in their referencing behavior. The principle of locality is discussed in Section 17.3.

Cache design basics are discussed next. Operation of cache memory requires four policies: block placement, location, replacement, and write policies. Section 17.5 discusses the block placement and location policies. The three policies presented here are the direct, associative, and set-associative mappings. Block replacement policies are described in the following section. Two replacement policies are discussed in this section. Two write policies and their advantages and disadvantages are discussed in Section 17.7. The space overhead introduced by the direct, associative, and set-associative mapping functions is discussed in Section 17.8. The following section presents some mapping function examples.

The next two sections describe various types of misses and cache types. Section 17.12 gives details on cache implementations of Pentium, PowerPC, and MIPS processors. A summary of cache operations and design issues is given in the next two sections. The chapter ends with a summary.

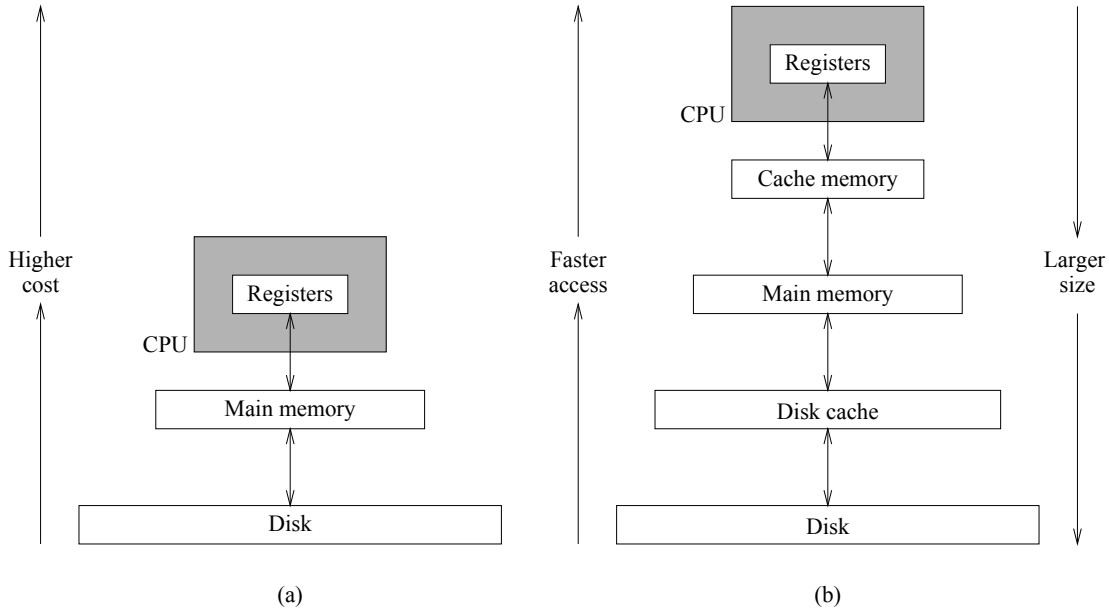


Figure 17.1 A memory hierarchy of computer systems.

17.1 Introduction

A memory hierarchy of computer systems is shown in Figure 17.1a. At the top we have registers within the CPU that provide very fast access to data. But only a few registers are available. For example, the Pentium has eight 32-bit general-purpose registers. PowerPC and MIPS processors provide 32 registers. As we move down the hierarchy, we encounter slower memory components: main memory and disk. For a variety of reasons, slower memories are cheaper. Because of this cost advantage, computer systems have larger amounts of slower memories: in general, as shown in Figure 17.1a, the slower the memory, the larger the capacity.

There is an order of magnitude difference in the access times of registers and main memory. Similarly, disk access times are several orders of magnitude higher than main memory access times.

Another factor is the speed difference between processors and main memory (mainly DRAM). This difference has been increasing with time as processor speeds are improving at a much faster rate than the memory speeds. Currently, processors seem to improve at 25% to 40% per year, whereas memory access times are improving at about 10% per year. For example, 2.4 GHz Pentium processors are available now, whereas only 300 MHz processors were available just a few years back.

Ideally, we would like to have a large fast memory without paying for it. The cache memory is a small amount of fast memory that is tactically placed between two levels of memory in the hierarchy to bridge the gap in the access times. Based on the hierarchy shown in Figure 17.1a,

there are two places we can use the cache concept: between main memory and CPU registers (cache), and between disk and main memory (disk cache) as shown in Figure 17.1*b*. In this chapter, we focus on the cache between the main memory and the CPU.

17.2 How Cache Memory Works

The cache memory is placed between the CPU and main memory as shown in Figure 17.1*b*. Cache memory is much smaller than the main memory. In PCs, for example, main memory is typically in the 128 to 256 MB range, whereas the cache is in the range of 16 to 512 KB. Cache memories are implemented using static RAMs whereas the main memory uses DRAMs.

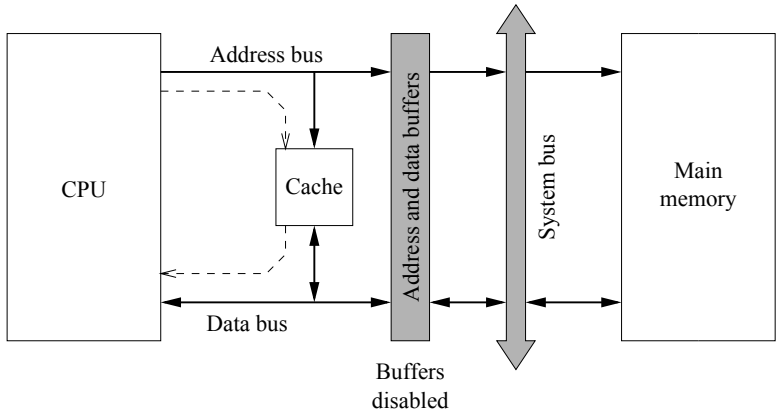
The principle behind the cache memories is to prefetch the data from the main memory before the processor needs them. If we are successful in predicting the data that the processor needs in the near future, we can preload the cache and supply those data from the faster cache. It turns out that predicting the processor future access requirements is not difficult owing to a phenomenon known as *locality of reference* that programs exhibit. We discuss this locality in the next section.

To understand how the cache memory works, we consider two memory operations: read and write. For each operation, there are two possible scenarios: the data are present in the cache (hit) or not present (miss).

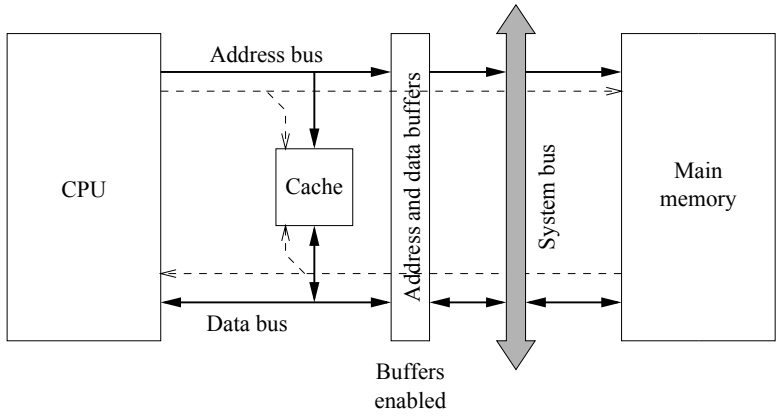
Figure 17.2 shows how memory read operations are handled to include the cache memory. When the data needed by the processor are in the cache memory ("read hit"), the requested data are supplied by the cache (see Figure 17.2*a*). In this case, we can disable the address and data buffers. Since the system bus and main memory are not involved, they can be used for some other purpose such as DMA operations. The dashed line indicates the flow of information in the case of a read hit. In the case of a read miss, we have to read the data from the main memory. While supplying the data from the main memory, a copy is also placed in the cache as indicated by the bottom dashed lines in Figure 17.2*b*. Reading data on a cache miss takes more time than reading directly from the main memory as we have to first test to see if the data are in the cache and also consider the overhead involved in copying the data into the cache. This additional time is referred to as the *miss penalty*.

Performance of cache systems can be measured using the *hit rate* or *hit ratio*. The hit ratio is the fraction of the memory accesses found in the cache. The *miss rate* or *miss ratio* is the fraction of memory accesses not found in the cache, which is equal to (1 − hit ratio). The time required to access data located in the cache is called *hit time*. Hit time includes the time required to determine if the data are in the cache.

The write operation actions are shown in Figure 17.3. In the case of a write miss, we need to update only the main memory copy of the data as there is no copy of them in the cache. However, when there is a write hit, we have two options: update only the cache copy or update both cache and main memory copies of the data. Both options are used in practice. In Figure 17.3*a*, we have shown the second option (called the write-through policy). Cache write policies are discussed in Section 17.7.



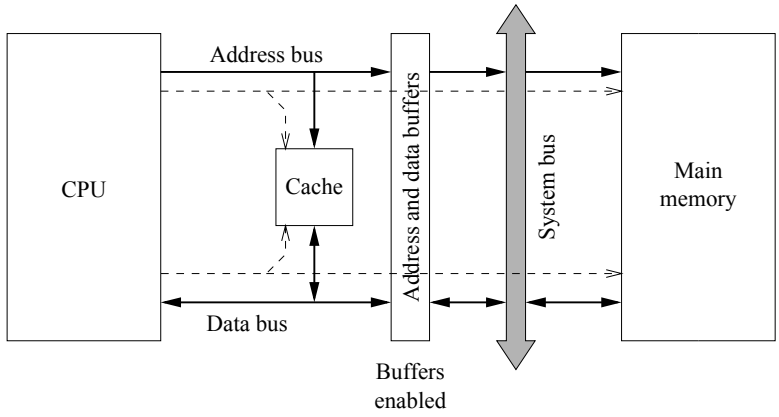
(a) Read hit



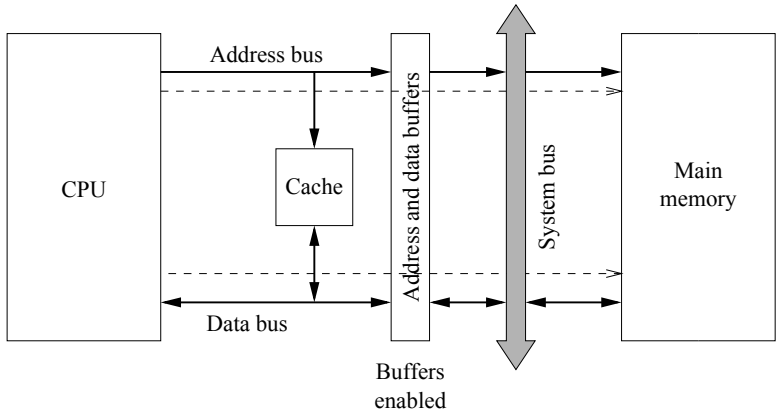
(b) Read miss

Figure 17.2 Details of cache read operation.

In our discussion, we have glossed over several complex questions that are at the heart of cache memory design: How can one detect if the data items are in the cache? If not, when do we move the data into the cache? How many data items should we move? What happens if there is no space in the cache? How do we predict which data the processor is likely to request in the near future? The rest of the chapter discusses these questions.



(a) Write hit



(b) Write miss

Figure 17.3 Details of cache write operation.

17.3 Why Cache Memory Works

In the last section, we have given a high-level view of the basic cache operations. Why should this scheme lead to improved performance? Let us suppose that we are executing the following matrix operation, which adds constant K to all elements of an M $\times$ N matrix **X**. Let us suppose that each element is a `double` data type requiring eight bytes.

```
for(i=0; i<M; i++)
    for(j=0; j<N; j++)
        X[i][j] = X[i][j] + K;
```

The statement `X[i][j]=X[i][j]+K` is executed M * N times. If we place the instructions for this loop in the cache, we avoid accessing them from the main memory. We can expect performance benefit in this case because we load the instructions once into the cache and use them repeatedly. *Repetitive use* of cached data and instructions is one of the two main factors that makes a cache memory work.

The other factor is the predictive loading of the cache with data and instructions before the processor actually needs them. Predictive loading is often implemented by moving more contiguous locations in memory than needed when there is a read miss. This type of predictive loading helps improve performance due to (i) the ability to mask the delay of slow main memory, and (ii) the efficiency with which a block of contiguous locations can be fetched as opposed to accessing each location individually.

Continuing with our example, there will be a read miss when the matrix element X[0][0] is accessed. We have to access the main memory to get X[0][0], which will also be placed in the cache. Now suppose that we not only get X[0][0] but also the next three elements in the first row (i.e., get four elements X[0][0], X[0][1], X[0][2], and X[0][3]) for a total of 32-byte data movement between the main memory and cache. Then, we can avoid accessing the memory for the next three iterations. Notice that, in this example, the data brought into the cache are not reused. Thus, we do not derive any benefit from reuse as we did for the instructions. However, we still get the benefit of using the cache because our prediction works well for the example.

To give a quantitative idea of how much difference this prefetching makes, we present the plot in Figure 17.4. The y-axis gives the execution time on a 300 MHz Pentium to add a constant to a square matrix of the size given on the x-axis. The row-order line gives the execution time when the matrix is accessed row-by-row. For example, the code we discussed before uses row-order access. If we interchange the two `for` statements, the matrix is accessed on a column-by-column basis. Execution time of column-order access is more as shown in Figure 17.4.

What is the reason for this performance difference? With column-order access, our prediction and hence the utility of preloading data into the cache goes haywire because C stores the matrix in row-major order (i.e., row-by-row). If you are using FORTRAN instead of C, column-order access benefits from the prediction as FORTRAN stores the matrix in column-major order. These data illustrate that, even though our program does not reuse the data, we get substantial benefit in predicting and preloading the data.

In summary, applications that exhibit one or both of the following characteristics will benefit from the cache: repetitive use and predictive preloading. It turns out that most programs exhibit these two characteristics to a varying degree.

Program referencing behavior has been extensively studied. These studies have led to the *principle of locality* in the referencing behavior of programs. Locality of reference states that programs tend to reference only a subset of the data and instructions repeatedly during a certain period of time. For example, if the code fragment shown before is part of a larger program, the processor executes the statement `X[i][j]=X[i][j]+K` repeatedly and accesses matrix **X** during this time.

There are two principles of locality: locality in space (*spatial locality*) and locality in time (*temporal locality*). The spatial locality principle states that programs tend to access data and

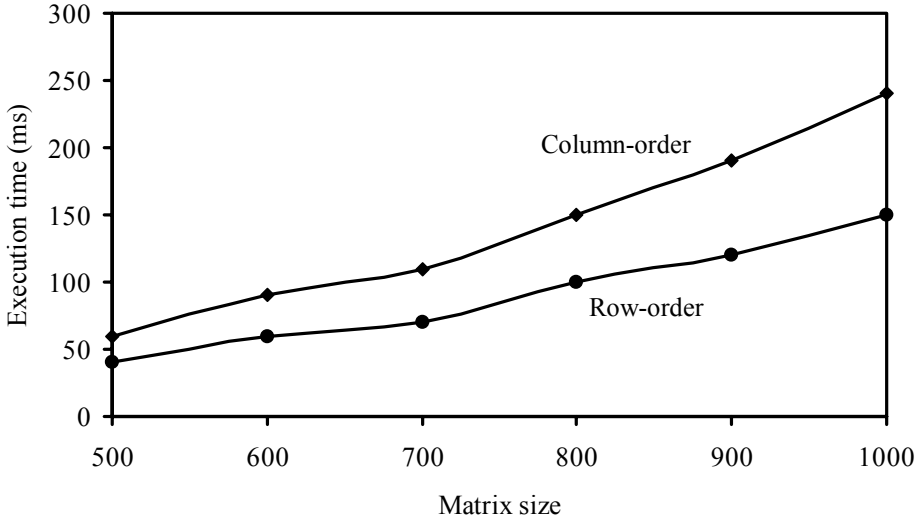


Figure 17.4 Impact of locality on the execution time of matrix addition.

instructions sequentially. This is fairly easy to understand. Most of the time, programs also tend to execute instructions sequentially. Of course, occasionally procedure calls and if-then-else types of conditional statements alter the sequential program execution path. The spatial locality characteristic allows us to successfully perform predictive preloading.

Temporal locality refers to the tendency of programs to repeatedly use a small part of code/data over a certain time. The loop is a prime example that contributes to this behavior. The previous code fragment exhibits temporal locality in referencing the code but not data. Temporal locality helps improve performance because, as we identified earlier, repeated access can be satisfied from the cache.

17.4 Cache Design Basics

We have looked at how the cache memory works and why it improves performance. There are several design issues we need to discuss in making the caches a practical reality. We present the basic design issues in the form of a series of questions that we have to answer when we trace the actions of a processor in executing a program.

When the processor starts executing a program, the cache is cold (i.e., contains garbage). The first instruction and the associated data are not expected to be in the cache. But no special consideration is given to the first instruction. Thus, we encounter our first problem: How do we determine if the instruction/data that the processor is looking for is in the cache? If it is in the cache, how do we find its address in the cache memory? We can reword this question as: Given the address of a memory word, how do we find the cache address? The only way we can get this address is by searching the cache memory. Of course, we have to do this search

in hardware because of the speed considerations. The complexity of the search (and hence the hardware cost) depends on the type of function used to map memory addresses to cache memory locations. There are three mapping functions: direct, associative, and set-associative mapping. A detailed discussion of these mapping functions is deferred to the next section.

If the answer to the previous question is no (i.e., not in the cache), the processor has to get the word from the main memory; along the way, the word is also copied to the cache. This leads to several questions. How many bytes do we transfer as a result of the read miss? Do we just transfer only the number of bytes for which the processor is looking? Our previous discussion on predictive loading suggests that we should load more than what the processor requests. We can make the number of bytes transferred vary from read miss to read miss to take care of variable instruction lengths and data types. However, such a strategy causes implementation problems. To simplify implementation, a fixed number of bytes B are transferred in response to every read miss. For example, in the Pentium and PowerPC, 32 bytes are transferred.

Implementation can be simplified further by restricting the starting location of these fixed-size blocks to addresses that are a multiple of the block size B. With this restriction, we can use the least significant $b = \log_2 B$ bits as the byte offset into the selected block. The remaining higher-order bits of the memory address are used as the block number. The cache is also divided into blocks of B bytes; cache blocks are called *cache lines*. Figure 17.5 shows how the main memory is partitioned into blocks. This example assumes that the block size B is 4 bytes. The 16-bit memory address is partitioned into a 2-bit byte offset and 14-bit block number. Data are transferred in blocks between the main memory and cache, whereas individual words are transferred between the cache and processor, as shown in Figure 17.6.

The next question is: Where in the cache do we place the incoming block? This is the assignment problem; cache line allocation depends on the mapping function used. We discuss three mapping functions in the next section.

After the assignment of a cache line (or, as we show in the next section, a set of cache lines) we can place the memory block in the assigned cache line, if the cache slot is free. If the assigned cache slot is not free, we have to decide which cache line is a candidate for replacement. If the mapping function specifies a particular cache line, there is no choice. However, if the mapping function specifies a set of cache lines, we need a replacement policy to select a cache line from this set. Replacement policies are described in Section 17.6.

The sequence of actions taken on a read cycle is summarized in Figure 17.7. The actions on a write cycle are similar. However, we have to answer an additional question: How are the writes handled? We have briefly mentioned that there are two basic alternatives: write-back or write-through. We look at these two policies and their implications in Section 17.7.

17.5 Mapping Function

The mapping function determines how memory blocks are mapped to cache lines. This function sometimes is also referred to as the *hashing function*. There are three types of mapping functions:

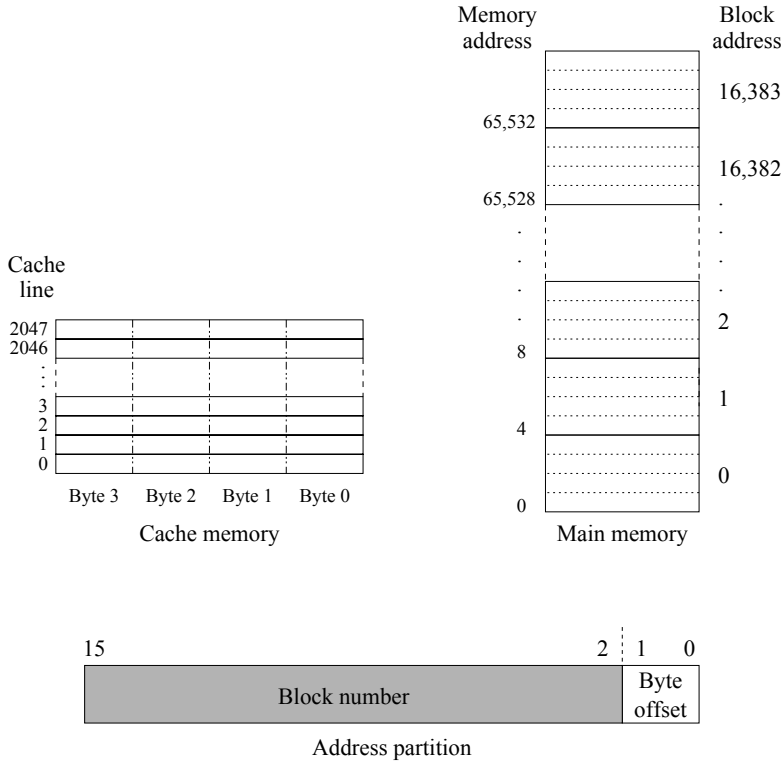


Figure 17.5 Main memory and cache memory are divided into blocks of equal size: With the blocks placed as shown here, we can use the lower-order bits for byte offset and higher-order bits for the block number. This example uses a 64 KB main memory and an 8 KB cache.

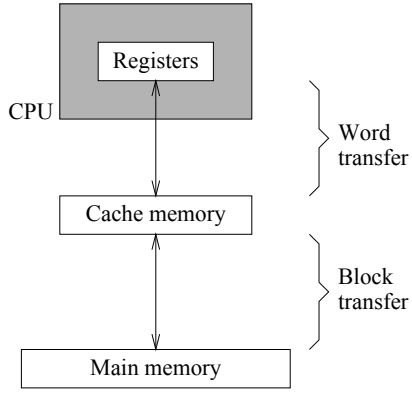


Figure 17.6 Data transfer between the top three levels of the memory hierarchy.

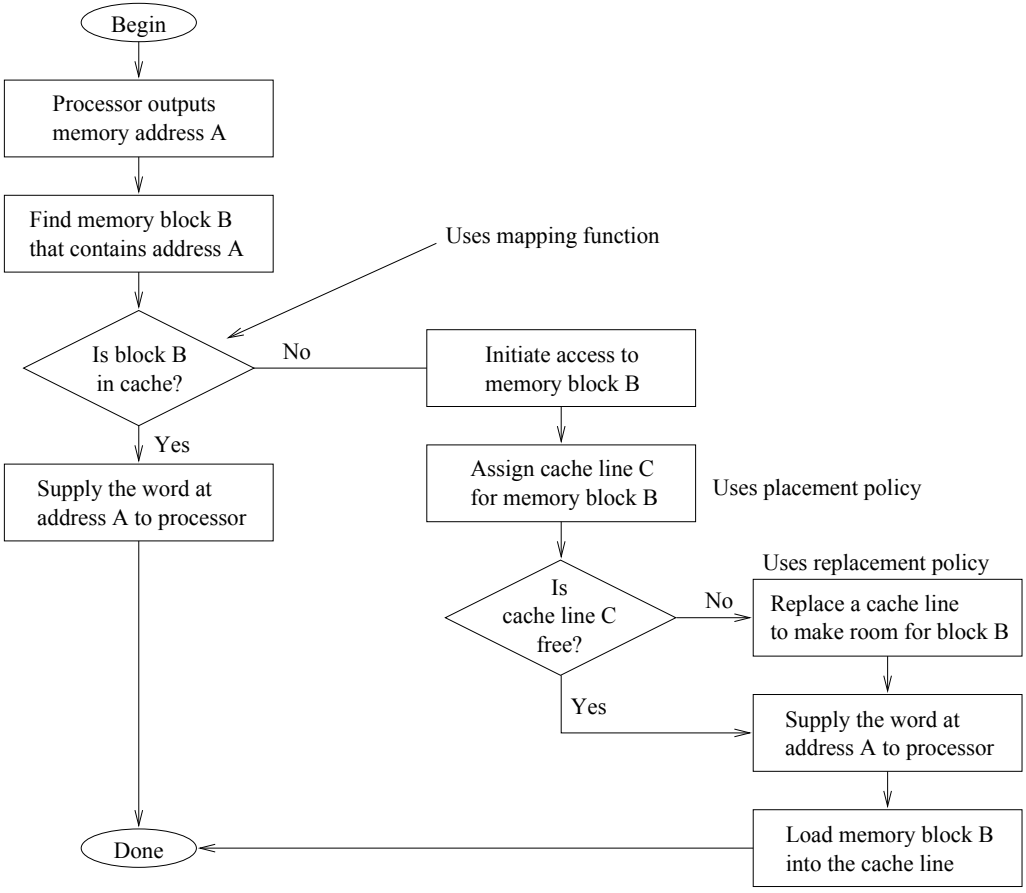


Figure 17.7 Cache operations for a processor read cycle.

1. *Direct Mapping:* Specifies a single cache line for each memory block;
2. *Set-Associative Mapping:* Specifies a set of cache lines for each memory block;
3. *Associative Mapping:* No restrictions; any cache line can be used for any memory block.

Direct mapping is relatively easy to implement but it is less flexible and can cause performance problems. Associative mapping allows maximum flexibility but is very complex to implement. Because of this complexity and the associated cost, associative mapping is not used in practice. The set-associative mapping strikes a compromise between the direct and associative mapping functions. Practical implementations more often use this function. The following subsections discuss these three functions in detail.

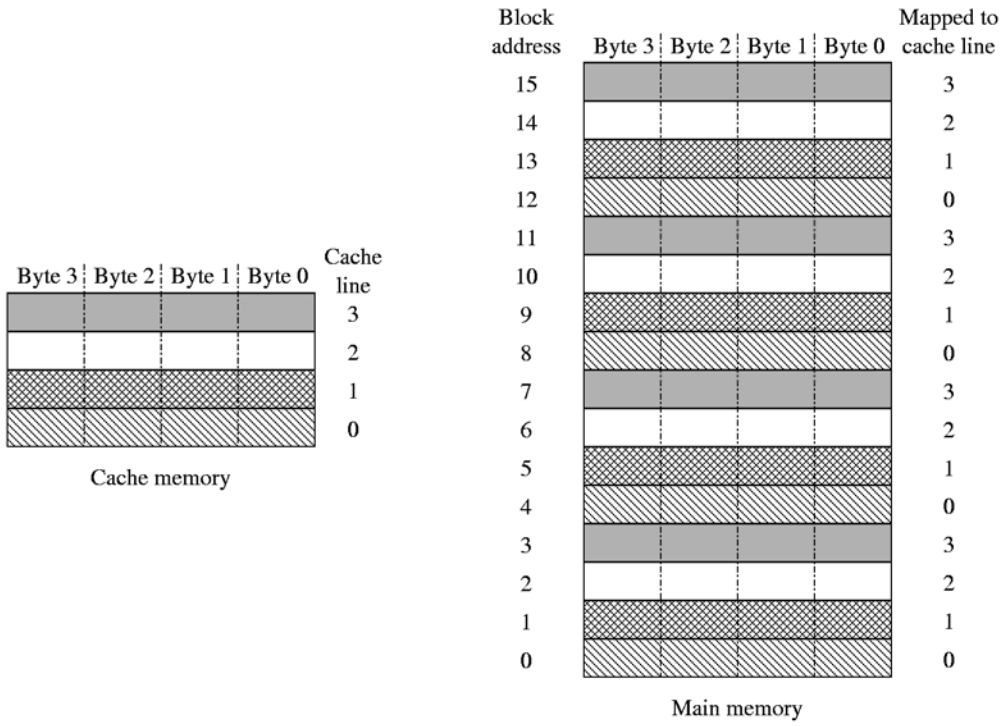


Figure 17.8 A direct mapping cache example: The main memory and cache are assumed to be 64 bytes and 16 bytes, respectively. The mapping of memory blocks to cache lines is shown by appropriate shading. For example, main memory blocks 0, 4, 8, and 12 are mapped to cache line 0.

17.5.1 Direct Mapping

The direct mapping function specifies a single cache line into which a memory block should be placed. A modulo function is used to achieve this mapping. If C is the number of cache lines, a memory block i is mapped to cache line $c = i$ **mod** C. Figure 17.8 shows how direct mapping works on a simple system with 64 bytes of main memory and 16 bytes of cache memory. We are using a block size of 4 bytes. Therefore, the number of cache lines is 16/4 = 4. Now, it is simple to find the cache line to which a memory block is mapped. For example, memory block 11 is mapped to cache line 3 (given by 11 **mod** 4).

Figure 17.9 shows details on the kind of information the cache memory should maintain. The address is divided into three components:

- Least significant b bits are used as the byte offset into a block. If the block size is B bytes, the number of bits for the byte offset is given by

$$b = \log_2 B.$$

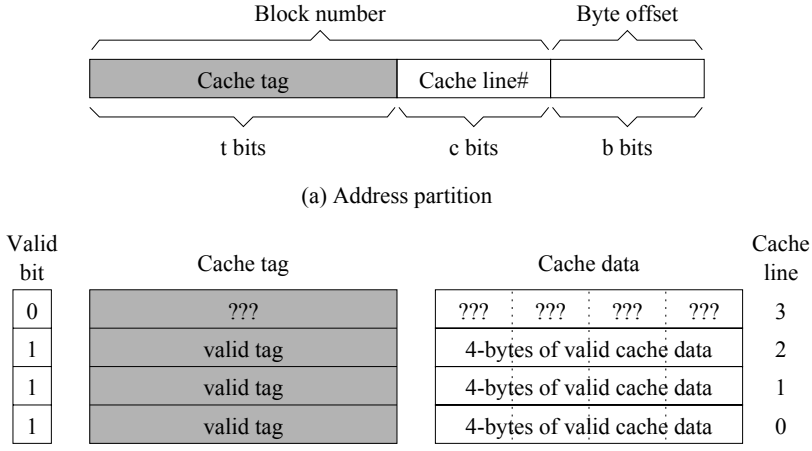


Figure 17.9 Details of address partition and cache memory organization for direct-mapped caches.

- The next c bits are used to identify a cache line. If the number of cache lines is C, we can get c as

$$c = \log_2 C.$$

- The remaining higher-order t bits are used as the cache tag. As we show next, cache memory would have to store this tag information for each cache line in order to identify the memory block stored in the cache line.

Example 17.1 *Suppose that we are considering a Pentium-like processor with 4 GB of memory address space and 256 KB of direct-mapped cache. Assuming that the block size and hence the cache line size is 32 bytes, find the number of bits for the byte offset b, the number of bits for the cache line c, and the tag size.*

The main memory address space of 4 GB implies 32-bit addresses. Since the system uses 32-byte blocks, we need $\log_2 32 = 5$ bits for the byte offset. To find the number of bits for the cache line, we need to know the number of cache lines. This is obtained by dividing the cache capacity in bytes by the block size. In our example, 256K/32 gives 8192 (8 K) lines. Thus, we need $\log_2 8192 = 13$ bits for the cache line. The remaining higher-order 14 bits $(32 - 13 - 5)$ represent the tag field. □

Cache implementations maintain three pieces of information as shown in Figure 17.9:

- *Cache Data:* These are the actual data copied from the mapped memory block.

- *Cache Tag:* Since more than one memory block is mapped to a cache line, we need to store the tag field for each cache line in order to know the address of the actual block stored. The size of the tag field t is equal to $\log_2 N$, where N is the number blocks mapped to each cache line.
- *Valid Bit:* We also need to store information on whether a cache line contains a valid block. This information is necessary so that we do not treat garbage as valid data. A single bit is sufficient to maintain this information. We use 1 to represent a valid cache line and 0 for an invalid cache line.

Direct mapping results in a simple implementation of the cache. To find if a memory block is in the cache, use the modulo function to find the cache line for the block. Then check if the corresponding valid bit is 1. If so, compare the tag value stored at the cache line with the higher-order t bits of the memory block address. If there is a match, the required block is in the cache; otherwise, there is a cache miss. Thus, answering the question, "Is the block cached?" involves an arithmetic operation and a compare operation.

If the block is not in the cache, we read the memory block and store it at the cache slot indicated by the modulo operation. If that slot is currently occupied by another memory block, we replace it. Action taken to replace a cache line depends on the type of write policy used and is discussed in Section 17.7. Thus, placement and replacement policies are straightforward: a single cache line is specified to place the memory block in the case of a cache miss.

What is the problem with direct mapping? The simplicity of the direct mapping function, although desirable from the implementation viewpoint, is also its disadvantage. By mapping each memory block to a single cache line, we lose flexibility that can lead to performance degradation. The following example illustrates this drawback:

Example 17.2 *Suppose that the reference pattern of a program is such that it accesses the following sequence of blocks: 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4. Find the hit ratio with a direct-mapped cache of four cache lines.*
The three blocks accessed by the program—0, 4, and 8—are all mapped to cache line 0. When block 0 is accessed for the first time, it is placed in cache line 0. When the program accesses block 4 next, it replaces block 0 in cache line 0. You can continue this process and see that every block access leads to a cache miss as shown in Table 17.1. Thus the hit ratio is zero. □

This example describes the worst-case behavior, where each access results in a miss. This scenario where a cache line is replaced before it is referenced again is called *cache thrashing*.

Example 17.3 *Suppose the sequence of main memory blocks accessed is 0, 7, 9, 10, 0, 7, 9, 10, 0, 7, 9, 10. Find the hit ratio in a direct-mapped cache with a cache size of four cache lines.*
This reference pattern shows the best behavior. There will only be four cache misses. After these four blocks have been loaded into the cache, the remaining eight references can obtain data from the cache as shown in Table 17.2. Thus the hit ratio is 8/12 or about 67%. For this example, this is the maximum hit ratio one can obtain. □

Table 17.1 Direct-mapped cache state for Example 17.2

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
8	Miss	Block 8	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???
0	Miss	Block 0	???	???	???
4	Miss	Block 4	???	???	???

Table 17.2 Direct-mapped cache state for Example 17.3

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
7	Miss	Block 0	???	???	Block 7
9	Miss	Block 0	Block 9	???	Block 7
10	Miss	Block 0	Block 9	Block 10	Block 7
0	Hit	Block 0	Block 9	Block 10	Block 7
7	Hit	Block 0	Block 9	Block 10	Block 7
9	Hit	Block 0	Block 9	Block 10	Block 7
10	Hit	Block 0	Block 9	Block 10	Block 7
0	Hit	Block 0	Block 9	Block 10	Block 7
7	Hit	Block 0	Block 9	Block 10	Block 7
9	Hit	Block 0	Block 9	Block 10	Block 7
10	Hit	Block 0	Block 9	Block 10	Block 7

Table 17.3 Fully associative cache state for Example 17.4

Block accessed	Hit or miss	Cache line 0	Cache line 1	Cache line 2	Cache line 3
0	Miss	Block 0	???	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
8	Miss	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
8	Hit	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
4	Hit	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
4	Hit	Block 0	Block 4	Block 8	???
0	Hit	Block 0	Block 4	Block 8	???
4	Hit	Block 0	Block 4	Block 8	???

17.5.2 Associative Mapping

This function is also called the *fully associative mapping* function to distinguish it from the set-associative function. This mapping does not impose any restriction on the placement of blocks in cache memory. A memory block can be placed in any cache line. This offers maximum flexibility, which we demonstrate by using the reference pattern of Example 17.2.

Example 17.4 *Consider the reference pattern of Example 17.2 that accesses the sequence of blocks 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4. Assuming that the cache uses associative mapping, find the hit ratio with a cache size of four cache lines.*

In fully associative mapping, the incoming block can take any free cache line. In Table 17.3, we use a FIFO allocation. Block 0 is placed in cache line 0, block 4 is placed in the next available cache line, and so on. Thus, after three misses to load the blocks 0, 4, and 8 initially, the remaining nine references can be read from the cache. This gives us a hit ratio of 75%. This is in contrast to the hit ratio of 0% obtained with direct mapping. Because these three misses are compulsory misses, this is the highest hit ratio we can get for this reference pattern. □

With associative mapping, we divide the address into two components as shown in Figure 17.10:

- The least significant b bits are used for byte offset as in the direct mapping scheme;
- The remaining bits are used as the tag field.

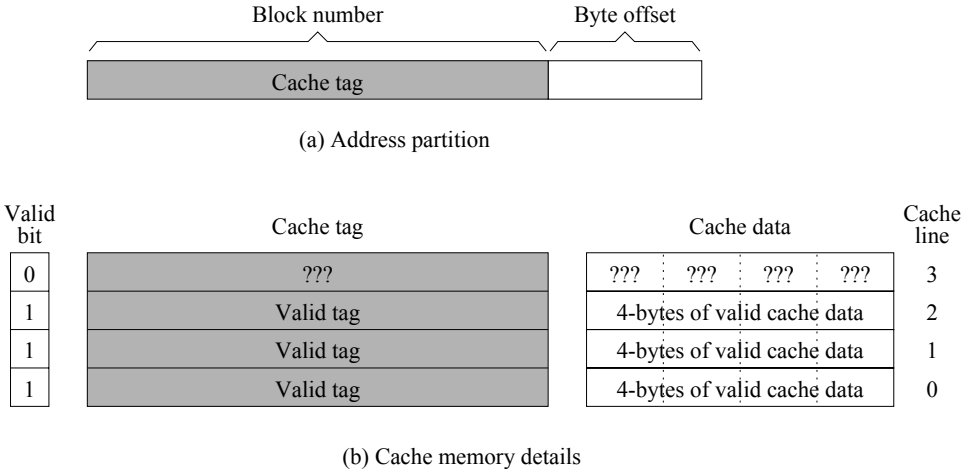


Figure 17.10 Details of address partition and cache memory organization for fully associative mapping

Thus, compared to the direct mapping scheme, the tag field is longer. This is not the major problem with this mapping scheme. The major drawback is the location of a block in the cache. Since a block can be in any cache line, we have to search all tag fields in parallel to locate a block. This means we need hardware to do 2^c comparisons, where 2^c is the number of cache lines.

Figure 17.11 shows details of the address match logic. It uses 2^c comparators; each compares a tag value from the cache tag memory to the tag field of the address. The outputs of these comparators are encoded to generate a binary number that identifies the cache line, if there is a match. Because the encoder will output 0 if there is no match or if the block is in cache line 0, we need to generate a match found output. This signal can be generated by ORing the outputs of the comparators as shown in Figure 17.11.

The sequence of steps involved in accessing an associative-mapped cache is shown in Figure 17.12. The tag field from the address is compared by the address match logic. If there is a match, the cache controller supplies the data from the cache data memory. These interactions are shown by dashed lines.

17.5.3 Set-Associative Mapping

Set-associative mapping is a compromise between direct and associative mapping. It divides the cache lines into disjoint sets. Mapping of a memory block is done as in direct mapping, except that it maps the block to a set of cache lines rather than to a single cache line. The block can be placed in any cache line within the assigned set as in the associative mapping. Set-associative mapping reduces the search complexity to the number of cache lines within a set. Typically, small set sizes—2, 4, or 8—are used. An example mapping is given in Figure 17.13.

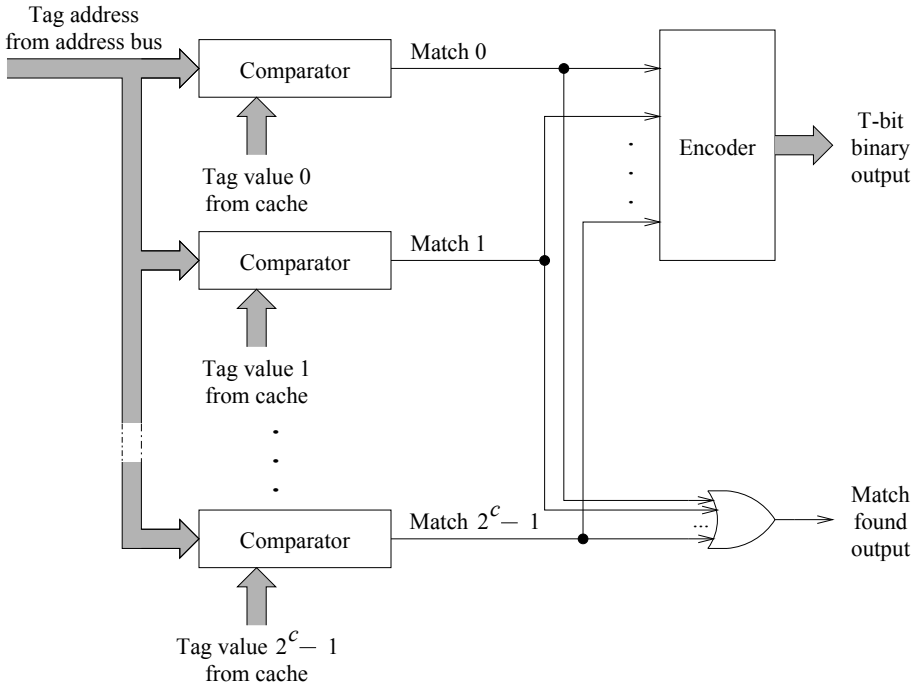


Figure 17.11 Details of address match logic for fully associative mapping.

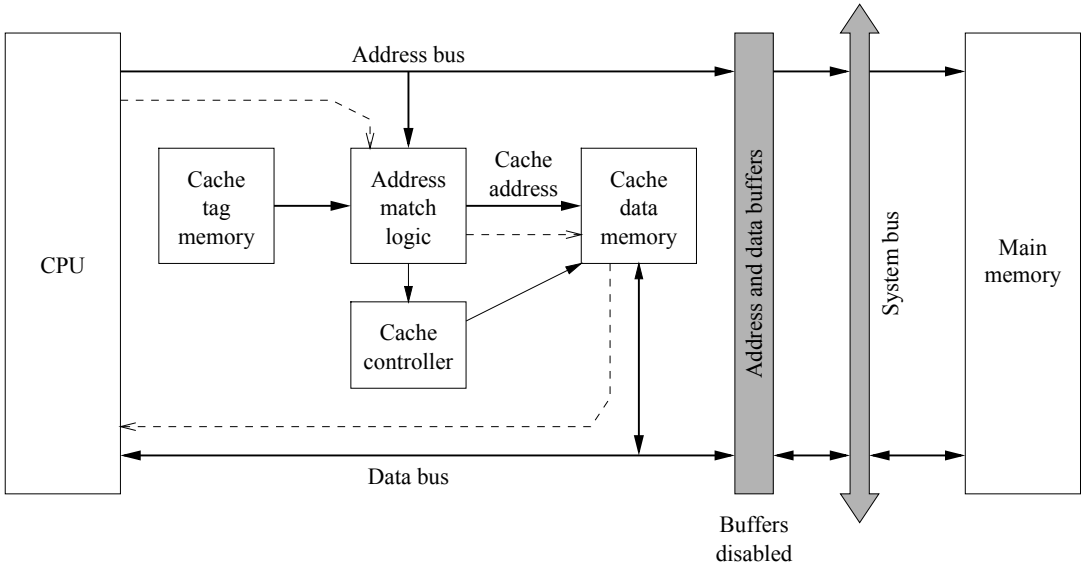


Figure 17.12 A fully associative cache with its address match logic.

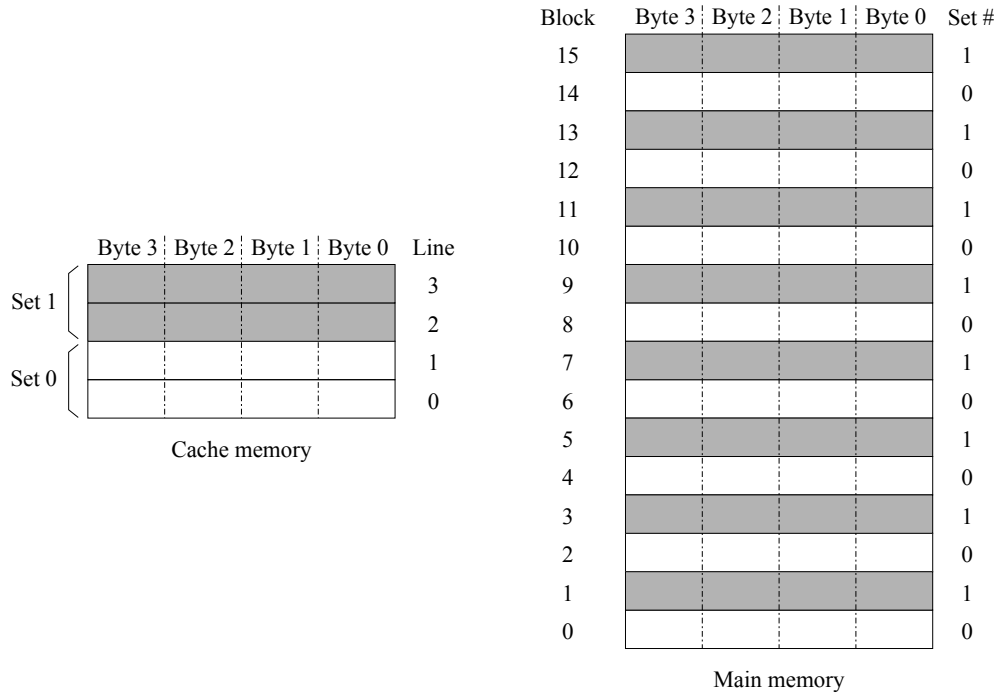


Figure 17.13 Set-associative mapping function details: The main memory is 64 bytes and the cache is 16 bytes in size. Cache line size is 4 bytes. With these parameters, the set-associative function maps all even-numbered blocks to set 0 (shown as white) and all odd-numbered blocks to set 1 (shown as gray).

Example 17.5 *Let us consider the reference pattern of Example 17.2 that accesses the sequence of blocks 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4. Assuming that the set size is 2, find the hit ratio with a cache size of four cache lines.*

Since the cache size is four lines and the set size is 2, we have two sets. Therefore, all blocks with an even number are mapped to set 0, and odd numbered blocks to set 1. In the example, all three blocks—0, 4, and 8—are mapped to set 0. The cache activity is shown in Table 17.4.

The first reference to block 8 presents an interesting situation. This even-numbered block is mapped to set 0. However, set 0 is full (with blocks 0 and 4). Now we need to make a decision as to which block should be replaced. This is determined by the replacement policy in effect. Replacement policies are discussed in the next section. For now, we use a policy that replaces a block that has not been accessed recently. Between blocks 0 and 4, block 0 has been accessed more recently. Thus, we replace block 4 with block 8. A similar situation arises when block 4 is accessed later.

Using set-associative mapping gives us $8/12 \approx 67\%$ as the hit ratio. As you can see, this hit ratio is better than that of the direct-mapped cache but lower than that of the fully associative-mapped cache. □

Table 17.4 Set-associative cache state for Example 17.5

Block accessed	Hit or miss	Set 0		Set 1	
		Cache line 0	Cache line 1	Cache line 0	Cache line 1
0	Miss	Block 0	???	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
8	Miss	Block 0	Block 8	???	???
0	Hit	Block 0	Block 8	???	???
8	Hit	Block 0	Block 8	???	???
0	Hit	Block 0	Block 8	???	???
4	Miss	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
4	Hit	Block 0	Block 4	???	???
0	Hit	Block 0	Block 4	???	???
4	Hit	Block 0	Block 4	???	???

Set-associative mapping partitions the address into three components as shown in Figure 17.14:

- The least significant b bits specify the byte offset as in the other two mapping functions.
- The next s bits identify a set. The number of bits for the s field is given by $s = \log_2 S$ where S is the number of sets.
- The remaining higher-order bits are used as the tag, as in the other two mappings.

17.6 Replacement Policies

Replacement policy comes into action when there is no place in the cache to load the memory block. The actual policy used depends on the type of mapping function employed. Direct mapping does not require a special replacement policy. In this case, the mapped cache line should be replaced if that cache line is not free. There is no choice.

Several replacement policies have been proposed for the other two mapping functions. Since the fully associative function is not used in practice, we focus on the set-associative mapping. In this mapping function, there is a choice as to which cache line should be replaced in a given set. Ideally, we want to replace a cache line that will not be used for the longest time in the immediate future. How can we predict future referencing behavior? We, of course, fall back on the principle of locality. That is, we use recent history as our guide to predict future references.

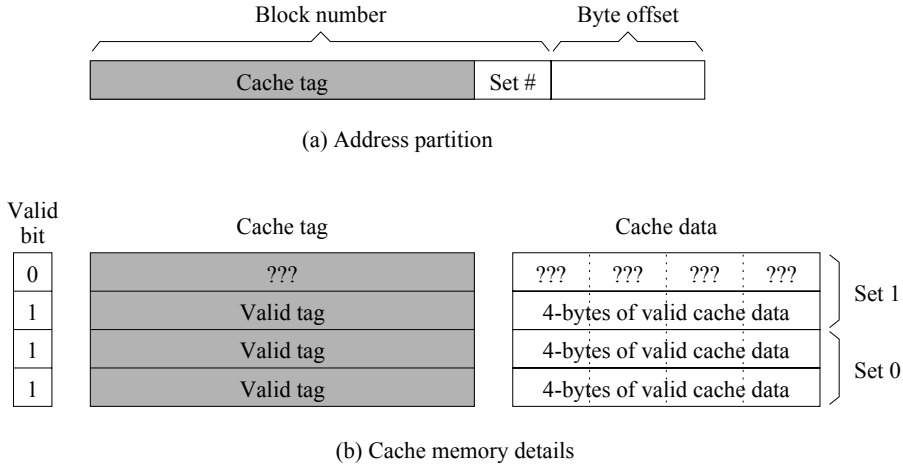


Figure 17.14 Details of address partition and cache organization of a set-associative mapped cache.

This leads us to what is known as the *least recently used* (LRU) policy. This policy states that we should replace the cache line that has not been accessed for the longest period. We have already used this policy in Example 17.5.

In practice, implementation of LRU is expensive for set sizes larger than four. For set sizes of two, it is easy to implement. We need only one bit to store the identity of the least recently used cache line in the set. However, as the degree of associativity increases, we need more bits. To give you an idea, consider a 4-way associative cache. To maintain true LRU information requires keeping information on $4! = 24$ combinations.

```
0123    1023    2013    3012
0132    1032    2031    3021
0213    1203    2130    3102
0231    1230    2103    3120
0312    1302    2301    3210
0321    1320    2310    3201
```

Thus, we need 5 bits to keep this information. If the degree of associativity is 8, we need 8! = 40,320 states, which requires $\lceil \log_2 8! \rceil = 16$ bits to store the LRU information.

Practical implementations typically use an approximation to the true LRU policy. The basic idea behind these "pseudo-LRU" policies is to partition the sets into two groups and maintain the group that has been accessed more recently. This requires only one bit of information. Each group is further divided into subgroups as shown in Figure 17.15. This scheme requires only $W - 1$ bits, where W is the degree of associativity. This is in contrast to $\lceil \log_2 W! \rceil$ bits required to implement the true LRU policy. For the 8-way associative mapped cache used by

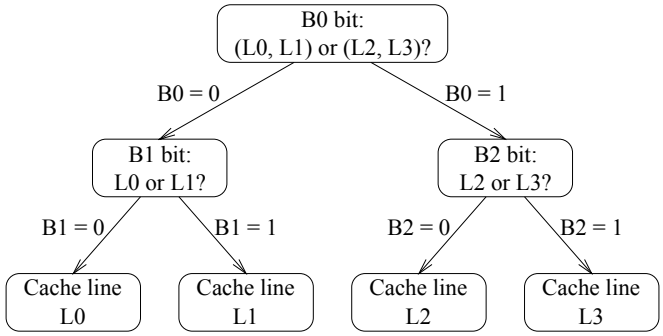


Figure 17.15 Pseudo-LRU implementation for a four-way associative cache: A similar scheme is used in the Intel i486 and PowerPC processors.

the PowerPC, pseudo-LRU implementation requires only 7 bits of information as opposed to 16 bits for the true LRU implementation.

Random replacement is an alternative to the LRU policy, which eliminates the need to maintain the state information. As the name suggests, a cache line is randomly selected for replacement. As the degree of associativity increases, random replacement becomes attractive. Even for smaller associativity, performance of random replacement approximates very closely that of the true LRU replacement. For example, the miss ratio might increase by about 10% when the random replacement is used with a two-way associative cache. For larger caches, the difference in the miss ratio between LRU and random policies becomes marginal [32]. In fact, when the LRU replacement is approximated, random replacement even performs better in some cases.

Other replacement policies of theoretical interest are the first-in-first-out (FIFO) and least frequently used (LFU). FIFO policy replaces the block that has been in the set for the longest time. FIFO policy can be implemented using a circular buffer. LFU replaces the block in the set that has been referenced the least number of times. Implementation of LFU requires a counter associated with each cache line. These policies are more appropriate for managing virtual memory, which is discussed in the next chapter.

17.7 Write Policies

So far in our discussion we have focused on reading data from the memory and how cache can handle this activity. If the processor is updating a variable or data structure, how do we handle this? We have two copies of the data: a main memory copy and a cached copy.

There are two basic write policies: *write-through* and *write-back*. In the write-through policy, every time the processor writes to the cache copy, it also updates the main memory copy. Thus, in this policy, both cache and memory copies are consistent with each other. Figure 17.3a shows the write-through policy in action.

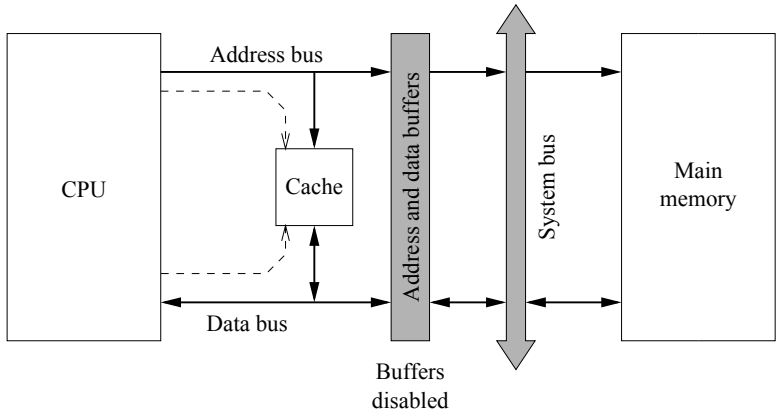


Figure 17.16 Write hit operation in a write-back cache.

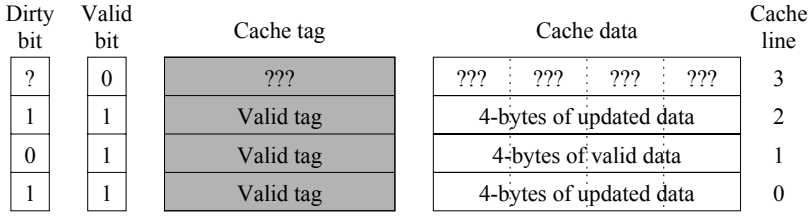


Figure 17.17 A typical write-back cache uses a dirty bit to indicate whether the cache line has been updated.

The write-back policy, also called the *copy-back* policy, updates only the cache copy (Figure 17.16). When do we update the main memory copy? This update is done at the time of replacement. Therefore, in a write-back cache, we have to first write the block back to main memory before loading the new memory block. This update is not required in the write-through policy. Thus, with the write-back policy, line replacements take twice as long as would a line replacement in a write-through cache.

We can reduce the number of writes in the write-back policy by noting that if a cache line has not been updated by the processor, it does not have to be written back to the memory. This would avoid unnecessary writes to the memory. Most write-back caches use this scheme, which is implemented by associating a bit (called a *dirty bit* or *update bit*) with each cache line. This bit indicates whether the cache copy has been updated. If the dirty bit is set for a cache line, that block has to be written back to memory before replacing it. Thus, in a write-back cache, there are two bits associated with each cache line—a valid bit and a dirty bit—as shown in Figure 17.17.

In Figure 17.17, cache lines 0 and 2 have been updated and thus have a more up-to-date copy of the block. If either of these two blocks is replaced, the corresponding memory block should be updated first. The dirty bit significantly improves performance of a write-back cache, as programs tend to read more often than write (typically, writes are about 15% of all memory references).

Another popular technique to reduce the write traffic is to use *write buffers*. Buffered writing is particularly useful for write-through caches. In buffered writing, writes to main memory are buffered and written to the memory at a later time. Buffered write allows write combining by caching multiple writes in the buffer; it therefore reduces the bus traffic. This technique is especially useful in cache designs that allow noncaching of write blocks. For example, the Pentium uses a 32-byte write buffer. This buffer is written to the memory at several trigger points. One of the trigger points is the buffer full condition.

What are the advantages and disadvantages of the two write policies? A good thing about the write-through policy is that both cache and memory copies are always consistent. This is important in situations where the data are shared among several processors as in a multiprocessor system. However, the drawback is that it tends to waste bus and memory bandwidth. To see why this is the case, consider a loop to sum an array of 10,000 elements. Assuming that the variable `sum` is not mapped to a register, a write-through cache updates the main memory copy of `sum` during each iteration. We could avoid all these writes by simply updating the memory copy of `sum` with the final value after the loop has terminated. This is effectively what a write-back cache does. The disadvantage of a write-back cache is that it takes longer to load a new cache line. Furthermore, its implementation requires additional dirty bit information.

17.8 Space Overhead

The three mapping functions also introduce additional space requirements to store the tag field. This overhead decreases with the decreasing degree of associativity. As the cache line size increases, this tag field overhead becomes relatively smaller. The following examples clearly illustrate these points:

Example 17.6 *Assume that a Pentium-like processor (with 4 GB of address space) uses a 32 KB direct-mapped cache. Find the total bit capacity required for the cache with a block size of 32 bytes.*

A memory address space of 4 GB implies 32-bit addresses. These 32 bits are partitioned into three groups as shown in Figure 17.9 on page 704. The byte offset is

$$b = \log_2 32 = 5 \text{ bits.}$$

The number of cache lines is 32 KB/32 = 1024 cache lines. Thus, the number of bits required to identify a cache line is

$$c = \log_2 1024 = 10 \text{ bits.}$$

The remaining 17 higher-order bits of the 32-bit address are used for the tag. To calculate the cache bit capacity, we note that each cache line consists of 32 bytes of data, a 17-bit tag field, and a valid bit. The total cache line size in bits is $32 \times 8 + 17 + 1 = 274$ bits. Therefore, the total cache capacity is $1024 \times 274 \approx 34.25$ KB. This represents an overhead of 2.25 KB, which is about 7%. If a dirty bit is used, we need an additional 1 Kbit. □

Example 17.7 *Repeat the above example for a block size of four bytes.*
The new value for the byte offset is $b = 2$ bits. The number of cache lines is 32 KB/4 = 8 K lines. Thus, to identify a cache line in these 8 K lines, we need c = 13 bits. This leaves the tag size as $32 - 13 - 2 = 17$ bits. Notice that the number of bits for these two fields (i.e., $c + a$) is always equal to $\log_2 C$, where C is the cache capacity in bytes. However, the reduced block size has a dramatic effect on the amount of overhead. Each cache line now consists of 32 bits of data, 17 bits of tag, and a valid bit, for a total of 50 bits. Thus the total cache capacity is 8 K $\times$ 50 = 50 KB. This represents an overhead of about 56%. The space requirement goes up by 8 Kbits if the dirty bit is used. □

Example 17.8 *Repeat Example 17.6 assuming a four-way set-associative cache instead of a direct-mapped cache.*
The byte offset field requires 5 bits as in Example 17.6. The number of bits s to identify a set is $\log_2 S$, where S is the number of sets in the cache. Since it is a four-way set-associative cache, the number of sets is

$$S = 32\ KB/(4 * 32) = 256 \text{ sets.}$$

This gives us $s = \log_2 S = 8$ bits. The remaining $32 - 8 - 5 = 19$ bits are used for the tag field. Therefore each cache line is 32 $\times$ 8 bits of data, 19 bits of tag, and a valid bit. The total cache capacity in bits is given by $256 \times 4 \times 276 = 34.5$ KB. This represents an overhead of about 7.8%. □

What happens when we reduce the block size to four bytes? Computing the overhead for 4-byte blocks, we find that the overhead increases to about 62.5%.

What about the fully associative cache? If we use a fully associative cache, the tag field gets even bigger. The following example computes the space overhead of this mapping function.

Example 17.9 *Repeat Example 17.6 assuming a fully associative cache instead of a direct-mapped cache.*
In fully associative mapping, the address is divided into two fields (Figure 17.10). Since the byte offset requires 5 bits for 32-byte blocks, the tag field is $32 - 5 = 27$ bits. In this case, there is an overhead of 28 bits (27 tag bits and a valid bit) for every cache line with 32*8 bits of data. Thus, it represents an overhead of about 11%. □

Table 17.5 Cache space overhead for the three organizations

Block size	Direct mapping (%)	4-way set-associative (%)	Fully associative (%)
32 bytes	7	7.8	11
4 bytes	56	62.5	97

If the block size is reduced to 4 bytes, the tag field size increases to 30 bits. The overhead is almost equal to the data field size. For each cache line, we need 32 bits for data and 31 bits (30-bit tag field and a valid bit) of overhead (97% overhead!). Table 17.5 summarizes the results for these examples. These examples demonstrate that fully associative mapping is not only complex in terms of the logic circuit required for block location, but it also requires more space for the tag field.

17.9 Mapping Examples

For the examples in this section, assume 16-bit addresses (i.e., memory address space is 2^{16} bytes) and 1 KB cache size. In addition, we use a block size B of 8 bytes.

Example 17.10 *Describe the cache controller actions to read a 16-bit word at memory address 2E2CH in a direct-mapped cache.*

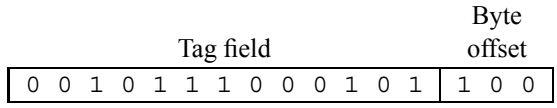
First, we have to find the number of bits used for each field of the address shown in Figure 17.9. Following the examples in the last section, we see that $b = \log_2 B = \log_2 8 = 3$ bits. The number of cache lines C is given by cache size/B. Thus, $C = 1024/8 = 128$ cache lines. The number of bits to identify a cache line c is given by $c = \log_2 C = 7$ bits. The remaining $16 - 7 - 3 = 6$ bits are stored in the tag. The address partition along with the contents of each field is shown below:

Tag field	Cache line #	Byte offset
0 0 1 0 1 1	1 0 0 0 1 0 1	1 0 0

The address 2E2CH is mapped to block 5C5H. Note that we use suffix H to indicate that the number is expressed in the hexadecimal system. For the direct-mapped cache with 128 cache lines, the block is mapped to cache line 45H. If the valid bit is 1, the tag field of this cache line is checked to see if it matches the higher-order 6 bits of the address (0BH). If not, the whole block of 8 bytes is read from the main memory. Note that the starting address of this block is 2E28H. □

Example 17.11 *Describe the cache controller actions to read a 16-bit word at memory address 2E2CH in a cache that uses fully associative mapping.*

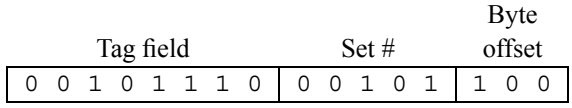
In this mapping, the tag field is $16 - 3 = 13$ bits long. The address partition along with the contents of each field is shown below:



The higher-order 13 bits of the address (5C5H) are compared to the tag field of all cache lines that have 1 as their valid bit. If the block is not present in any of these cache lines, the block is fetched from the main memory and placed in a cache line that is free. If all cache lines have valid blocks, a block is replaced according to the replacement policy in effect. □

Example 17.12 *Describe the cache controller actions to read a 16-bit word at memory address 2E2CH in a cache that uses a four-way set-associative mapping.*

Since this cache uses a four-way set-associative mapping, there are 128/4 = 32 sets. Thus, we need $\log_2 32 = 5$ bits to identify a set. The tag field, therefore, is $16 - 5 - 3 = 8$ bits long. The address partition along with the contents of each field is shown below:



The address 2E2CH is mapped to set 5. The tag fields of the cache lines in this set that have their valid bit on are searched for a match with the address tag field (2EH). If there is no match, the block is not in the cache. The block from the main memory is placed in a free cache line in this set; if all four cache lines have valid blocks, the replacement policy is used to evict a cache line to make room for this new block. □

17.10 Types of Cache Misses

One of the metrics used to measure the performance of a cache system is the number of cache misses. To see if a cache memory design can be improved further, we need to understand the types of misses. Basically, there are three types of cache misses:

- *Compulsory Misses:* These are cache misses generated due to first-time access to the block. Since these blocks have not been previously loaded into the cache, for a given block size, all designs would have to experience these cache misses. In our Example 17.2 reference pattern on page 705, there should be three cache misses as the program references three distinct blocks. These misses are also called *cold-start misses* or *compulsory line fills*.
- *Capacity Misses:* These cache misses are induced due to the capacity limitation of the cache. Since caches do not have the capacity to store all the referenced blocks, we have to replace some of the blocks that may be referenced again later. If we had enough room we could have avoided replacing these blocks.
- *Conflict Misses:* These cache misses, also called *collision misses*, are due to a conflict caused by the direct and set-associative mapping functions. Conflict misses are those that

are caused by direct and set-associative mapping functions that are eliminated by using the fully associative mapping.

The cache design and the parameters used can reduce these three types of misses. The main factor that affects the compulsory misses is the block size. Increasing the block size has the effect of prefetching additional memory addresses, which is useful because of spatial locality characteristic of program referencing behavior. However, as you increase the block size beyond a certain point, the miss rate increases. The increase is in part due to larger blocks being replaced, which might throw some data that a smaller block might have kept in the cache: more on this subject in Section 17.13.

Increasing the cache size can reduce capacity misses. As we increase the cache size, the law of diminishing returns takes over. Larger caches also take more die space. Furthermore, larger caches increase the access time. All these factors may lead to lower overall performance.

Increasing the degree of associativity can reduce conflict misses. In the Example 17.2 reference pattern, all blocks are mapped to the same cache line causing each access to result in a cache miss. We have eliminated some of these misses by using the set-associative mapping (see Example 17.5). Further improvement was obtained by using fully associative mapping (see Example 17.4). Clearly, the degree of association can reduce these cache misses. By definition, a fully associative cache will eliminate these misses completely.

17.11 Types of Caches

Our discussion so far has been on the basic aspects of cache memory design. In this section, we look at some variations on the basic cache organization that lead to performance improvements.

17.11.1 Separate Instruction and Data Caches

Caches can be classified along several dimensions. One of them consists of the cache being a common cache for both instructions and data (i.e., unified cache) or separate caches existing for data and instructions. Early cache designs used unified caches. The recent trend is to use separate caches for data and instructions. Such caches are organized as shown in Figure 17.18.

There are several reasons why a separate cache organization is preferred in current designs. Locality tends to be stronger when instruction references and data references are considered separately. This fact implies that split caches tend to outperform a unified cache of equivalent size. We can also have different designs for these two caches. Since the instruction cache is read-only, we do not have to worry about the write policies. Similarly, program execution tends be sequential most of the time; it may be even possible to use a direct-mapped cache to simplify the design. For the data cache, we can use a set-associative cache with an appropriate write policy.

A disadvantage of the split cache is that you cannot dynamically allocate more cache lines to data and vice versa. On the other hand, a unified cache can change the size of the cache allocated to data and instructions. If a particular application is data-intensive, a unified cache automatically allocates more space to data. Thus, the static bounds on the cache size in a split cache may cause performance problems for some applications.

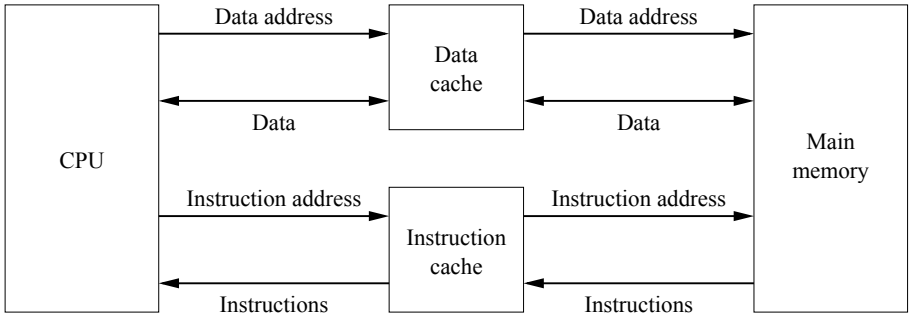


Figure 17.18 Organization of separate data and instruction caches.

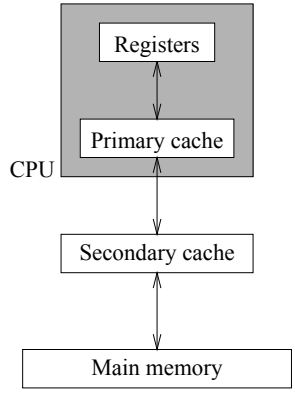


Figure 17.19 Primary and secondary caches in the memory hierarchy.

17.11.2 Number of Cache Levels

Most of the current processors support two-level caches. These systems have a primary cache (also called level 1 or L1 cache) and a secondary (or level 2 or L2) cache as shown in Figure 17.19. The primary cache is on the chip (either on the same die or on a separate die); for this reason, it is also called an onchip cache. The secondary cache is outside the CPU chip but typically on the CPU (mother)board. The secondary cache can be designed to meet the specifications of a given system to augment performance. In general, the size of the secondary cache is larger than the primary cache. For example, on Pentium family processors, the L1 cache is about 32 KB, whereas the L2 cache can be up to 2 MB. Similarly, on the PowerPC, the L1 cache is 64 KB, and the L2 cache can be up to 1 MB.

A typical two-level cache system works as follows:

1. The processor first attempts to access the required data (instruction or data) from the L1 cache.
    - If the data are present in the L1 cache, they are retrieved from the L1 cache ("L1 cache hit").
    - If not, an L1 cache miss occurs, and the data must be retrieved from the L2 cache or the main memory.
2. If an L1 cache miss occurs, the cache controller attempts to access data from the L2 cache.
    - If the data are present in the L2 cache, they are retrieved from the L2 cache ("L2 cache hit"). The word is supplied to the processor, and the data block is written into the L1 cache.
    - If not present in the L2 cache, an L2 cache miss occurs, and the data must be retrieved from the main memory and supplied to the processor. The block from the main memory is also written into both caches.

From the description, it is clear that the same data are present in three places simultaneously: the L1 cache, L2 cache, and main memory.

The secondary cache referencing pattern could be different from the primary cache, particularly when the write-through policy is used at the primary level. In this case, the secondary cache may see more writes than reads. Remember that the primary cache sees far fewer writes than reads.

The purpose of the secondary cache is to catch the misses from the primary cache. If we assume that the primary cache is able to support a hit ratio of 90%, the secondary cache will at most be supporting the remaining 10%. If we further assume that the secondary cache also has a hit ratio of 90%, main memory will end up supporting the $0.1 * 10\% = 1\%$ of the references. From this description, it is clear that the primary cache is almost always busy, whereas the secondary cache is idle most of the time.

Secondary cache placement in the hierarchy differs from system to system. The placement shown in Figure 17.19 follows the standard hierarchy. A principle of this hierarchy is that the data located at a higher level are contained in the lower level. Do we have to cache data in both caches? If we follow the inclusion principle of the memory hierarchy, we have to; but it is not strictly necessary. Two-level cache systems can be built by excluding the contents of the primary cache from the secondary cache. A cache controller can be designed to manage the two caches appropriately. However, due to the complexities associated with this exclusion technique, it is not often used in practice.

Although most processors support two-level caches, three-level caches are also used in some processors. The Compaq Alpha (previously DEC Alpha) processor 21164, for example, provides support for three-level caches. The first two levels of the cache, L1 and L2, are onchip

and the third-level cache L3 is offchip. The L1 cache is a split cache with 8 KB each of instruction and data caches. Both are direct-mapped caches and use 32-byte blocks. The data cache uses the write-through policy. The L2 cache is a unified cache of 96 KB. It is a three-way set-associative cache that uses the write-back policy. The L3 cache is an external cache, which is optional. It uses direct mapping and the block size can be 32 or 64 bytes. Like the L2 cache, it also uses the write-back policy. The size of the L3 cache can be 1, 2, 4, 8, 16, 32, or 64 MB.

17.11.3 Virtual and Physical Caches

Our discussion implicitly assumed that we are dealing with physical memory addresses; however, there is no reason why caches should operate on the physical addresses. With some modifications, the same principles apply to virtual addresses as well. Essentially, the location of the cache determines whether it is a physical or virtual cache as shown in Figure 17.20. All addresses upstream (toward the processor) of the memory management unit (MMU) are virtual addresses. The MMU translates virtual addresses into physical addresses. Physical caches are located downstream of the MMU. Details on the MMU operation are given in the next chapter.

In a two-level cache system, the primary can be virtual or physical depending on its location relative to the MMU. Since the MMU is integrated into the processor, system designers do not have an option for the secondary cache; it has to be a physical cache.

In the Alpha, the L1 instruction cache is a virtual cache; other caches are physical caches. Pentium and PowerPC processors use only physical caches. The MIPS uses virtual caches for L1 data and instruction caches. For L2, it uses physical caches.

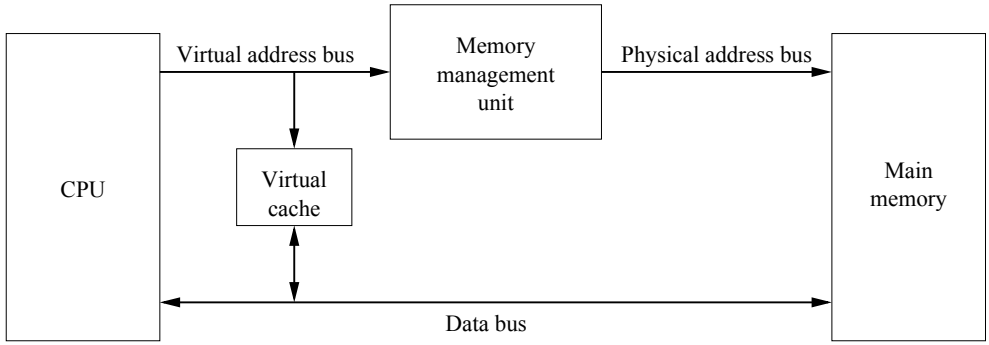
17.12 Example Implementations

In this section, we look at the cache implementations of Pentium, PowerPC, and MIPS processors.

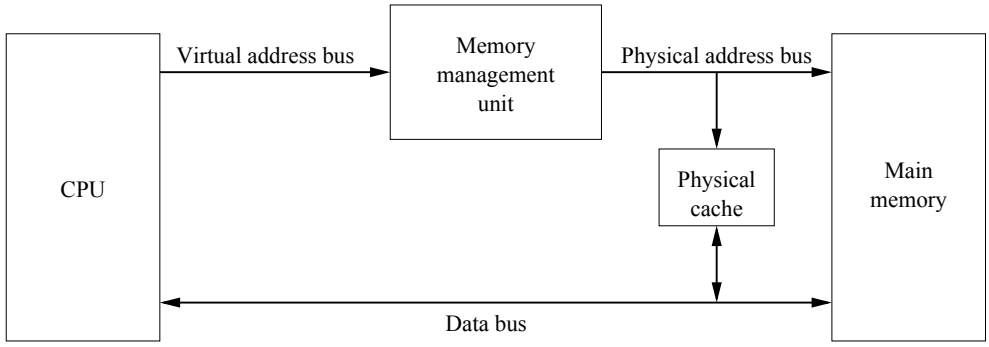
17.12.1 Pentium

The Pentium supports two-level caches. The L1 cache is a split cache, whereas the secondary cache is a unified cache. Pentium family processors (including Pentium-Pro, Pentium II, and Pentium III processors, which are referred to as P6 family processors) have 8 or 16 KB L1 instruction caches (I-caches). This is a four-way set-associative cache with a cache line size of 32 bytes. In contrast, the earlier i486 processor used a unified four-way set-associative L1 cache with a 16-byte cache line. The L1 data cache (D-cache) is similar to the I-cache; it is a 16 KB four-way set-associative cache with a cache line size of 32 bytes.

The L2 cache is unified and is external to the processor in i486 and Pentium processors. In P6 family processors (including Pentium-Pro, Pentium II, and Pentium III processors), the L2 cache is internal to the processor package. The L2 cache can be 128, 256, 512 KB, or 1 or 2 MB in size. As with the L1 cache, it is also a four-way set-associative cache with a cache line size of 32 bytes. Cache lines are filled from memory with a four-transfer burst transaction, where each transfer brings 64 bits. Chapter 5 describes burst transfer of data.



(a) Virtual cache



(b) Physical cache

Figure 17.20 Physical and virtual cache systems.

Read misses always load the cache (L1, L2, or both). The Pentium family supports both write-through and write-back policies. However, only write-through was supported in the i486. On i486 and Pentium processors, write misses will not cause cache line fills. Writes are sent directly to the memory through a write buffer.

The processor allows any memory region to be cached in both levels of caches. It is also possible to set the type of caching for each page or region of memory. These options are set through system flags and registers. The following five modes are supported:

- *Uncacheable:* In this mode, the corresponding memory locations are not cached. All reads and writes go to the main memory. This mode of operation is useful for memory-mapped I/O devices. Furthermore, it can also be used in situations where a large data structure is read once but will not be accessed again until some other entity (processor, I/O device, etc.) updates the data. Similarly, uncacheable mode is useful for write-only data structures.

Table 17.6 Pentium family cache operating modes

CD	NW	Write policy	Read miss	Write miss
0	0	Write-through	Cache line filled	Cache line filled
0	1	Invalid combination—causes exception		
1	0	Write-through	No cache line fills	No cache line fills
1	1	Qrite-back	No cache line fills	No cache line fills

- *Write Combining:* In this mode also, memory locations are not cached. However, writes may be buffered and combined in the write buffer to reduce access to the main memory. This mode is useful for video frame buffers, where the write order is not important as long as all the changes can be seen on the display.
- *Write-Through:* This mode implements the write-through policy. However, writes to main memory may be delayed as they go through the write buffer as in the write combining mode.
- *Write-Back:* This mode implements the write-back policy. As with the write-through mode, write combining is allowed to reduce bus traffic.
- *Write Protected:* This mode inhibits cache writes; such writes are propagated to the memory.

Two flag bits in control register CR0 determine the mode. The cache disable (CD) flag bit controls caching of memory locations. If this bit is set, caching is disabled. The other bit—not write-through (NW)—controls the type of write policy in effect. When this bit is one, write-back policy is used; otherwise, write-through policy is used. Table 17.6 gives the three valid combinations for these two bits. During normal cache operation, both CD and NW are set to zero.

17.12.2 PowerPC

The PowerPC also supports two-level caches. The L1 cache uses separate instruction and data caches. Each is a 32 KB, eight-way set-associative cache. The block size is 32 bytes. The L1 cache uses a pseudo-LRU (PLRU) replacement policy (described later). The instruction cache is read-only, whereas the data cache supports read/write. Write policy can be set to either write-through or write-back.

The L2 cache is a unified cache as in the Pentium with an LRU replacement. The PowerPC 750 can support up to a 1 MB external L2 cache, organized as a two-way set-associative cache. The L2 cache can use synchronous SRAM with sizes 256 or 512 KB, or 1 MB. The block size is 32 bytes as in the L1 cache. The write policy can be write-through or write-back. As in the Pentium, both the L1 and L2 caches are physical caches.

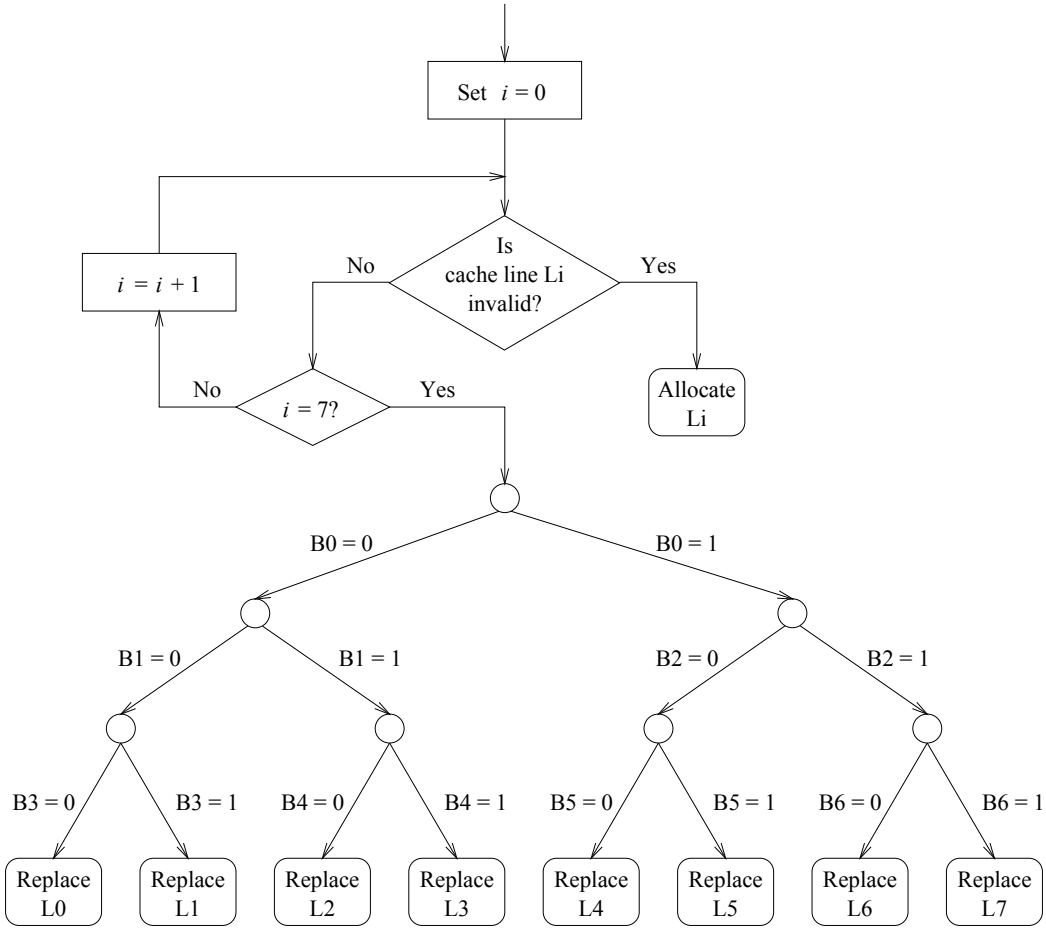


Figure 17.21 PowerPC placement policy. It implements a pseudo-LRU to approximate the true LRU policy.

Write policy type and caching attributes can be set by the operating system at the block or page level using W and I bits. The W bit determines whether write-through or write back policy is used. Whether to cache a block/page is determined by the caching-inhibit (I) bit.

Since the L2 cache is two-way set-associative, implementing strict LRU requires only a single bit. This bit identifies the set that is least recently accessed. However, implementing a strict LRU for the L1 cache is more complex because it is eight-way associative. As mentioned in Section 17.6, most implementations use an approximated version of the true LRU.

PowerPC placement policy is shown in Figure 17.21. The policy first tries to find a block that is free (i.e., valid bit is off) by scanning cache lines from L0 to L7 in that order. If an invalid cache line is found, it is allocated to the incoming block. If all eight cache lines have valid blocks, the pseudo-LRU replacement policy is invoked.

Table 17.7 Pseudo-LRU bit update rules for the PowerPC

Current access	Change PLRU bit to:						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	NC	1	NC	NC	NC
L1	1	1	NC	0	NC	NC	NC
L2	1	0	NC	NC	1	NC	NC
L3	1	0	NC	NC	0	NC	NC
L4	0	NC	1	NC	NC	1	NC
L5	0	NC	1	NC	NC	0	NC
L6	0	NC	0	NC	NC	NC	1
L7	0	NC	0	NC	NC	NC	0

NC: No change.

To implement the PLRU policy, each set maintains seven PLRU bits B0 to B6. The policy follows the basic description given in Section 17.6. The PLRU bits determine the cache line selected for replacement. For example, assume that B0 is 0. This limits the set of cache lines for replacement to L0 to L3. In this case, we look at B1 to restrict the set further to either L0 and L1 or L2 and L3. If B1 is 0, the set is restricted to L0 and L1; otherwise, only L2 and L3 are considered. The final selection is made using either the B3 or B4 bit depending on the value of the B1 bit (see Figure 17.21).

The seven PLRU bits are updated as shown in Table 17.7. Whenever a cache line is accessed, it sets the bit values of three of the seven bits that are on the path from root to leaf (see Figure 17.21). For example, if L0 is accessed, it sets bits B0, B1, and B3 to 1 to indicate that it should not be the candidate for replacement. Notice that all the other bits remain unchanged (NC). As another example, if L4 is accessed, the B0 bit is cleared, and the B2 and B5 bits are set.

17.12.3 MIPS

Here we describe the MIPS R4000MC processor cache structure. This processor supports two-level caches. The L1 cache is a split cache. The instruction cache (I-cache) is a direct-mapped, virtual cache. It is treated as a read-only cache. As a result, it just maintains a single bit to indicate whether the cache line is valid. The block size can be either 16 or 32 bytes. The L1 data cache is also a direct-mapped virtual cache with 16- or 32-byte block size. It uses the write-back policy. The sizes of the instruction and data caches can each be in the range of 8 to 32 KB. The processor uses the write-back (W) bit to indicate whether the cache line was modified since it was loaded. The W bit serves the same purpose as the dirty bit we discussed in Section 17.7.

The L2 is a physical cache that can be configured at boot time either as a unified cache or a split cache. The L2 cache is also organized as a direct-mapped cache and uses the write-back policy. The cache block size can be 16, 32, 64, or 128 bytes. This size can be set at boot time through boot-mode bits.

Note that when the L2 cache is configured as a unified cache, its size can range from 128 KB to 4 MB. On the other hand, if it is configured as a split cache, each cache can be between 128 KB and 2 MB.

The L1 cache line size cannot be greater than the L2 cache size. This condition implies that the L2 cache cannot use the cache line size of 16 bytes when the L1 cache is using 32-byte blocks.

Since this processor uses direct mapping, the replacement policy is rather straightforward to implement. There is no need to devise ways to implement the LRU policy, as is the case with the other processors.

17.13 Cache Operation: A Summary

In this section, we summarize the various policies used by a cache system.

17.13.1 Placement of a Block

In direct-mapped caches, each block can be placed only in a single, specified cache line. Implementation of this scheme is simple but may lead to performance problems for some applications. The MIPS R4000 processor uses direct mapping for both L1 and L2 caches.

In fully associative mapping, a block can be placed anywhere in the cache. This scheme provides the ultimate flexibility in terms of block placement. However, its implementation is complex as discussed before (see Section 17.5.2). For this reason, this scheme is not used in practice. Also, fully associative mapping requires more tag space (see Section 17.8).

Set-associative mapping is a compromise between these two extremes. The cache lines are divided into sets; each set typically contains a few (two to eight) cache lines. Direct mapping is used to map the block address to a set, and the fully associative method is used to allocate a cache line within the assigned set. This reduces the complexity substantially when compared to the fully associated scheme (discussed further in the "Location of a Block" subsection next). In fact, direct mapping and fully associative mapping can be considered as a special case of the set-associative mapping scheme. Direct mapping is a set-associative scheme with a set size of one cache line. Fully associative mapping is a set-associative scheme with a single set consisting of all the cache lines.

Some systems allocate a cache line only on read misses. Write misses directly update the memory. This scheme is sometimes referred to as the read-allocate policy. Similarly, we can also define a write-allocate policy that allocates a cache line only in response to a write miss. For example, the Alpha uses the read-allocate policy for the L1 cache and the write-allocate policy for L2 caches.

17.13.2 Location of a Block

The cache controller will have to determine whether the requested block is in a cache line. How a block is found depends on the placement policy. In direct-mapped caches, only one cache line tag has to be compared to find if the block is currently cached. In caches using fully associative mapping, all tags will have to be searched for a match. On a Pentium-like machine consisting of a 16 KB cache with a block size of 32 bytes, we have to compare 512 tags. If we assume that the physical address is 32 bits wide, we need 512 27-bit comparators just for the I-cache. If, for example, the cache size is doubled, we also double the number of comparators. This is the main reason that fully associative mapping is not used in practice.

Set-associative mapping reduces this complexity, yet allows more flexibility in placement of blocks compared to the direct mapping scheme. For example, the Pentium uses a four-way set-associative mapping. This requires only four comparisons after determining the potential set that may contain the block. In this mapping, we need only four 25-bit comparators. Furthermore, increasing the cache size does not increase the number of required comparators.

17.13.3 Replacement Policy

The replacement policy is invoked when the target cache lines as determined by the placement policy are all full. In the direct mapping scheme, no special replacement policy is needed. Since each block is mapped to exactly one cache line, if that cache line has a valid block, it will have to be evicted to make room for the incoming block.

For the set-associative and fully associative policies, there is a choice. Since programs exhibit temporal locality, it is better to replace the least recently used block from the cache. However, implementation of LRU is complex; each set requires $\lceil \log_2 N! \rceil$ bits, where N is the number of cache lines in the set. For example, the PowerPC uses eight-way set-associative mapping. Implementation of a true LRU requires $\lceil \log_2 8! \rceil = \lceil \log_2 40320 \rceil = 16$ bits per set. The number of bits can be reduced to $N - 1 = 7$ by using the pseudo-LRU scheme of the PowerPC. For this reason, most implementations use some sort of approximation to the true LRU scheme. Note that a true LRU implementation requires only a single bit if the system uses two-way associative mapping.

17.13.4 Write Policy

Cache systems typically use one of the two write policies: write-through or write-back. Write-through policy always updates the main memory, and if the block is cached, it updates the cache copy as well. Write-back updates only the cache copy. The main memory copy is typically updated when the cache line is evicted.

Write-back tends to reduce the bus traffic due to writes; write-through caches reduce the traffic caused by reads but have no effect on the write traffic. Systems that use write-through may employ write buffers to facilitate buffered writes. For example, the Pentium uses a 32-byte write buffer. This buffer can combine several writes and update the main memory block to reflect these changes. Even write-through policies with sufficient write buffers can approximate the write bus traffic of a write-back cache. But, write buffering introduces complications. For a detailed discussion, see [17].

17.14 Design Issues

There are several design parameters that have significant impact on the performance of a cache system. These include:

- Cache capacity,
- Cache line size or block size,
- Degree of associativity,
- Unified or split cache,
- Single- or two-level cache,
- Write-through or write-back, and
- Logical or physical caches.

We have already discussed the impact of the last four factors on performance. Here we focus on the first three design parameters.

17.14.1 Cache Capacity

From the miss rate point of view, we would like to have a cache as big as possible. On the other hand, a larger cache increases the cost. Therefore, we have to strike a balance between these two requirements. It is helpful to observe that beyond a certain cache size, we see only a marginal improvement in the miss rate (see Figure 17.22a). Furthermore, larger caches mean more gates to address the cache. This slows down the cache as the cache size increases. Also, space availability may impose a limit on the cache size. For example, for L1 onchip caches, the die real estate (area) imposes a limit on the L1 cache size. Typical cache sizes for the L1 cache are in the range of 16 KB to 64 KB. L2 caches typically range between 256 KB to 2 MB.

17.14.2 Cache Line Size

The impact of cache line size or block size is shown in Figure 17.22b. Initially, miss rate decreases as we increase the block size. This is mainly due to the prefetching achieved with larger block sizes. Since programs exhibit spatial locality, prefetching reduces the miss rate. However, as we continue to increase the block size, the miss rate starts to increase. This is due to the block replacements as we run out of space in cache memory. When we replace a larger block to make room for an incoming block, we also throw away data/instructions that the processor might reference in the future. This sensitivity can best be explained by considering the two extreme block size values.

At one end of the spectrum, consider a block size of one word. In this case we are not prefetching data/instructions from the main memory. Thus, if we go from one-word to two-word cache lines, we reduce the miss rate. We can extend this argument to larger block sizes. Larger block size also has the benefit of taking less time to load from main memory in a burst rather than by reading individual words. Most processors support burst transfers. For example, the Pentium requires three clocks to transfer 64-bit noncacheable data from the memory. However, in the burst mode, after an initial delay of two clocks, one 64-bit word is transferred each clock

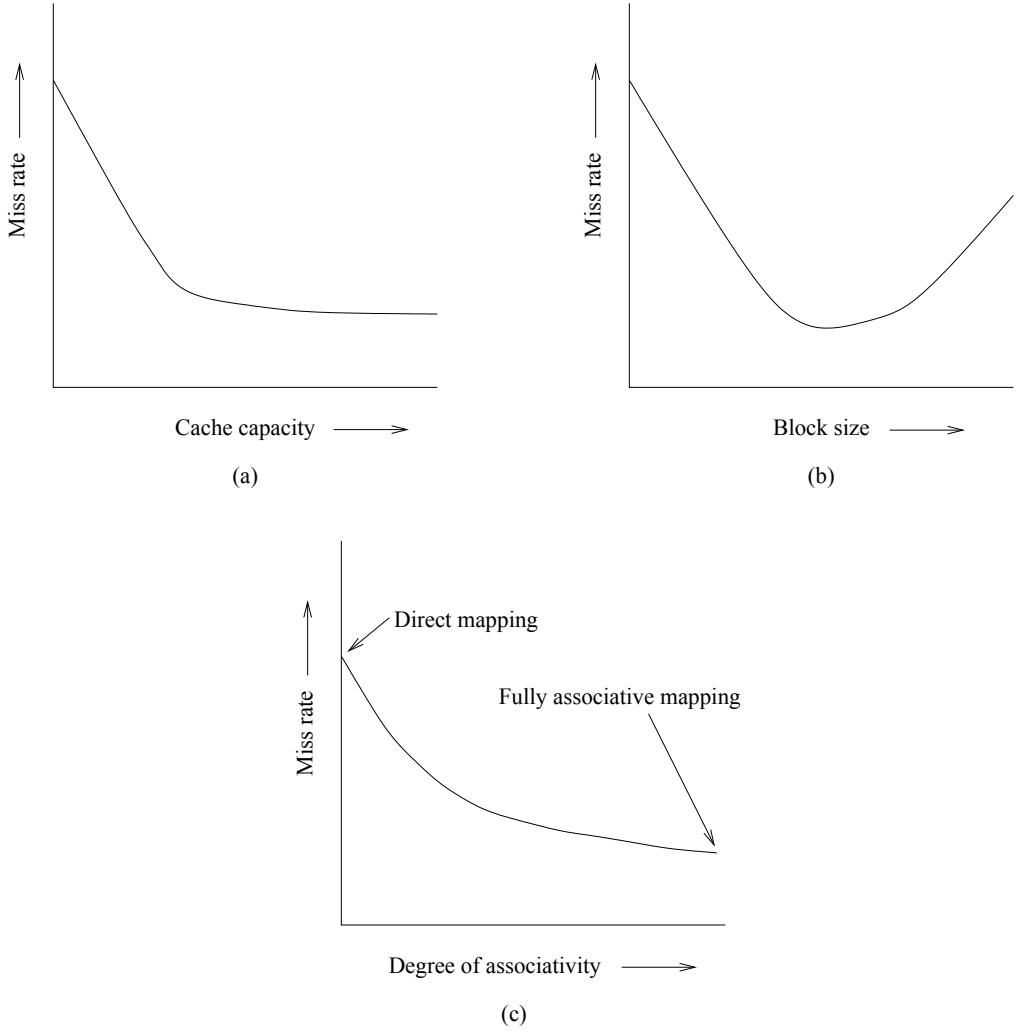


Figure 17.22 Miss rate sensitivity to various cache system parameters.

cycle. In this mode, it can transfer up to four 64-bit words in six clocks. In contrast, we need 12 cycles to transfer the same data using single cycles. This is true whether it is a memory read or a write. Notice that the Pentium uses the 32-byte cache line size, which is exactly the amount of data transferred in one burst cycle (four transfers of 64-bit words each).

At the other extreme, consider a block size equal to the whole cache. In this case, whenever there is a miss, we throw the entire cache contents. In the process, we lose a lot of data that the processor might reference in the future. In addition, such large block sizes affect performance

as the miss penalty increases with the block size. Most processors tend to use a block size in the range of 8 to 64 bytes, with 32 bytes being the most common. Some processors such as the PowerPC and MIPS R4000 allow the system designer to program the cache line size at boot time.

17.14.3 Degree of Associativity

Improving the degree of associativity can significantly improve the miss rate, particularly for smaller caches (see Figure 17.22*c*). We have illustrated this by means of examples in Section 17.5. We know that an increasing degree of associativity calls for more complex implementation. In particular, it is almost impossible to implement full associativity for reasonably large caches. As shown in Figure 17.22*c*, a fairly small degree of associativity can lead to 90% of the gains we can get with full associativity. In practice, two-, four-, or eight-way set-associative mapping is used. Some processors such as the MIPS R4000 and Alpha use direct mapping to reduce the cache controller complexity and the amount of cache memory bits required. In Section 17.8, we have illustrated the space requirements of the three mapping functions by means of several examples.

17.15 Summary

Cache memory plays an important role in improving the performance of computer systems. Cache memory is a small amount of fast memory, usually implemented using SRAMs, which sits between the processor and main memory in the memory hierarchy. The cache effectively isolates the processor from the slowness of the main memory, which is DRAM-based. The idea behind the cache is to keep in the cache data and instructions that are needed in the immediate future by the processor.

Cache memory improves performance because programs exhibit locality in their referencing behavior. We have identified two key locality types: spatial and temporal. Spatial locality behavior suggests that programs tend to access memory in sequence. Sequential code execution supports this claim. Temporal locality suggests that a small part of the code is repeatedly executed for some time. A simple example is the body of a loop structure. These two locality types also apply to data access patterns.

There are four components to a cache implementation policy: block placement, block location, block replacement, and write policies. Block placement can use direct mapping, associative mapping, or set-associative mapping. These mapping functions represent a tradeoff between implementation complexity and flexible placement of blocks. The complexity of block location is related to the mapping function used. Direct mapping is efficient to implement: both in terms of implementation logic and cache memory overhead. Full associativity, on the other hand, requires complex logic as well as more cache memory space to keep the tag information. Set-associative mapping strikes a compromise between these two extremes. In fact, direct and associative mapping functions can be considered as special cases of the set-associative mapping. Most processors implement a set-associative mapping, but direct mapping is also used in some processors.

The block replacement policy determines which block should be evicted in case the cache is full. There are two main policies: random replacement or LRU replacement. Random replacement is easy to implement. True LRU implementation is complex and requires more bits. For these reasons, processors that implement the LRU policy usually implement a pseudo-LRU that approximates the behavior of the true LRU policy.

Write policy can be either write-through or write-back. In the write-through policy, both the cache and main memory copies are updated in response to a write operation by the processor. The write-back policy, on the other hand, updates only the cache copy. The advantage of the write-through policy is that the main memory is (almost) always consistent with the cache copy. But it generates too many writes to the main memory. Practical implementations often use write buffers to combine all the updates before writing to the main memory. Write-back updates the main memory copy only when the cache line is replaced or an explicit cache instruction has been issued. The majority of the implementations use the write-back policy. On some processors, write policy can be selected at boot time.

We have also discussed several variations on the basic cache design. These include split caches and multilevel caches. Split cache designs maintain separate data and instruction caches. Multilevel caches are usually limited two-level caches. Typically, there is a level 1 (L1) or primary cache within the processor chip. A larger level 2 (L2) or secondary cache is used from the chip. Some processors such as the Alpha place the first two levels of caches on the processor chip and allow a third level cache from the chip. Most processors use split cache design for the L1 cache. L2 cache implementations are either unified caches or an option is given to the system designer.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Associative mapping
- Cache capacity
- Cache levels
- Cache miss types
- Cache types
- Capacity misses
- Compulsory misses
- Conflict misses
- Data cache
- Degree of associativity
- Design issues
- Direct mapping
- Dirty bit
- Fully associative mapping
- Hit
- Hit rate
- Hit ratio
- Hit time
- Instruction cache
- Least frequently used (LFU) policy
- Least recently used (LRU) policy
- Line size
- Locality
- Location policies
- Memory hierarchy
- Memory management unit (MMU)

- Miss
- Miss penalty
- Miss rate
- Miss ratio
- Physical cache
- Placement policies
- Pseudo-LRU replacement policy
- Replacement policies
- Set-associative mapping
- Space overhead
- Spatial locality
- Tag field
- Temporal locality
- Update bit
- Valid bit
- Virtual cache
- Write combining
- Write policies
- Write-back
- Write-back bit
- Write-through

17.16 Exercises

17–1 Explain the two components of the locality. Give a simple example code to support these two components.

17–2 How can a cache memory system benefit from the presence of spatial locality?

17–3 How can a cache memory system benefit from the presence of temporal locality?

17–4 What is the purpose of the valid bit?

17–5 What is the purpose of the dirty bit?

17–6 Do you need to maintain the dirty bit information if you are using the write-through policy?

17–7 In a direct mapped cache, do you have a choice to select the replacement policy? Explain your answer.

17–8 What are the advantages and disadvantages of a direct mapped cache over a fully associative cache?

17–9 Why is a fully associative cache not implemented in practice?

17–10 What are the advantages and disadvantages of the set-associative cache over direct mapped and fully associative caches?

17–11 We have discussed two components of locality that benefit cache memory systems. When we load a block of data from the main memory, which of these components shows that this is a beneficial action?

17–12 Discuss the tradeoffs associated with the block size. That is, discuss the pros and cons of small and large block sizes.

17–13 Example 17.5 (on page 710) shows an example reference pattern that results in a 67% hit ratio. Can you devise a 12-block long reference pattern that gives the best hit ratio using the same set-associative cache? You must use four distinct blocks in your reference pattern.

17–14 Suppose that the reference pattern of a program is such that it accesses the following sequence of blocks: 0, 4, 3, 10, 10, 3, 4, 0, 0, 4, 3, 10, 4, 10. Find the hit ratio with a direct mapped cache of four cache lines.

17–15 Using the reference pattern given in Exercise 17–14, find the hit ratio of a set-associative cache of four lines with a set size of 2. Use the true LRU replacement policy.

17–16 Consider the reference pattern given in Exercise 17–14. Without going through a detailed analysis, can you tell if we get the best hit ratio with a fully associative cache of four cache lines? If so, what is the hit ratio?

17–17 Suppose that the reference pattern of a program is such that it accesses the following sequence of blocks: 0, 1, 2, 3, 4, 5, 5, 4, 3, 2, 1, 0, 1, 10. Find the hit ratio with a fully associative cache of four cache lines. Do the exercise with FIFO and LRU replacement policies. Is there a difference in the hit ratio?

17–18 Repeat Exercise 17–17 with the following reference pattern: 0, 1, 2, 0, 3, 2, 5, 3, 6, 0, 2, 1.

17–19 Consider a system with 4 GB of memory address space. It uses a 64 KB direct mapped cache. Assuming a write-back policy, find the space overhead when

(a) Block size is 16 bytes;
(b) Block size is 32 bytes;
(c) Block size is 64 bytes.

17–20 Repeat Exercise 17–19 for a fully associative cache.

17–21 Repeat Exercise 17–19 for an eight-way set-associative cache.

17–22 Consider a system with 4 GB of memory address space. It uses a 32 KB cache. Assuming a direct mapped cache that uses 32-byte blocks, find the size (in bits) of the following: tag field, cache line number, and byte offset.

17–23 Consider a system with 4 GB of memory address space. It uses a 32 KB cache. Assuming a fully associative cache that uses 32-byte blocks, find the size of the tag field in bits.

17–24 Consider a system with 4 GB of memory address space. It uses a 32 KB cache. Assuming an eight-way set-associative cache that uses 64-byte blocks, find the size (in bits) of the following: tag field, cache line number, and byte offset.

Chapter 18

Virtual Memory

Objectives

- To explain the basic concepts of virtual memory;
- To discuss design and implementation issues;
- To present details about TLBs to perform fast address translation;
- To describe segmentation;
- To provide details about virtual memory implementations of the Pentium, PowerPC, and MIPS processors.

Virtual memory was developed to give programs a larger memory space than the system's main memory. The appearance of larger address space is realized by using much slower disk storage. The concepts and the principal implementation techniques, discussed in Section 18.1, are very similar to the cache systems discussed in the last chapter. In implementing virtual memory, the main memory and disk are divided into fixed size pages. A mapping table called a page table does the mapping of pages between the disk and main memory. Section 18.2 presents virtual memory concepts. The page table structure is described in Section 18.3.

When we use such a page table, every memory access involves two accesses: one to access the page table and the other to access the program's requested page. To reduce this unacceptable overhead, the processor maintains a cache of most recently used page table entries. This cache is called the translation lookaside buffer (TLB). Section 18.4 gives details on the TLB organization.

Page tables used to translate virtual addresses to physical addresses tend to be very large. Consequently, these tables are stored in the virtual address space. Section 18.5 describes the page table structure in detail. The size of these page tables is proportional to the virtual address space. For 64-bit processors, the table size can be very large. To reduce the table size, an inverted page table has been proposed (Section 18.6). In the inverted page table, the number of entries is proportional to the physical memory, not the virtual memory.

Processors such as the Pentium and PowerPC use segmentation to facilitate implementation of protection as well as to extend the virtual memory. Section 18.7 discusses the motivation and concepts involved in segmentation. Virtual memory implementations of Pentium, PowerPC, and MIPS processors are discussed in Section 18.8. The chapter concludes with a summary.

18.1 Introduction

When you write programs, you are not really concerned with the amount of memory available on your system to run the program. What if your program requires more memory to run than is available on your machine? This is not a theoretical question, in spite of the amount of memory available on current machines. Even on a single-user system, not all the memory is available for your program. The operating system takes quite a big chunk of it. If you consider a time-shared multiprogrammed system, the problem becomes even more serious. Virtual memory was proposed to deal with this problem.

Before the virtual memory technique was proposed, one had to resort to a technique known as overlaying in order to run programs that did not fit into the physical memory. With only a tiny amount of memory available (by current standards) on the earlier machines, only a simple program could fit into the memory. In this technique, the programmer divides the program into several chunks, each of which can fit in the memory. These chunks are known as *overlays*. The whole program (i.e., all overlays) resides in the disk. The programmer is responsible for explicitly managing the overlays. Typically, when an overlay in the memory is finished, it will bring in the next overlay that is required for program execution. Needless to say, this is not something a programmer would like to do. Virtual memory automates the management of overlays without requiring any help from the programmer. As we show in this chapter, virtual memory implementations typically provide much more functionality than just managing overlays. These functions include the following:

- *Relocation:* Each program can use its own virtual address space; when they run, they may be mapped to different physical memory locations. These run-time details do not have any impact on code generation because virtual memory addresses generated by processors are mapped to physical addresses on the fly at run time.
- *Protection:* Since each program is working in its own virtual memory address space, virtual memory facilitates isolation of programs from each other and implementation of protection.

Many of the concepts we use here are similar to the concepts used in cache systems. If you are not familiar with the cache concepts, it is a good time to review the cache principles presented in the last chapter. Caches use a small amount of fast memory but to a program it appears as a large amount of fast memory. As mentioned before, virtual memory also provides a similar illusion. As a result of this similarity, the principles involved are the same. The details, however, are quite different because the motivations for cache memory and virtual memory are different. We use cache memory to improve performance whereas virtual memory is necessary to run programs in a small amount of memory.

As in the cache memory, the concept of locality is important to make the virtual memory work efficiently. Both types of locality—temporal as well as spatial —are important. These two locality types were discussed in the last chapter. You should, however, note that even if a program exhibits no locality in its referencing behavior, we would still use virtual memory just to run the program. This is mainly due to the difference in the objectives of the two systems: in one it is optional, and in the other it is required for the execution of programs. The implementation of virtual memory also differs from cache implementation due to the fact that the lower-level memory (disk) is several orders of magnitude slower than the main memory. As a result, we do not have to implement the mapping function in hardware. We can implement it in software.

18.2 Virtual Memory Concepts

In simple terms, virtual memory implements a mapping function between a much larger virtual address space and the physical memory address space. For example, in the PowerPC a 48-bit virtual address is translated into a 32-bit memory address. On the other hand, the Pentium uses 32-bit addresses for both virtual and physical memory addresses. However, it uses segmentation to effectively increase the physical memory.

To implement virtual memory, the virtual address space is divided into fixed-size chunks called *virtual pages*. The virtual address is divided into a *virtual page number* and a *byte offset* within a page. Similarly, the physical memory is also divided into similar-size chunks called *physical pages* or *page frames*. Each physical address also consists of a physical page number and an offset. In translating a virtual address to a physical address, the offset part remains the same; only the virtual page numbers are mapped to physical page numbers. This concept of a page is similar to that of a cache line, except that the page size is much larger than the cache line size. A typical page size in current processors such as the Pentium is 4 KB.

Figure 18.1 shows the mapping of a 32-bit virtual address space to a 24-bit physical address space. Assuming a page size of 4 KB, the least significant 12 bits of the virtual and physical addresses are used to identify a byte in the page (page offset). The remaining upper address bits are used to identify a page number. In this example, the upper 20 bits in the virtual address identify a virtual page, and the upper 12 bits in the physical address refer to a physical page. Thus, the mapping is from 2^{20} virtual pages to 2^{12} physical page frames as shown in Figure 18.2.

From the operating system point of view, a virtual page is in one of two places: either it is in the main memory or on disk. Note that a virtual page that is in main memory is also on disk. If a page is not in the main memory, a *page fault* occurs. Page fault corresponds to a cache miss. Then the operating system takes control and transfers the missing page into a main memory page and updates the page table to reflect this action as shown in Figure 18.3. This is called *demand paging* as pages are transferred to the main memory on demand. If the main memory is full, a page replacement policy such as LRU is used to make room for the new page. The memory management unit is responsible for translation of virtual addresses to physical addresses.

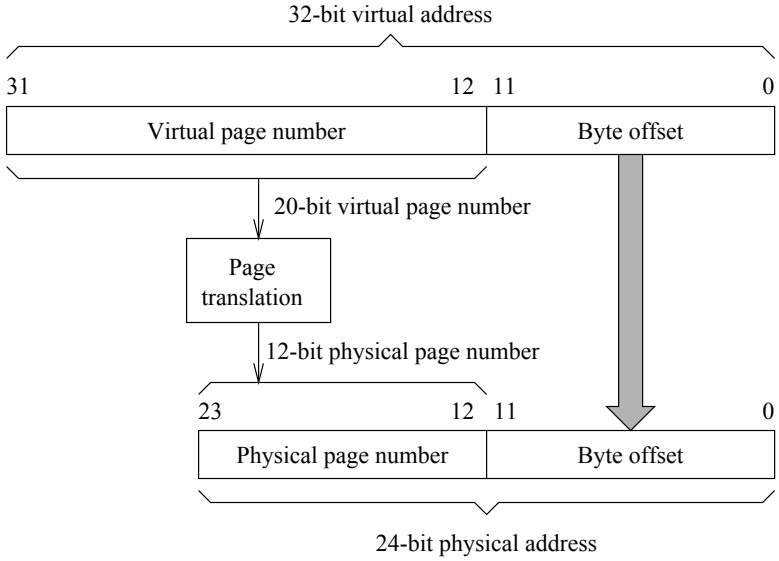


Figure 18.1 Translation of virtual address to physical address: A page size of 4 KB is used.

18.2.1 Page Replacement Policies

In the last chapter, we discussed several replacement policies for cache systems (see Section 17.6 on page 711). Cache systems do not use elaborate replacement policies for two main reasons: the miss penalty is not as high as in the virtual memory systems, and the implementation is typically done in hardware.

Virtual memory systems implement the replacement policy in software. Furthermore, since the miss penalty is several orders of magnitude larger, a good page replacement is important. Consequently, several policies have been proposed. Here we briefly mention some of the interesting ones.

The first policy we look at is the first-in-first-out policy, which is simple to implement. As the name suggests, the page that has been loaded the earliest into the memory is the one that should be replaced. In other words, FIFO policy replaces the page that has been in memory the longest. A drawback is that FIFO does not consider the usage of a page (i.e., whether the page has been recently referenced). As a result, it might remove a page that is in fact needed. For this reason, the FIFO policy is not used in practice.

Another policy, called the *second chance* policy, has been proposed to avoid the problem of evicting pages that are useful. Before replacing a page, this new policy looks at whether the page has been referenced. Yet another policy is the not frequently used (NFU) policy, where a software counter is associated with each page to keep track of its usage. When a page needs to be replaced, NFU selects the page with the smallest count value. More details on these policies can be found in [36].

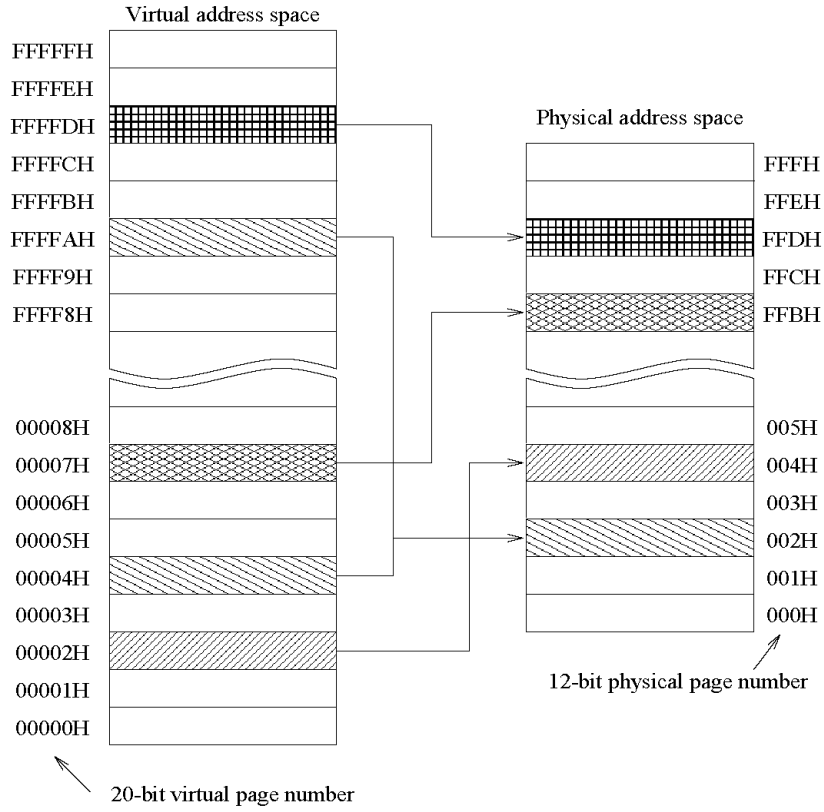


Figure 18.2 Mapping of a 2^{32}-byte virtual address space to a 2^{24}-byte physical address space: A page size of 4 KB or 2^{12} bytes is used. The suffix H indicates that the numbers are in hexadecimal notation.

Most virtual memory systems implement the least recently used policy in some form. We have discussed this policy in the last chapter. Even though the operating system does the page replacement in software, we still cannot implement a true-LRU policy due to the overhead caused by large page tables. Instead, as discussed in Section 17.6, LRU policy is approximated. To assist the operating system in implementing a pseudo-LRU policy, each page table entry has a reference bit to indicate whether the page has been recently accessed. We discuss page table entry format on page 742.

18.2.2 Write Policy

We discussed two basic ways of handling writes in cache systems: *write-through* and *write-back* (see Section 17.7 on page 713). A write-through policy is not suitable for virtual memory due to high disk access times. One difference between a cache write and a write in a virtual memory system is that the virtual memory system implements protection at the page level. Thus, if the

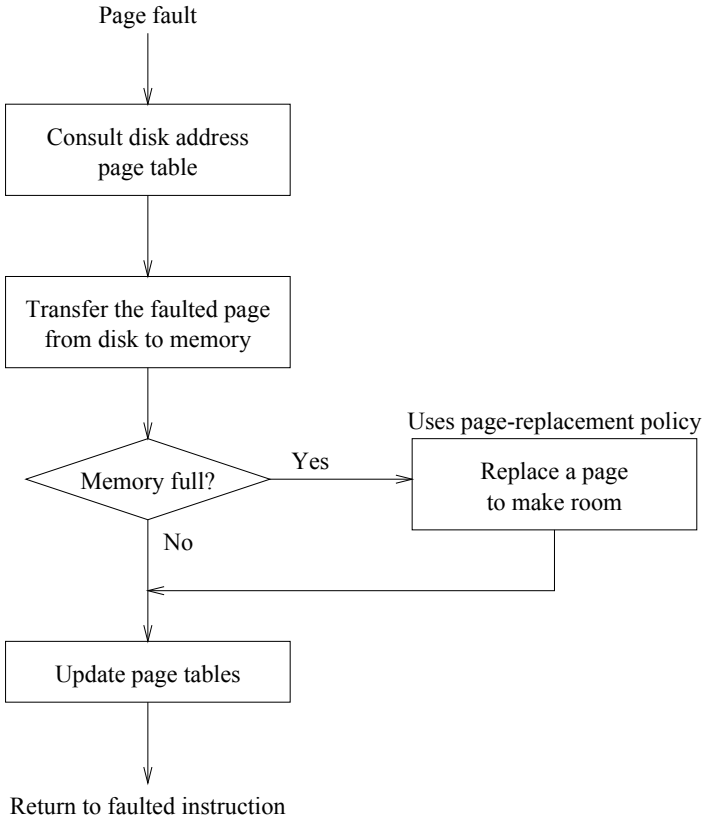


Figure 18.3 Page fault handling routine: Systems typically implement a psuedo-LRU page replacement policy.

program does not have write permission on the referenced page, a write protection exception is generated.

18.2.3 Page Size Tradeoff

The operating system can select a page size that results in the best performance. However, several factors influence the best page size.

Factors favoring small page sizes are as follows:

- *Internal Fragmentation:* Since data, code, and stack are not going to be an integral number of pages, the last page is going to be half full on average. This is referred to as internal fragmentation. Clearly, using smaller pages reduces internal fragmentation.

- *Better Match:* Large page size tends to load more, potentially unused, portions of the program into main memory. A better match with the working set of the program can be obtained by using smaller pages.

Factors favoring large page sizes are as follows:

- *Smaller Page Tables:* Larger pages reduce the number of page table entries, which results in smaller page tables. For example, in Figure 18.2, if we use 32 KB pages instead of 4 KB pages, we reduce the number of virtual pages from 2^{20} to 2^{17} pages. As we show in the next section, the number of page table entries of a page table is equal to the number of virtual pages.
- *Disk Access Time:* Accessing a page on disk is slow mainly due to seek time and rotational delays. The transfer time is smaller than these two delays. Thus, transferring larger pages from disk to main memory takes almost the same time as a smaller page.

Notice that some of these points are similar to the tradeoffs discussed for cache line sizes (see page 729). Both the Pentium and PowerPC use 4 KB pages. With increasing virtual address space and physical memory, page sizes of up to 64 KB are supported in new systems. In some processors, page size is not fixed. For example, the MIPS R4000 supports seven page sizes between 4 KB and 16 MB. Note, however, optimum page size is often selected by the operating system. The operating system may use the natural hardware-defined page size, or it can use a larger page size. For example, the operating system may treat two hardware-defined pages as a single page.

18.2.4 Page Mapping

Since the miss penalty is very high in virtual memory, we want to minimize the miss rate. This implies that we have to use a fully associative mapping scheme. Recall that fully associative mapping can place a virtual page in any physical page (see Section 17.5.2 on page 707).

In a cache system design, we could not afford to implement a sophisticated mapping function because the miss penalty is not high enough to justify such an implementation. Furthermore, the implementation has to be done in hardware. Since the disk represents the lower-level store in virtual memory and is slow, we can implement mapping in software. In addition, due to the high miss penalty, we do not have to use a quick and dirty mapping function. In virtual memory, mapping is actually done by using a translation table called the *page table*. The page table organization is described next.

18.3 Page Table Organization

In a simple page table implementation, each entry consists of a virtual page number and the corresponding physical page number. This type of organization, however, leads to unacceptable overhead, as it involves sorting the table entries on virtual page numbers. In addition, there is considerable overhead to search for an entry in such a table. To speed up the page table lookup, we can use the virtual page number as an index into the page table.

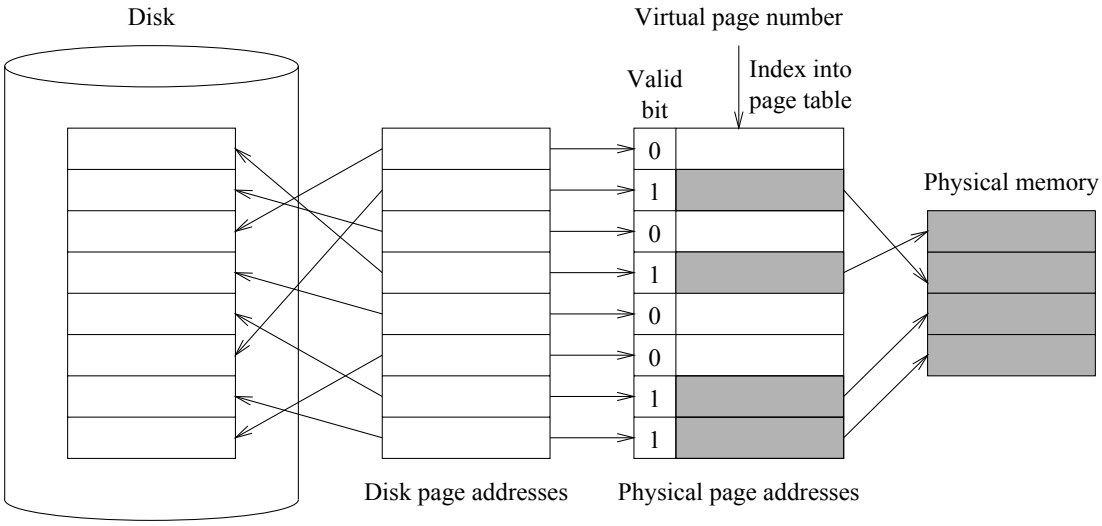


Figure 18.4 Page table organization for a hypothetical system with four pages of main memory supporting a virtual address space of eight pages.

Figure 18.4 shows the organization of a page table. A typical page table is implemented using two data structures: one stores the physical page addresses, and the other maintains the disk addresses of all pages. We can use the virtual page number (VPN) as an index into both tables. The physical page address table contains valid entries only for those pages that are in the main memory. The valid bit is used to indicate this fact. If we are looking for a VPN for which the valid bit is 0, a page fault is generated. As discussed before, the operating system loads the page before handing control to the program, which reexecutes the instruction that caused the page fault (see Figure 18.3).

18.3.1 Page Table Entries

Each entry in the page table, called a page table entry (PTE), provides the mapping information from virtual to physical page. Since we use VPN as an index, PTEs need to store only physical page numbers. In addition, there is a valid bit to indicate whether the virtual page is in memory or on disk. As mentioned in Section 18.1, virtual memory also supports address-space and page-level protection. So, additional control bits are usually added to provide this capability.

Each PTE stores the following information:

- *Physical Page Number:* Gives the location of the page in main memory if the page is in memory.
- *Disk Page Address:* Specifies the location of the page on disk. This information is kept on all pages, whether they are in memory or not as shown in Figure 18.4.
- *Valid Bit:* Indicates whether the page is in memory.

- *Dirty Bit:* Indicates whether the page has been modified. If the page has been written into, we have to write this back to disk before discarding it.
- *Referenced Bit:* Used to implement the pseudo-LRU algorithm. The OS periodically clears this bit to check the usage of pages. Accessing the page turns this bit on.
- *Owner Information:* To implement proper access control, the OS needs to know the owner of the page.
- *Protection Bits:* Indicates the type of privilege the owner of the page has (read-only, execute, read/write, etc.). For example, the PowerPC uses three protection bits to give various types of access to supervisor-mode and user-mode access requests.

18.4 The Translation Lookaside Buffer

Earlier systems with smaller virtual address spaces could maintain the translation table in hardware. However, with larger address spaces supported by the current systems (e.g., the Intel Itanium supports 64-bit virtual addresses), the translation table must be stored in the virtual address space. This means that, for every virtual address generated by the processor, two memory accesses are required: one to get the physical page number corresponding to the virtual page number and the other to access the program's memory location.

To reduce this inherent overhead, processors maintain most recently used PTEs in a cache. For historical reasons, this cache is referred to as the translation lookaside buffer. The TLB is small in size as it is part of the processor. It typically contains 32 to 256 entries. Each TLB entry consists of

- A virtual page number,
- Corresponding physical page number, and
- Various control bits including the valid and reference bits.

Note that all pages with a PTE in the TLB are in the main memory. Most systems maintain a separate (split) TLB for data and instructions. For example, the Pentium and PowerPC use split TLBs.

With a TLB in place, the translation process works as shown in Figure 18.5. The TLB is used to see if there is an entry for the VPN. If so, the associated physical page number is used to generate the physical page address. If there is no entry for the VPN, we have to search the page table in main memory. If the requested page is in the main memory, the page table will have an entry (with a valid bit). In this case, the TLB is updated with an entry for the new page. The TLB update can be done either in hardware or software. The TLB update step requires a replacement policy: which entry is to be replaced when the TLB is full. If the update is done in software, we can implement various replacement policies. However, if this update is done by hardware, simplicity is important. Typically, a random or pseudo-LRU policy is used. For example, MIPS processors use a random replacement policy.

To reduce TLB misses, most systems use a fully associative map for TLBs. This is particularly true for small TLBs. If the TLB is large, a set-associative map is used. Pentium and PowerPC processors use a fully associative map.

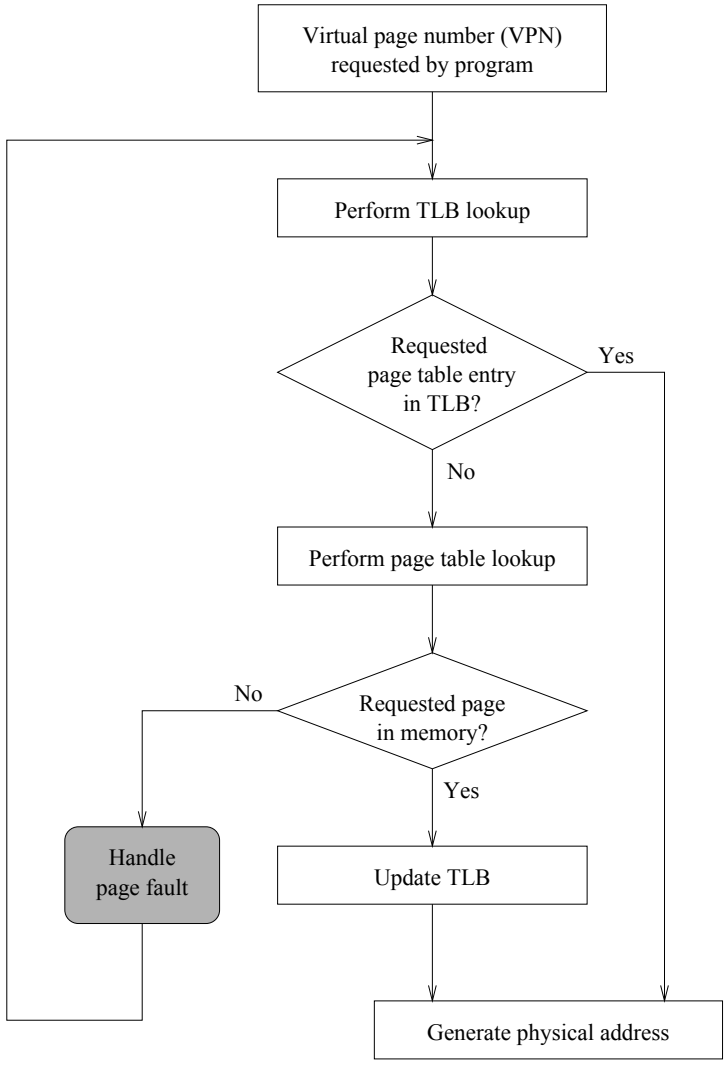


Figure 18.5 Translation of virtual memory address to physical page address using a TLB.

18.5 Page Table Placement

We have mentioned that page tables tend to be very large. To get an idea of the page table size, we look at an example next:

Example 18.1 *Page table size calculation.*
Consider a system with 40-bit virtual addresses and 4 KB pages. The lower 12 bits are used to locate a byte in a given page. Then, the number of virtual pages is $2^{40} - 2^{12} = 2^{28}$. A typical page table entry is 4 bytes long. Thus, we need 4×2^{28} bytes (or 1 GB) of main memory for the page table! □

As the last example shows, a large virtual space requires huge page tables. For example, 64-bit processors such as the Itanium support a 64-bit virtual address space, requiring large page tables. With such large page tables, it is not feasible to place them in physical memory. The only solution is to map the page tables to virtual address space. This may sound tricky at first, but we show that this scheme really works. Since we don't need the entire page table at any time, we can partition the page table into pages (as we did with the user space) and bring only the pages of the table that are needed. We build a second-level page table to point to these first-level page table pages. We can recursively apply this procedure until we end up with a small page table that can sit in main memory.

Example 18.2 *Page size calculation for three-level hierarchical page tables.*
Let us continue with Example 18.1. Note that page tables at all levels use the same page size (i.e., 4 KB in our example). The 1 GB page table is partitioned into $2^{30}/2^{12} = 2^{18}$ pages. Therefore, the level-2 page table requires $2^{18} \times 4 = 2^{20}$ bytes (i.e., 1 MB). This still may be too large to keep the page table in main memory. If so, we can create one more level. This top level would have $2^{20}/2^{12} = 2^8$ PTEs. Since each PTE is 4 bytes long, we just need 1 KB memory to keep the top-level table. □

The organization of this three-level page table hierarchy is shown in Figure 18.6. The 40-bit virtual address is divided into four groups of bits. As usual, the lowest 12 bits are used for byte offset into the selected physical page. The highest 8 bits of the virtual address are used as an index into the level-3 page table. The base address of this page table is stored in a register. The entries in this table are used as the physical base address pointers to the level-2 page tables. The next 10 bits of the virtual address serve as an index into this 1024-PTE level-2 table. Repeating this process one more time for the level-1 page table gives us the final physical page number. As shown in Figure 18.2, the physical page number can be formed by combining the 12-bit byte offset value with the physical page number from the level-1 table.

As we show in Section 18.8, the Pentium uses a two-level hierarchy of page tables. Processors such as the Alpha use a four-level hierarchy. The hierarchical page table is also called the *forward-mapped page table* because the translation process proceeds from the virtual page number to the physical page number. We discuss another type of page table organization, called the inverted page table, in the next section.

18.5.1 Searching Hierarchical Page Tables

We can search the hierarchical page tables in one of two ways: *top down* or *bottom up*.

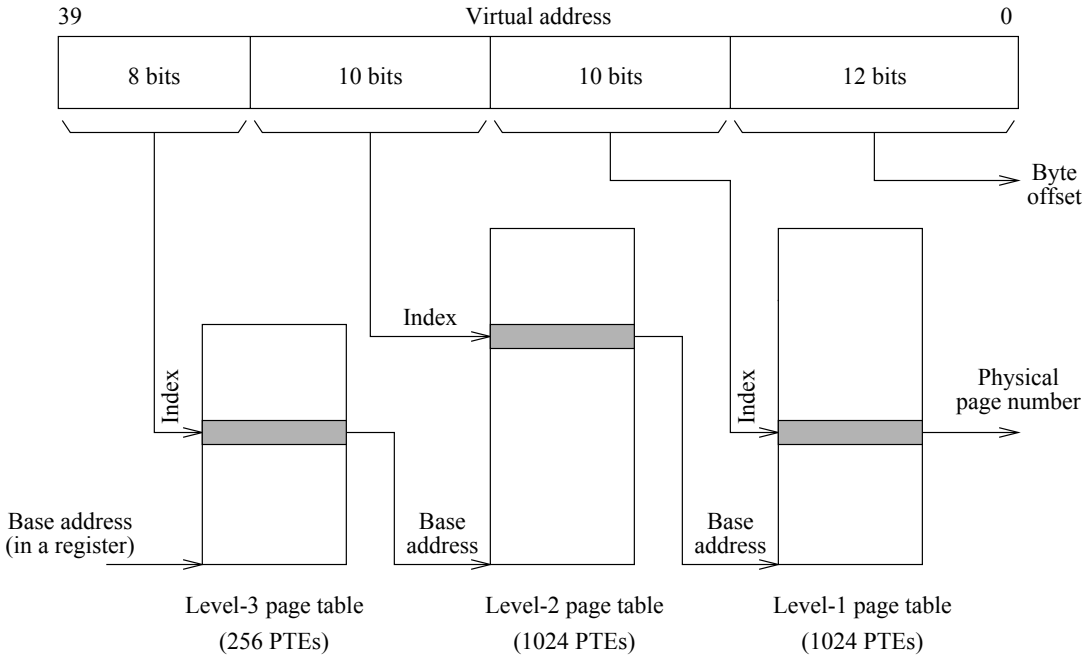


Figure 18.6 An example three-level hierarchical page table.

Top-Down Search: The top-down search follows a simple procedure. It starts at the root of the hierarchy and follows all the levels in the hierarchy. For the previous example, we need to access main memory four times to get the user-requested data: three accesses to the three page tables in the hierarchy and one additional access to read the user data.

Bottom-Up Search: To reduce this unacceptable overhead, processors such as the PowerPC and Alpha use a bottom-up search. In this search method, table lookup is first done at the bottom level. If the required page of the level-1 page table is in memory, the physical page number is obtained directly avoiding the hierarchical top-down search. If not, it resorts to the top-down search process. A detailed description of these two search methods is given by Jacob and Mudge [22].

18.6 Inverted Page Table Organization

One of the main problems with the forward page table organization is the size of the page table. The number of entries in this table is equal to the number of virtual pages. This can be quite large for modern 64-bit processors. Assuming a page size of 4 KB, these systems can have 2^{52} virtual pages. The reason for such large page tables is that we use VPN as an index into this table. One way to reduce the size of page tables is to use the physical page number as the index.

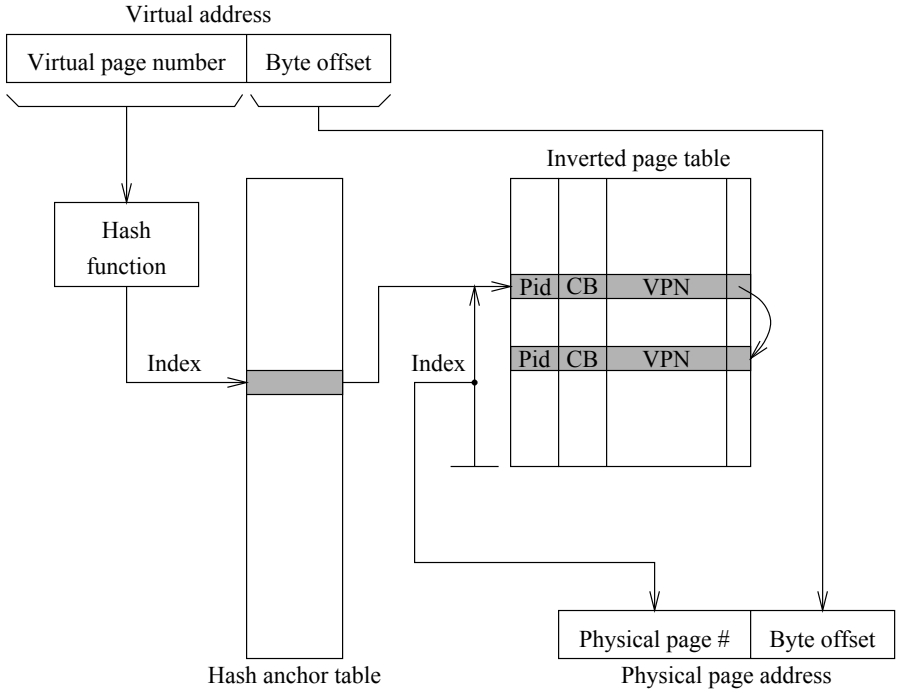


Figure 18.7 Structure of an inverted page table: The CB field represents the control bits such as valid and reference bits (see page 742). The number of entries in the inverted page table is equal to the number of physical pages. There is only one, systemwide, page table in this organization (as opposed to a forward page table for each process). Thus, we need to keep process id (Pid) in each PTE.

Such an organization is called the *inverted page table*. An advantage of the inverted page table is that it grows with the size of the main memory. There is only one systemwide inverted page table, as the entries in the inverted table are organized by the physical page number (PPN). In contrast, we will have a page table for each process in the forward page table organization.

The inverted page table leads to complications in page translation. Since the PPN is what we want to find for a given VPN, we cannot use the PPN as an index into the page table. To facilitate table lookup, the VPN is hashed to generate an index into the page table. A problem with hashing is that several VPNs may hash to the same index value. This is called *collision*. We can select a hash function that reduces the frequency of collision. However, we cannot avoid a certain degree of collision as the hash function is mapping a large number of virtual pages to a small number of physical page entries. Thus we need to have a mechanism to handle collisions.

Two techniques are used to handle collisions: open chaining and secondary hashing. In open chaining, when the hashed slot is not empty, the next open slot is used to store the PTE. This new slot is linked to the original slot as shown in Figure 18.7. This figure shows that the hash function output is not directly used as an index into the inverted page table. There is a

level of indirection by means of the hash anchor table (HAT). The rationale for using the HAT is to reduce the collision frequency. We can reduce collision frequency by increasing the page table size, but we don't want to increase it as it should be equal to the number of physical pages so that we can use the PPN as an index into the table. Furthermore, increasing the page table also increases the storage space due to long PTEs containing process ids, the chain pointer, and so on. The HAT provides an efficient way to increase the size: each entry in HAT just stores the pointer to the head of a chain.

In the second technique, when a collision occurs, a second hash function is applied to resolve the collision. This technique is used by the PowerPC, which uses the inverted page table. We describe this technique in more detail in Section 18.8.2.

The translation process is straightforward if the page is in memory. The requested VPN is hashed to get an index into the hash anchor table, which in turn gives the index value into the inverted table. Note that the page table stores the VPN to indicate which virtual page is in the physical page. The index gives the physical page number. If the requested page is not in memory, we will need to consult the standard page table to bring the page into memory. But this table can be kept on disk so size should not be a problem. Thus, in the inverted table method, the inverted table is kept in memory, and the forward page table is stored on disk.

18.7 Segmentation

Virtual address space is linear and one-dimensional as is the physical address space. Segmentation introduces a second dimension to the virtual address space. In segmentation, each process can have several virtual address spaces called *segments*. Each segment starts from address 0 to a system-specific maximum value. In the Pentium, for example, a segment can be as large as 4 GB. In a segmented memory, the address consists of two parts: a segment number and an offset within the segment.

The Pentium and PowerPC support segmented-memory architecture. Unlike paging, which is transparent to the programmer, segmentation is logical and is visible to the programmer. We have already seen how segmentation is visible in writing Pentium assembly programs. For example, even the simple assembly language program we have written used three segments: data, stack, and code. Even though we have not used them in our example programs, each program can have several segments.

Segmentation offers several key advantages, including the following.

- *Protection:* Customized protection can be provided on a segment-by-segment basis depending on the segment contents. For example, the code segment can be protected from writing into it (it could be made execute-only). Since segmentation is visible to the programmer, programs can be divided into logical segments that make sense. For example, we don't put code and data together in one segment. In contrast, protection provided at the page-level is transparent to the contents of the page. In paging systems, it is harder to achieve the level of protection one can get with segmentation.
- *Multiple Address Spaces:* Segmentation also provides multiple virtual address spaces. This is particularly useful when you have a dynamic data structure that grows and shrinks.

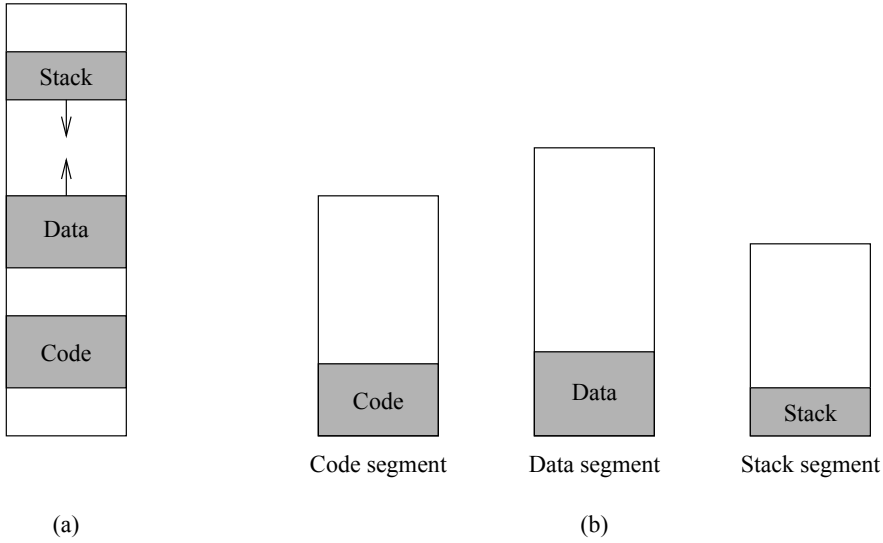


Figure 18.8 Dynamic data structure allocation: (a) in a single address space, boundary problems can lead to a growing data part bumping into the stack area; (b) segmentation avoids such boundary problems by allocating separate segments.

Stack and dynamic arrays are examples of such data structures. If you have a single address space, it is difficult to accommodate dynamic data structures. A growing data structure might bump into the next one. In Figure 18.8*a*, for instance, if the data part grows beyond the free space available, it may overwrite the stack area. With segmentation, we can assign each data structure its own segment (see Figure 18.8*b*) so that it can grow and shrink without the boundary problems. Of course there is a segment boundary problem we have to deal with but segments are usually very large. For example, the Pentium allows segments as large as the physical address space (i.e., segments can be up to 4 GB long).

- *Sharing Among Processes:* Since segmentation is logical, segments can be shared among the processes. For example, several processes can share library procedures. There is no need for each process to have a copy.

Another advantage of segmentation is that changes to a part of the program only require compilation of that part mapped to a segment. This is possible because each segment starts at address 0. For example, if a procedure is mapped to a segment, we need to just compile the procedure after it has been modified.

Even though both paging and segmentation support virtual address space, the underlying concepts of paging and segmentation are quite different.

- Paging uses fixed size blocks called pages. Pages of a contiguous virtual address space can be scattered in the physical memory. Paging is related to memory, not to the logical objects of the program. This makes paging transparent to the program.
- Segmentation uses variable size blocks called segments. Segments are related to the objects of the program. Segment sizes may vary during the execution of a program. As mentioned, protection and sharing are possible at the object level. Unlike paging, segmentation is visible to the program. Segmentation can leave gaps in main memory, as segments are created and destroyed. This checkerboard of holes in memory, shown in Figure 18.8*a*, is referred to as *external fragmentation*. In contrast, paging causes internal, but not external, fragmentation.

If segments are large, it is not possible to keep an entire segment in the main memory. We can apply the paging technique to each segment to keep only a part of the segment. Modern systems such as the Pentium and PowerPC use segmentation with paging. More details on this integration are given in the next section.

18.8 Example Implementations

We now look at the virtual memory implementations of Pentium, PowerPC, and MIPS processors.

18.8.1 Pentium

The Pentium supports both segmentation and paging, each of which can be selectively disabled. Thus, it is possible to operate in a purely segmented mode with paging turned off. As mentioned, this mode is not suitable if the segments are large. The Pentium can also work without using its segmentation capabilities; in this mode, the system can be thought of as a single segment. Indeed, UNIX and Linux systems use such a flat model and implement protection at the page level. The address translation mechanism of the Pentium is shown in Figure 7.11 on page 265. Segmentation translates a 48-bit logical address into a 32-bit linear address, which is translated into a physical address by the paging system. If paging is disabled, the 32-bit linear address is treated as the physical address.

Figure 18.9 shows details about segmentation with paging. We have already discussed the translation process involved in translating the logical address into a linear address (see Figure 7.12 on page 266). The 13-bit segment selector is used as an index into a segment descriptor table. The base of the segment descriptor table is stored in a register (LDTR or GDTR). Each segment descriptor provides a 32-bit base address, access rights to the segment, and the size of the segment. The 32-bit offset from the logical address is added to the 32-bit base address to derive the 32-bit linear address. More details on this translation process are on pages 265 to 270.

For page translation, the Pentium uses a two-level page table hierarchy. The root-level page table is referred to as the page directory. The page directory consists of 1024 page directory entries (PDEs) as shown in Figure 18.9. The physical address of the current page directory is

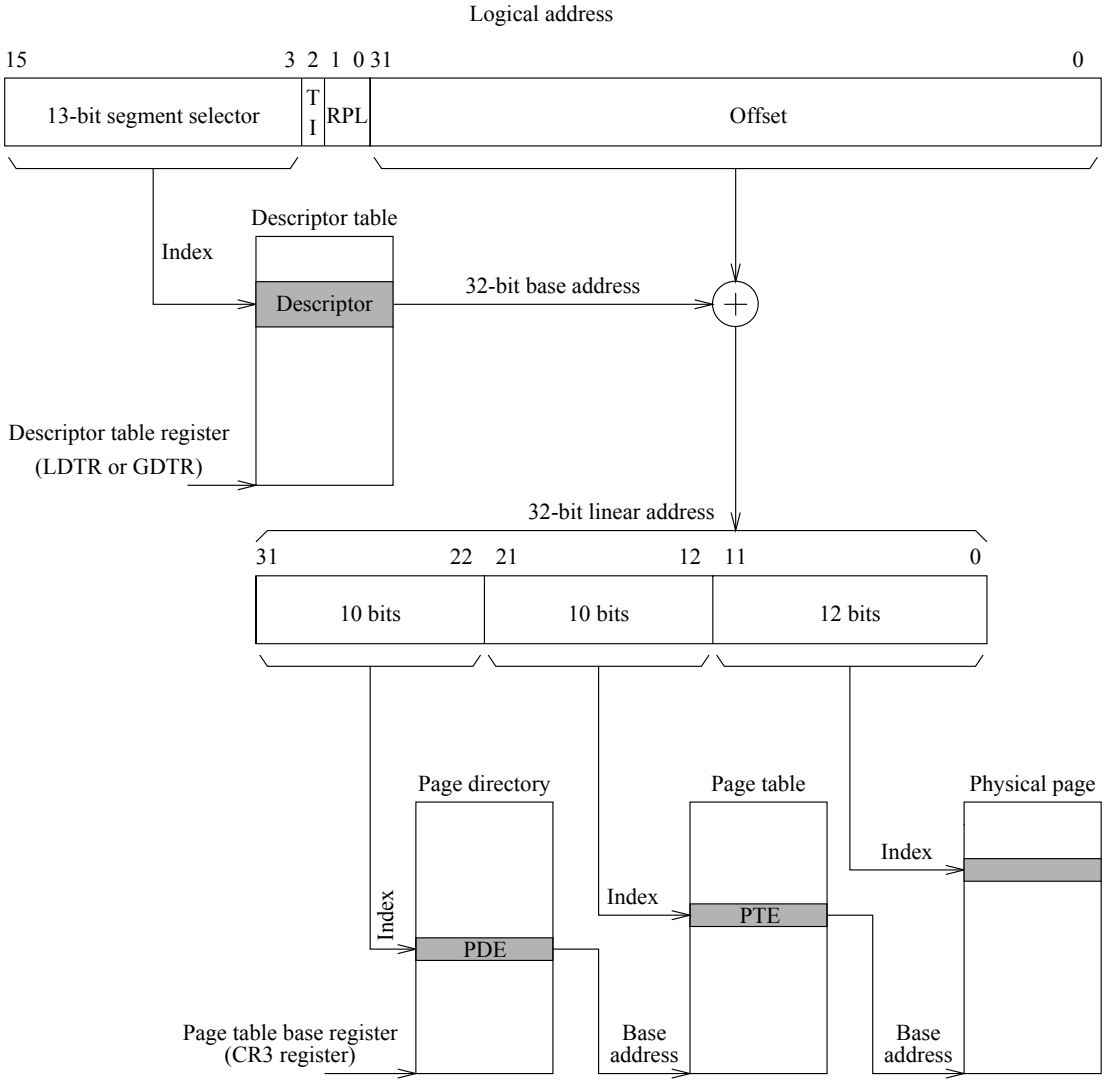


Figure 18.9 Logical address to physical address translation in the Pentium.

stored in the page directory base register (the CR3 register). The most significant 10 bits are used as an index into this table to select a PDE. The selected PDE provides the physical base address of a page table at the second level. The middle 10 bits are used to select a page table entry from this table, which gives the physical base address of the memory page. The 12-bit offset is used to select a location in this page. With both segmentation and paging, each segment can have its own page table as shown in Figure 18.10.

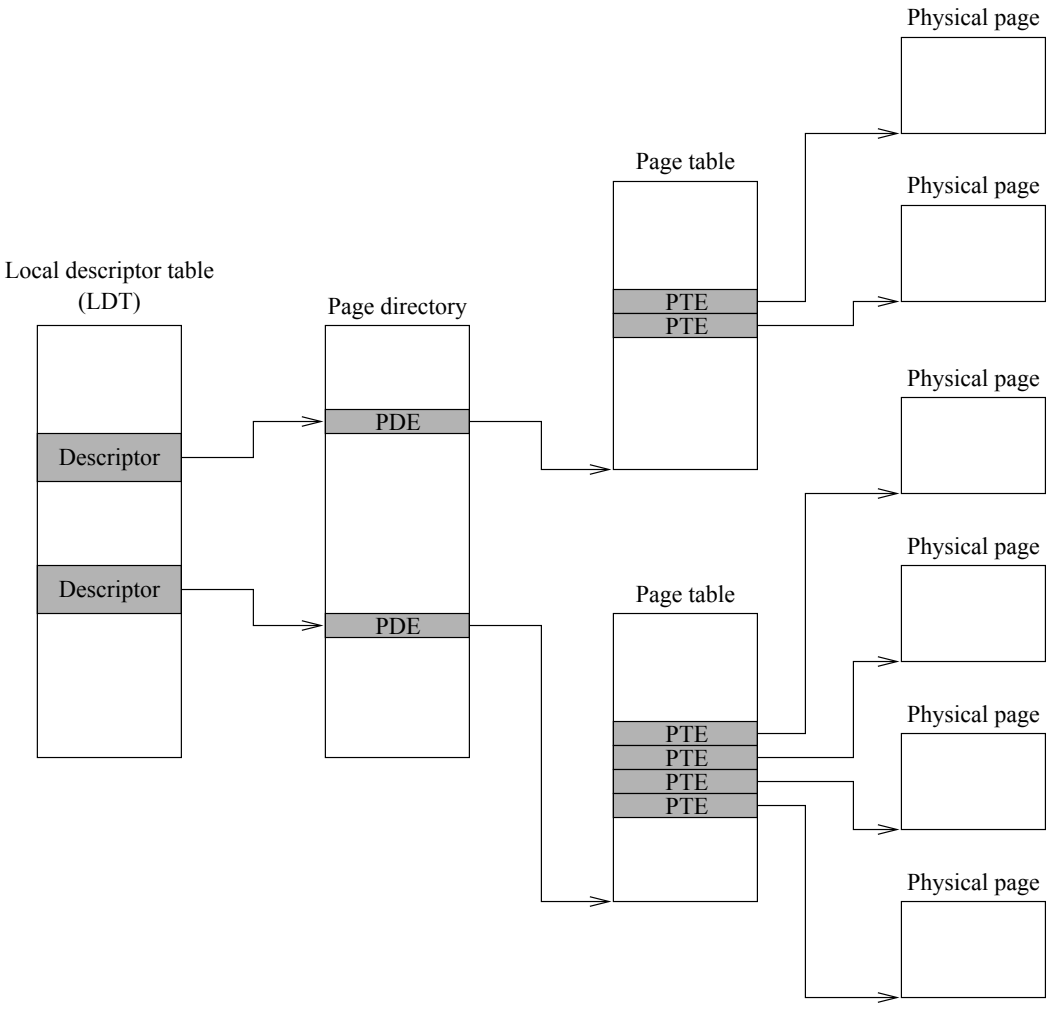
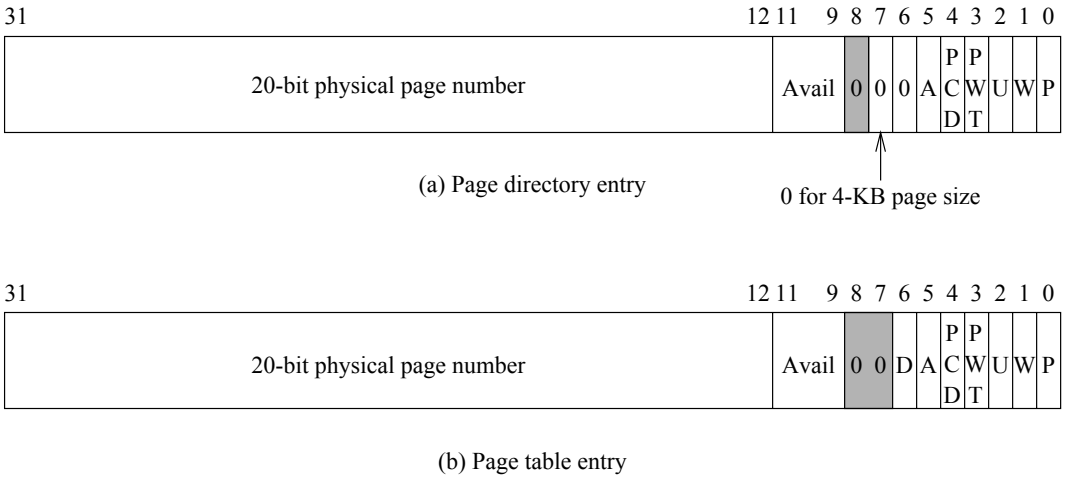


Figure 18.10 The Pentium provides support for each segment to have its own page table.

Page Table Entries

Page table entries are four bytes long. Since the default page size in the Pentium is 4 KB, each page can hold 1024 entries. The details of the entries in the page directory and table are shown in Figure 18.11. The PDE and PTE differ only in a couple of bits. So, we first discuss the common details.

Each entry maintains a 20-bit physical page address. This means that the page table and pages should be aligned at 4 K boundaries. Since a page table itself is a page, aligning pages at 4 K boundaries is sufficient. Bit 0 stores the valid bit to represent that the page is present in the



Bit 0 Page present bit (P)
Bit 1 Writes permitted (W)
Bit 2 User/supervisor (U)
Bit 3 Page-level write-through (PWT)
Bit 4 Page-level cache-disable (PCD)
Bit 5 Page accessed (A)
Bit 6 0 in PDT
dirty bit (D) in PTE
Bit 7 Page size in PDE (0 for 4 KB pages)

Avail: Available for system programmer use

Shaded bits: Reserved by Intel (must be zero)

Figure 18.11 Page directory and page table entry format: (a) in a page directory entry, bit 7 indicates page size information; (b) in a page table entry, bit 6 indicates whether the page has been written into.

memory. Note that when this bit is zero (i.e., P = 0), the remaining 31 bits in a PTE are available for the system programmer's use. Bit 1 indicates whether the page is read-only (W = 0) or read-write (W = 1). Bit 2 identifies two levels of privileged access to the page: supervisor or user level. Supervisor level (U = 0) is used for the operating system, system software such as device drivers, and protected data such as page tables. The user level (U = 1) is used to access user data and code. The U and W bits together can be used to implement page-level protection and access control.

Note that the two RPL bits of the segment selector are used to provide four levels of protected access. Figure 18.12 shows the four levels at the segment level and their typical use. In contrast, there are only two levels of privilege: supervisor and user. The four privilege levels of segmentation are mapped into the two privilege levels used for paging. Segmentation privilege levels 0, 1, and 2 are mapped to the supervisor level; level 3 is mapped to the user level.

The next two bits—PWT and PCD—are used to control page-level caching. The PCD bit can be used to selectively disable caching on a page-by-page basis. The PWT bit controls the

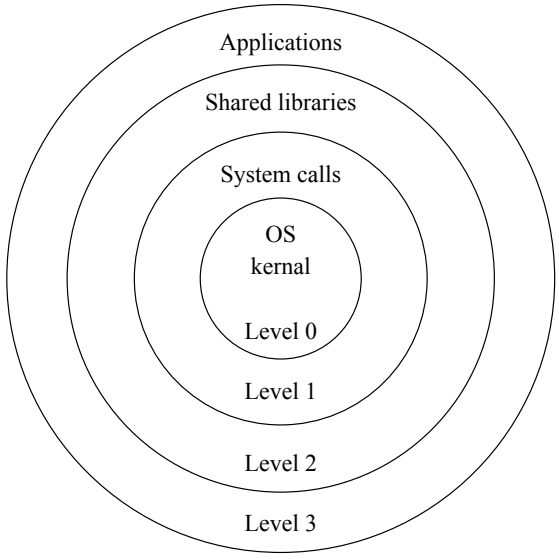


Figure 18.12 Pentium protection rings and their typical uses.

write policy; a write-through policy is used if PWT = 1; otherwise, a write-back policy is used. Both these bits are available for external caches through Pentium PCD and PWT pins.

Bit 5 in both the PDE and PTE is used as the reference bit (called the *accessed* bit). The Pentium sets the accessed bit A before allowing a read/write operation to the page. The OS can periodically clear this bit to implement a page replacement policy (see page 738). The dirty (D) bit indicates whether a write has taken place on the page. If the page is not modified, it does not have to be written back to disk at replacement time. The processor sets the dirty bit before a write operation. The OS must clear this bit when a page is brought into the memory.

The Pentium uses separate instruction and data TLBs. The instruction TLB uses a 32-entry, four-way set associative organization. The data TLB is similar except that it is a 64-entry TLB.

The Pentium supports segments that are larger than the size of a page when paging is used. In addition, it is also possible to pack several small segments into a single page. Just as with data alignment, the Pentium allows nonaligned page and segment boundaries. However, aligning the boundaries will lead to more efficient memory management.

18.8.2 PowerPC

As does the Pentium, the PowerPC supports both segmentation and paging. To facilitate comparison with the Pentium, we describe the 32-bit implementation of the PowerPC architecture. The PowerPC address translation method is shown in Figure 18.13.

As with the Pentium, both logical and physical addresses are 32-bits long, and pages are 4 KB in size. The 32-bit logical address consists of a 12-bit byte offset into a page, a 16-bit

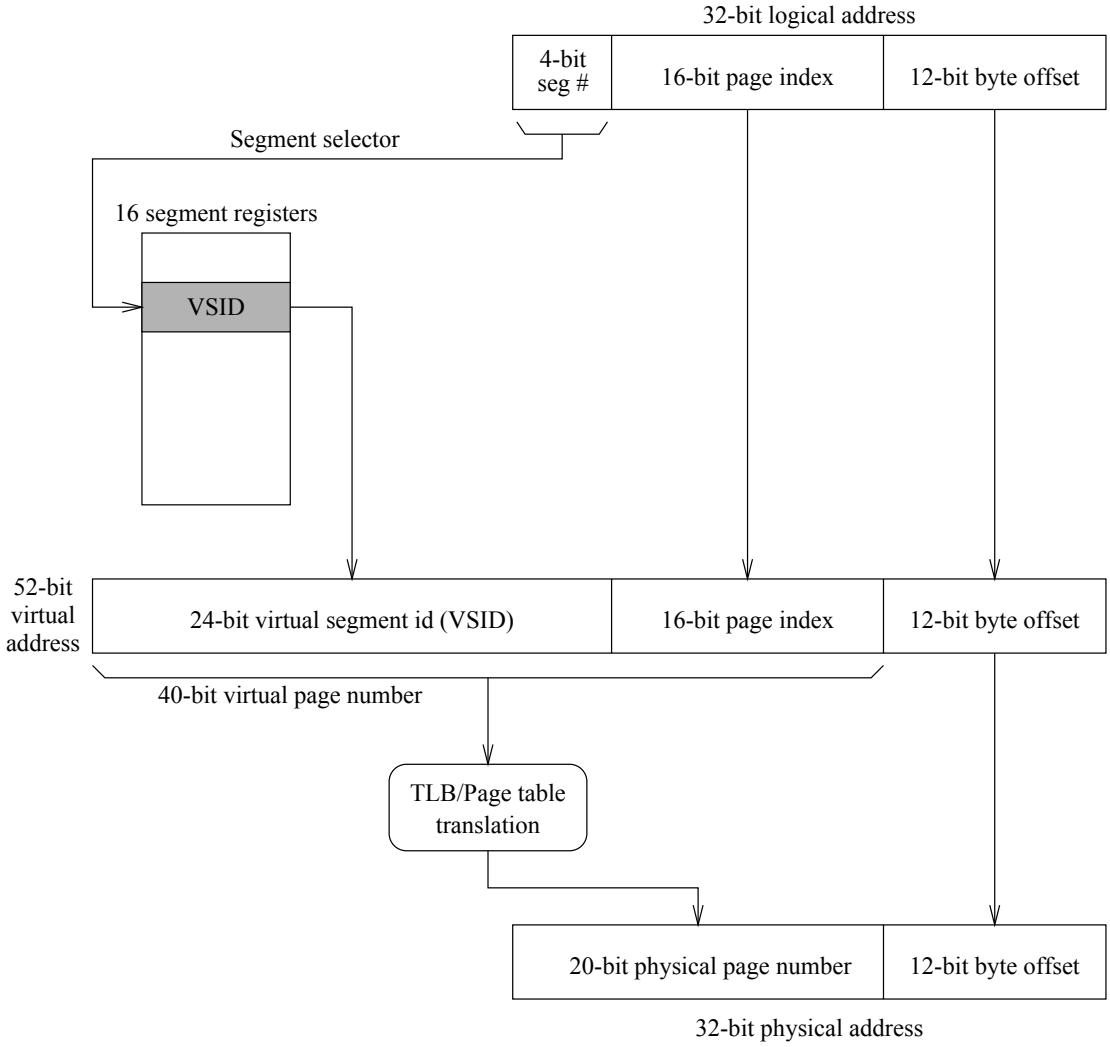


Figure 18.13 Logical address to physical address translation in the PowerPC.

page index, and a 4-bit segment number. The segment number field is used to select one of 16 segment registers. Each segment register contains a segment descriptor that is 32-bits wide. Each segment descriptor contains a 24-bit virtual segment id (VSID). Three of the remaining 8 bits are used for protection information. The 52-bit virtual address is formed by replacing the 4-bit segment number by the selected VSID. The virtual address consists of a 40-bit virtual page number and a 12-bit byte offset. The virtual page number is used to search the TLB/page table for a mapped 20-bit physical page number. When address translation is disabled, the virtual

address part is skipped; the 32-bit logical address is used as the 32-bit physical address. Since these details are qualitatively similar to the translation process of the Pentium, we focus on the differences between the two.

Unlike the Pentium, which provides four levels of page-level protection, the PowerPC provides eight levels. The PowerPC supports two modes: supervisor and supervisor/user. In each mode, four types of accesses are supported. In the supervisor mode, the following four access types can be specified: no-execute, write-only, write-only-no-execute, unrestricted access. The user/supervisor mode supports no-execute, read-only, read-only-no-execute, and unrestricted access. In contrast, the Pentium supports only two: read-only and read/write. However, it provides protection rings.

Page Table Organization

The PowerPC uses an inverted page table to speed up table lookup. The organization is different from the canonical inverted page table described in Section 18.6. The PowerPC does not use the hash anchor table, which reduces the number of memory accesses by one. Instead it uses two hash tables with the structure shown in Figure 18.14.

The PowerPC uses eight-way associative page table entry groups (PTEGs). The main idea behind this organization is to eliminate the need for collision chaining. Up to eight VPNs that map to the same hash table index can be stored in the primary hash table. If more than eight VPNs map to the same PTE group, a secondary hash function is used to map to the secondary hash table. The secondary hash function output is the 1's complement of the main hash function as shown in Figure 18.14. The output of the primary hash function is equal to the exclusive-or of the lower 19 bits of VSID and the 16-bit page index with three zeros padded.

In the PowerPC inverted table organization, the location of a PTE does not have any relation to the physical page number. Therefore, each PTE stores both physical and virtual page numbers. As a result, PTEs are longer than in forward page tables. PTEs are 8-bytes wide. Each PTE also stores a valid bit, a reference bit, and a dirty bit (called *changed* bit). In addition, two bits (W and I) in each PTE are used for cache control. The write-through (W) attribute controls the type of cache write policy. If W = 1, a write-through policy is used; otherwise, a write-back policy is used. If caching-inhibited (I) is 1, main memory is accessed by bypassing the cache.

The PowerPC uses split TLBs. Both instruction and data TLBs are 128 entries long and use two-way associative organizations.

18.8.3 MIPS

We discuss the MIPS R4000 processor MMU in detail. Specifics of the MIPS R2000/3000 and R10000 processors can be found in [23]. MIPS processors do not use segmentation; instead, they use address space identifiers (ASIDs) to provide protection as well as virtual address space extension. The R4000 processor can operate in either 32- or 64-bit mode. In 32-bit mode, the virtual address consists of an 8-bit ASID, a virtual page number (VPN), and a byte offset. Figure 18.15 shows these three components when a page size of 4 KB is used. The virtual address for each virtual address space, identified by ASID, is 2^{32} bytes. The number of virtual

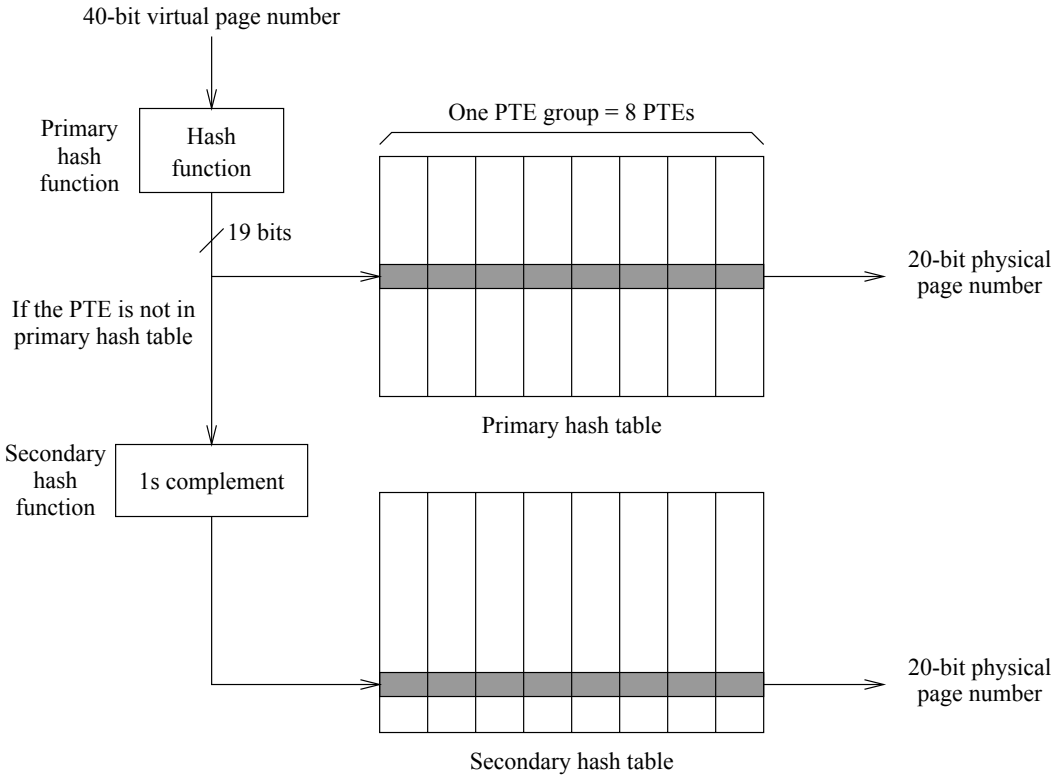


Figure 18.14 Hash table organization in the PowerPC.

pages depends on the page size used. The R4000 supports seven page sizes ranging from 4 KB to 16 MB in multiples of 4 (i.e., 4, 16, 64, or 256 KB, or 1, 4, or 16 MB).

Figure 18.16 shows the virtual to physical address translation mechanism when 16 MB pages are used. The physical address is 36-bits wide. In 64-bit mode, it supports a virtual address space of 2^{40} bytes. It still supports the same seven page sizes and 36-bit physical addresses. The ASID is 8-bits wide in both 32- and 64-bit modes. In the remainder of this section, we focus on the 32-bit mode.

By using an 8-bit ASID, the R4000 can support 256 different virtual address spaces. Logically, we can think of the total virtual address space as 2^{40} bytes. However, this virtual address space is more restrictive than the segmentation with paging used by the Pentium and PowerPC. The total virtual space of the R4000 consists of eight contiguous virtual spaces. The main difference from the Pentium and PowerPC architectures is that a virtual address space is allocated exclusively to a process. In the MIPS segmented architecture, allocation can be done at the page level. Thus, it easily supports sharing of pages by different processes.

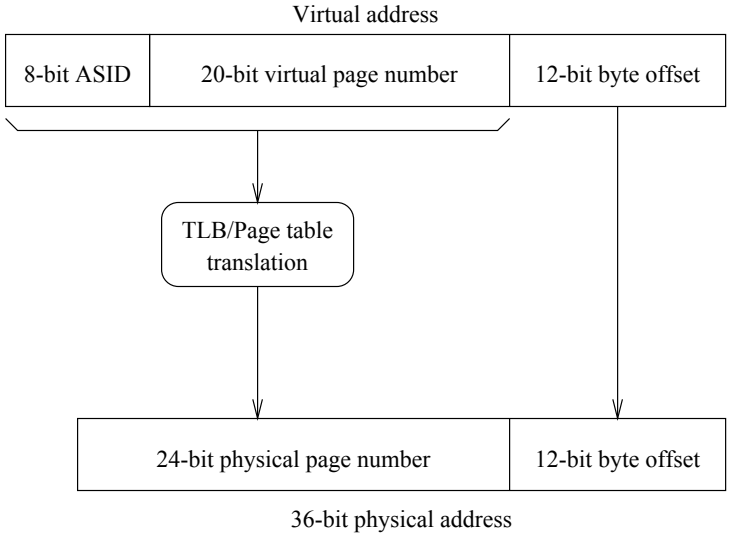


Figure 18.15 Virtual to physical address translation in the MIPS R4000 processor (with 4 KB pages).

TLB Organization

The MIPS TLB is organized differently from that discussed in the last two sections. It uses the fully associative organization with 48 entries for odd and even virtual pages (for a total of 96 pages). Each TLB entry is 128-bits wide as shown in Figure 18.17.

The 12-bit mask in each PTE specifies the page size. Since the mask information is associated with each PTE, variable page sizes can be implemented on a page-by-page basis. The VPN2 field gives the virtual page number of an odd/even page pair. That is, VPN2 is the virtual page number divided by 2. Thus each PTE maps two contiguous odd/even virtual pages. This format saves on the number of VPNs to store. If the global (G) bit is set, the processor ignores ASID in the TLB lookup. This feature is useful for sharing virtual pages among the processes. EntryLo0 and EntryLo1 give the physical page numbers of even and odd pages, respectively. Each of these entries has three fields: three C bits, one D bit, and one V bit. The C bits are used for cache control to indicate uncached, cached noncoherent, and three types of cache coherent modes. The D bit is used to indicate that the page is writable (a different meaning than the usual dirty page). This bit can be used to make the page read-only. The V bit indicates that the page entry is valid.

The TLB entries are divided into wired and random entries. Wired entries, which are the lower part of the TLB, are fixed and cannot be changed by a TLB write instruction. These entries are reserved for exclusive use by the OS. Random entries can be updated. The boundary between wired and random entries can be specified by software by writing into the wired register.

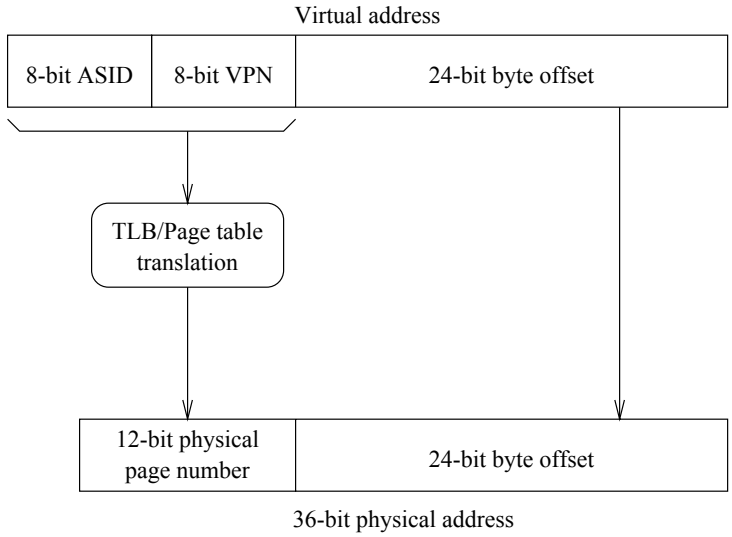


Figure 18.16 Virtual to physical address translation in the MIPS R4000 processor (with 16 MB pages).

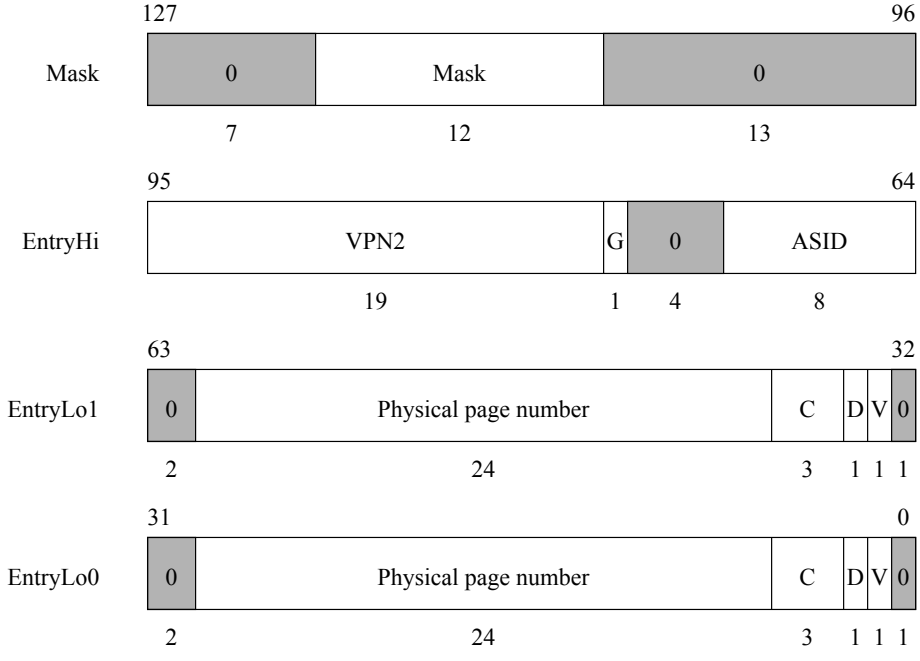


Figure 18.17 The MIPS R4000 TLB entry format.

The R4000 supports two TLB entry replacement policies: random and indexed. In the random policy, the processor randomly selects a TLB entry. In the other policy, software can specify the entry to be replaced. Two registers—Random and Index—support these two policies. Random register value is used in the random policy. This register is decremented with each instruction executed. The value of the random register is varied within the range of TLB random entries. Random replacement can be done by the TLBWR instruction.

The indexed replacement policy selects the entry specified by the Index register. We can use the TLBWI instruction to perform indexed replacement. The Index register also specifies the TLB entry read by the TLBR (TLB Read) instruction.

18.9 Summary

Virtual memory shares several basic concepts with the cache systems discussed in the last chapter. Virtual memory has been proposed to eliminate the main memory size restrictions and to allow sharing of memory among multiple programs running concurrently. In addition, virtual memory facilitates relocation of code by running each program in its own virtual address space.

The first objective of providing an illusion of much larger memory is similar to that of the cache systems. Consequently, cache and virtual memory systems share many of the basic concepts. However, there is also a significant difference between the two systems. In cache memories, the miss penalty is tens of clocks. On the other hand, virtual memory systems experience a several orders of magnitude (i.e., millions of clocks) larger miss penalty, as the lower level is a disk device. This difference dictates a different implementation for the virtual memory.

Analogous to cache lines, virtual memories use fixed-sized pages. A typical page size is 4 KB. Virtual memory can be thought of as a mapping function from a large virtual address space to a much smaller physical address space. A page table is used to translate virtual page addresses to physical page addresses. For large virtual address spaces, this table can be very large. For example, 64-bit processors such as the Itanium support a virtual address space of 2^{64} bytes. If we use 4 KB pages, we will have 2^{52} virtual pages. In a conventional page table organization, there is an entry for each virtual page. Therefore the table size is proportional to the number of virtual pages. Thus, the larger virtual address space of recent 64-bit processors forces us to use bigger pages. For example, the MIPS processor allows the page size to be as large as 16 MB. In the previous example, if we use 16 MB pages instead of 4 KB pages, we would reduce the number of virtual pages from 2^{52} to 2^{40}.

In cache memory, translation is done on the fly in hardware. However, in virtual memory, memory accesses by programs take at least twice as long. To speed up the translation, most systems maintain the recent address translations in a special cache called the translation lookaside buffer. We have presented details on the TLB organization.

We have also discussed a technique to reduce the page table size. The inverted page table organization requires the number of page table entries to be proportional to the physical address space as opposed to the virtual address space. This organization, however, complicates the virtual-to-physical page translation.

Virtual address space is one-dimensional as is the physical memory. Processors such as the Pentium and PowerPC use segmentation to add another dimension to the virtual address space. In segmentation, each process can have several virtual address spaces. We have discussed the concepts involved in implementing segmentation.

The last section described virtual memory implementations of Pentium, PowerPC, and MIPS processors. As mentioned, both the Pentium and PowerPC use segmentation.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Dirty bit
- External fragmentation
- FIFO replacement policy
- Internal fragmentation
- Inverted page table
- Least recently used (LRU) policy
- Locality
- Memory management unit (MMU)
- Multiple address spaces
- Not frequently used (NFU) policy
- Overlays
- Page fault
- Page frames
- Page table entries (PTEs)
- Page table hierarchy
- Page table organization
- Page table placement
- Protection bits
- Reference bit
- Replacement policies
- Second chance replacement policy
- Segmentation
- Translation lookaside buffer (TLB)
- Valid bit
- Virtual address
- Virtual page number
- Virtual pages
- Write policies
- Write-through

18.10 Exercises

18–1 What is the motivation for proposing virtual memory?

18–2 What are the similarities between cache memory and virtual memory?

18–3 What are the differences between cache memory and virtual memory?

18–4 In the context of virtual memory, we talk about virtual address and physical address. Describe the process of translating virtual addresses to physical addresses.

18–5 What is the purpose of the valid bit in a page table entry? Is it absolutely necessary to keep this bit in the PTE?

18–6 What is the purpose of the dirty bit in a page table entry? Is it absolutely necessary to keep this bit in the PTE?

18–7 What is the purpose of the reference bit in a page table entry? Is it absolutely necessary to keep this bit in the PTE?

18–8 Describe the information stored in a typical PTE. In your answer, do not include the valid, dirty, and reference bits.

18–9 Consider a Pentium-like processor that uses 4 KB pages to support a 32-bit virtual address space. Each PTE is 4 bytes. Calculate the page table size in bytes.

18–10 In the last exercise, suppose that the processor supports a 64-bit virtual address space. Calculate the size of the page table in bytes. Use the same page size and 4-byte PTEs.

18–11 In the last exercise, we used 4 KB pages. In this exercise, assume that we use 16 MB pages. Calculate the page table size using the same virtual address space and PTE size.

18–12 Page size is an important parameter in virtual memory design. Discuss the impact of page size on the cost and performance.

18–13 Explain why the write-through policy can be used in caches but not in virtual memory implementations.

18–14 In hierarchical page tables, page table size is proportional to the number of virtual pages. Explain the reason for this size.

18–15 Explain why inverted page tables reduce the page table size.

18–16 Explain the search process used in the inverted page table.

18–17 Explain why it is necessary to map page tables to virtual address space. What problems do we encounter in mapping page tables to physical address space?

18–18 Consider a system with a 48-bit virtual address space and a 32-bit physical address space. If we use 16-KB pages, find the number of PTEs in a conventional page table (i.e., in a forward-mapping page table).

18–19 In the last exercise, recalculate the number of PTEs if we use the inverted page table.

18–20 Consider a system with a 64-bit virtual address space. Assume that the page size is 4 KB and each PTE is 4 bytes long. How many levels of hierarchy would you consider in implementing the page table? Assume that we have less than 1 MB for the top-level page table.

18–21 In the last exercise, suppose we use 64 KB pages instead of 4 KB pages. Will it decrease the number of levels in the hierarchy? If so, by how many levels?

18–22 The Pentium and PowerPC support segmentation. What are the advantages of segmentation?

18–23 What is the motivation for combining segmentation with paging?

18–24 Why do most of the operating systems ignore segmentation?

18–25 Can we completely turn off the segmentation feature in the Pentium? How do we do this?

18–26 What is internal fragmentation? Which system—paging or segmentation—causes internal fragmentation?

18–27 What is external fragmentation? Which system—paging or segmentation—causes external fragmentation?

18–28 We have discussed virtual memory implementations of three processors. Of these three, only the PowerPC uses the inverted page table. Describe the inverted page table organization of the PowerPC.

Chapter 19

Input/Output Organization

Objectives

- To discuss the basics of I/O addressing and access;
- To describe I/O data transfer techniques;
- To introduce programmed I/O;
- To present details on direct memory access (DMA);
- To present details on external interfaces including parallel and serial buses such as EIA-232, SCSI, USB, and FireWire.

This chapter looks at the input/output (I/O) interface to the system. Computer systems typically have several I/O devices, from slow devices such as the keyboard to high-speed disk drives and communication networks. Irrespective of the type of device, the underlying principles of interfacing an I/O device are the same. This interface typically consists of an I/O controller. We describe these details in the first section.

I/O devices contain several internal registers. These registers are used, for example, to receive commands from the processor and to supply the status of the I/O operation. We need to understand two basic issues. How do we map these internal registers? How does the processor access these registers? Section 19.2 presents these details. The next section describes how the keyboard is interfaced to the system. Section 19.4.1 uses the keyboard to illustrate the programmed I/O technique, which is one of the ways to transfer data to an I/O device. There are other means of transferring data. In programmed I/O, the CPU directs the transfer of data. Direct memory access (DMA) transfers data without involving the processor in the transfer process. The DMA process is described in Section 19.4.2.

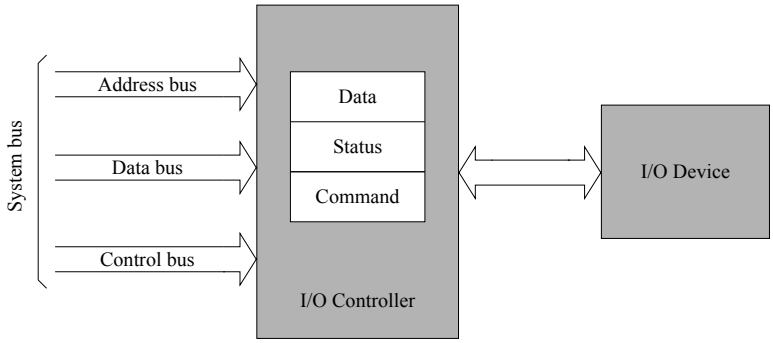


Figure 19.1 Block diagram of a generic I/O device interface.

External interfaces use longer wires than the internal buses. Furthermore, there can be external interferences affecting the quality of data transmission. Thus the probability of transmission error is higher in external cables than on internal buses. Because of this possibility, we need error detection capability. It is also nice to have error correction capability. We discuss error detection and correction codes in Section 19.5. This section presents details on parity encoding and cyclic redundancy check (CRC) codes.

Data transmission can be done in one of two basic ways: serial or parallel. Section 19.6 describes these two modes of data transmission. This section also presents details on three interface standards: EIA-232, parallel interface, and SCSI. The next two sections present details on two external serial buses: the universal serial bus (USB) and IEEE 1394 (also known as FireWire). The chapter concludes with a summary.

19.1 Introduction

Input/output devices provide the means by which a computer system can interact with the outside world. An I/O device can be an input device (e.g., keyboard, mouse), an output device (e.g., printer, display screen), or both an input and output device (e.g., disk drive).

Computers use I/O devices, also called *peripheral devices*, for two main purposes: to communicate with the outside world, and to store data. I/O devices such as printers, keyboards, and modems are used for communication purposes and devices like disk drives are used for data storage. Regardless of the intended purpose of the I/O device, all communications with these devices must involve the systems bus. However, I/O devices are not directly connected to the system bus. Instead, there is usually an *I/O controller* that acts as an interface between the system bus and the I/O device, as shown in Figure 19.1.

There are two main reasons for using an I/O controller. First, different I/O devices exhibit different characteristics and, if these devices were connected directly, the CPU would have to understand and respond appropriately to each I/O device. This would cause the CPU to spend a lot of time interacting with I/O devices and spend less time executing user programs. If we

use an I/O controller, this controller could provide the necessary low-level commands and data for proper operation of the associated I/O device. Often, for complex I/O devices such as disk drives, there are special I/O controller chips available.

The second reason for using an I/O controller is that the amount of electrical power used to send signals on the system bus is very low. This means that the cable connecting the I/O device has to be very short (a few centimeters at most). I/O controllers typically contain driver hardware to send current over long cables that connect I/O devices.

I/O controllers typically have three types of internal registers—a data register, a command register, and a status register—as shown in Figure 19.1. When the CPU wants to interact with an I/O device, it communicates only with the associated I/O controller.

To focus our discussion, let us consider printing a character on the printer. Before the CPU sends a character to be printed, it has to first check the status register of the associated I/O controller to see whether the printer is online/offline, busy or idle, out of paper, and so on. In the status register, three bits can be used to provide this information. For example, bit 4 can be used to indicate whether the printer is online (1) or offline (0), bit 7 can be used for busy (1) or not busy (0) status indication, and bit 5 can be used for out of paper (1) or not (0). The I/O controller gets this status information from the interface (see Section 19.6.2 on page 797).

The data register holds the character to be printed and the command register tells the controller the operation requested by the CPU (e.g., send the character in the data register to the printer). The following summarizes the sequence of actions involved in sending a character to the printer:

- Wait for the controller to finish the last command. This can be done by repeatedly checking bit 7 of the status register.
- Place a character to be printed in the data register.
- Set the command register to initiate the transfer.

The CPU accesses the internal registers of an I/O controller through what are called *I/O ports*. An I/O port is simply the address of a register associated with an I/O controller.

What we have just described is a simple protocol to transfer data to the printer. Complex I/O devices have more complex protocols to facilitate communication. Nevertheless, this simple example brings out the two main problems in communicating with an I/O device:

- We need to access various registers such as the data and status registers. How can the CPU get access to the I/O controller's registers? It depends on the I/O mapping. The next section presents two mapping methods to access I/O devices.
- The mapping method provides the basic means of accessing the I/O device. Still, we have to figure out a protocol to communicate with the I/O device. In our printer example, we presented a simple protocol to transmit data. It essentially loops on the busy bit and transmits a byte whenever the busy bit is off. This is just one way of communicating with the device. Later, we look at two other ways.

19.2 Accessing I/O Devices

19.2.1 I/O Address Mapping

As discussed in Section 1.7, there are two ways of mapping I/O ports: memory-mapped I/O and isolated I/O. Memory-mapped I/O maps I/O port addresses to memory address space. Processors such as the PowerPC and MIPS support only memory-mapped I/O. In these systems, writing to an I/O port is similar to writing to a memory location. Memory-mapped I/O does not require any special consideration from the processor. Thus, all processors inherently support memory-mapped I/O.

Isolated I/O maintains an *I/O address space* that is separate from the memory address space. The Pentium supports isolated I/O. In these systems, special I/O instructions are needed to access the I/O address space. The Pentium provides two basic I/O instructions—`in` and `out`—to access I/O ports. The next subsection gives details on the Pentium input/output instructions.

The Pentium provides 64 KB of I/O address space. This address space can be used for 8-, 16-, and 32-bit I/O ports. However, the combination cannot exceed the total I/O address space. For example, we can have 64 K 8-bit ports, 32 K 16-bit ports, 16 K 32-bit ports, or a combination of these that fits the 64 K address space. As I/O instructions do not go through segmentation and paging units, the I/O address space refers to the physical address rather than the linear address.

Systems designed with processors supporting the isolated I/O have the flexibility of using either the memory-mapped I/O or isolated I/O. Typically, both strategies are used. For instance, devices like printer or keyboard could be mapped to the I/O space using the isolated I/O strategy; the display screen could be mapped to a set of memory addresses using the memory-mapped I/O.

19.2.2 Accessing I/O Ports

In a memory-mapped system, we can use the standard memory access instructions to access I/O ports. Therefore, we focus on the isolated I/O scheme, as it requires special I/O instructions.

To facilitate access to the I/O ports, the Pentium provides two types of instructions: register and block I/O instructions. Register I/O instructions are used to transfer a single data item (byte, word, or doubleword) between a register and an I/O port. Block I/O instructions are used for block transfer of data between memory and I/O ports.

Register I/O Instructions

The Pentium provides two register I/O instructions: `in` and `out`. The `in` instruction is used to read data from an I/O port, and the `out` instruction to write data to an I/O port. A port address can be any value in the range 0 to FFFFH. The first 256 ports (i.e., ports with addresses in the range 0 to FFH) are directly addressable (i.e., the address is given as a part of the instruction) by the `in` and `out` instructions.

The in/out instructions can be used to read/write 8-, 16-, or 32-bit data. Each instruction can take one of two forms, depending on whether a port is directly addressable. The general formats of the in instruction are

```
in      accumulator,port8 — direct addressing format,
in      accumulator,DX    — indirect addressing format.
```

The first form uses the direct addressing mode and can only be used to access the first 256 ports. In this case, the I/O port address, which is in the range 0 to FFH, is given directly by the port8 operand. In the second form, the I/O port address is given indirectly via the DX register. The contents of the DX register are treated as the port address.

In either form, the first operand accumulator must be AL, AX, or EAX. This choice determines whether a byte, word, or doubleword is read from the specified port.

The corresponding forms for the out instruction are

```
out     port8,accumulator — direct addressing format,
out     DX,accumulator    — indirect addressing format.
```

Notice the placement of the port address. In the in instruction, it is the source operand and in the out instruction, it is the destination operand signifying the direction of data movement.

Block I/O Instructions

The Pentium also supports two block I/O instructions: ins and outs. These instructions can be used to move blocks of data between I/O ports and memory. These I/O instructions are, in some sense, similar to the string instructions discussed in Chapter 12. For this reason, block I/O instructions are also called string I/O instructions. In fact, ins stands for "input string" and outs for "output string." As with the string instructions, ins and outs do not take any operands. Also, we can use the repeat prefix rep as in the string instructions.

For the ins instruction, the port address should be placed in DX and the memory address should be in ES:(E)DI. The address size determines whether the DI or EDI register is used (see Chapter 7 for details). Block I/O instructions do not allow the direct addressing format for the I/O port specification.

For the outs instruction, the memory address should be in DS:(E)SI, and the I/O port should be specified in DX. You can see the similarity between the block I/O instructions and the string instructions.

We can use the rep prefix with ins and outs instructions. However, we cannot use the other two prefixes—repe and repne—with the block I/O instructions. The behavior of the rep prefix is similar to that in the string instructions. The direction flag (DF) determines whether the index register in the block I/O instruction is decremented (DF = 1) or incremented (DF = 0). The increment/decrement value depends on the size of the data unit transferred. For byte transfers the index register is updated by 1. For word and doubleword transfers, the corresponding values are 2 and 4, respectively. The size of the data unit involved in the transfers can be specified as in the string instructions. We use insb and outsb for byte transfers, insw and outsw for word transfers, and insd and outsd for doubleword transfers.

19.3 An Example I/O Device: Keyboard

To concretize our discussion of I/O device access and communication, let's write a program to read input from the keyboard. We start with a description of the keyboard and how it is interfaced to the system.

19.3.1 Keyboard Description

We talked about I/O controllers associated with the devices to facilitate interfacing. For the keyboard, there is a keyboard controller (a chip dedicated to servicing the keyboard) that scans the keyboard and reports key depressions and releases. The keyboard controller supplies the key identity by means of a scan code. The *scan code* of a key is simply an identification number given to the key based on its location on the keyboard. The counting for the scan code starts at the top right-hand side of the main keyboard (i.e., with the `Esc` key) and proceeds left to right and top to bottom. Thus, the scan code for the `Esc` key is 1, the next key `1` is 2, and so on. Table 19.1 shows the scan codes for the standard PC keyboard.

The scan code of a key does not have any relation to the ASCII value of the corresponding character. The input routine will have to derive the ASCII value from the key scan code. Later we show how this translation is done.

The system interfaces the keyboard via an 8-bit parallel I/O port, originally supported by a peripheral interface chip. We present details on this chip next.

19.3.2 8255 Programmable Peripheral Interface Chip

The 8255 programmable peripheral interface (PPI) chip was developed for use with the 16-bit 8086 processor. Even though current systems do not use this chip, it provides a simple example to help understand how I/O devices are interfaced. We use this interface to write a sample keyboard program. The 8255 chip provides three 8-bit general-purpose registers that can be used to interface I/O devices. These three registers—called PA, PB, and PC—are mapped to the I/O space shown in Table 19.2.

Since we are interested in the keyboard, we need to know details only about the PA port, as this port provides the keyboard interface. The hardware within the keyboard scans the keys to check the state of the keys (i.e., depressed or released). The scan code of the key whose state has changed (i.e., depressed or released) is provided by the keyboard controller at the PA port.

The scan code of the key can be read from the PA port. Bits PA0 to PA6 give the scan code of the key whose state has changed. PA7 is used to indicate the current state of the key (up or down):

PA7 = 0 — key is depressed,
PA7 = 1 — key is released.

For example, if the `Esc` key is pressed, PA supplies 01H as 1 is the scan code for the `Esc` key. When the `Esc` key is released, PA supplies 81H. We look at an example later in this chapter.

To get an idea of how I/O port mapping is done, let's look at mapping the four 8255 ports

Table 19.1 Keyboard scan codes

Key	Scan code		Key	Scan code		Key	Scan code	
	Dec	Hex		Dec	Hex		Dec	Hex
Alphanumeric keys								
A	30	1E	M	50	32	Y	21	15
B	48	30	N	49	31	Z	44	2C
C	46	2E	O	24	18	1	02	02
D	32	20	P	25	19	2	03	03
E	18	12	Q	16	10	3	04	04
F	33	21	R	19	13	4	05	05
G	34	22	S	31	1F	5	06	06
H	35	23	T	20	14	6	07	07
I	23	17	U	22	16	7	08	08
J	36	24	V	47	2F	8	09	09
K	37	25	W	17	11	9	10	0A
L	38	26	X	45	2D	0	11	0B
Punctuation keys								
`	41	29	[	26	1A	,	51	33
–	12	0C	]	27	1B	.	52	34
=	13	0D	;	39	27	/	53	35
\	43	2B	′	40	28	space	57	39
Control keys								
Esc	01	01	Caps Lock	58	3A	Right Shift	54	36
Backspace	14	0E	Enter	28	1C	Ctrl	29	1D
Tab	15	0F	Left Shift	42	2A	Alt	56	38
Function keys								
F1	59	3B	F5	63	3F	F9	67	43
F2	60	3C	F6	64	40	F10	68	44
F3	61	3D	F7	65	41	F11	133	85
F4	62	3E	F8	66	42	F12	134	86
Numeric keypad and other keys								
1/End	79	4F	6/→	77	4D	Del/.	83	53
2/↓	80	50	7/Home	71	47	Num Lock	69	45
3/Pg Dn	81	51	8/↑	72	48	–	74	4A
4/←	75	4B	9/Pg Up	73	49	+	78	4E
5	76	4C	0/Ins	82	52			
Print Screen	55	37	Scroll Lock	70	46			

Table 19.2 8255 port address mapping

8255 register	Port address
PA (input port)	60H
PB (output port)	61H
PC (input port)	62H
Command register	63H

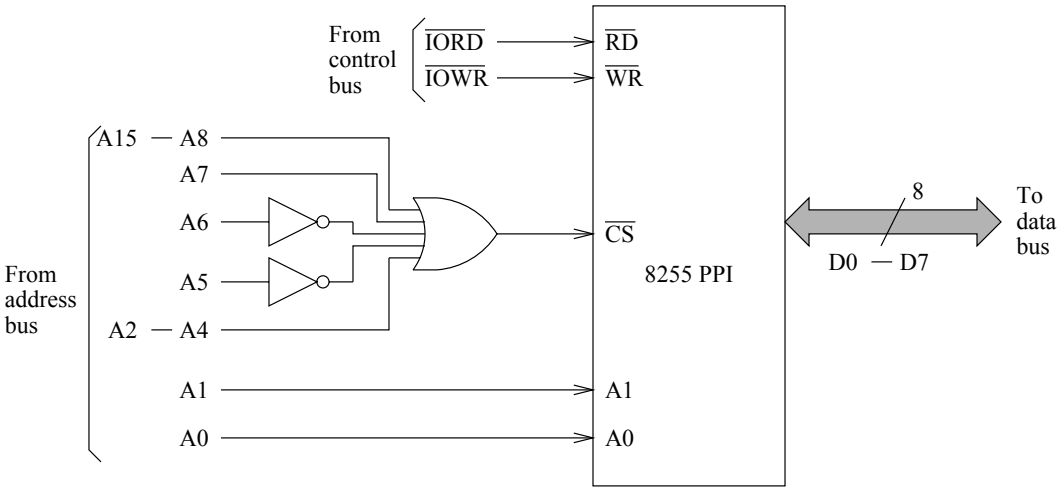


Figure 19.2 A simplified design to map the 8255 I/O ports to the I/O address space in Table 19.2.

to addresses 60H to 63H (see Table 19.2). The mapping is very similar to the memory mapping we have discussed in Section 16.6 on page 681. The only difference is that we use I/O read and write signals rather than memory read and write, as we are mapping the I/O ports into the I/O address space. Figure 19.2 shows a simplified digital logic circuit to map the four I/O ports.

Memory-mapped I/O logic is very similar to the memory mapping discussed in Section 16.6 on page 681. In this mapping, memory read and write signals are used to read and write from the I/O ports.

19.4 I/O Data Transfer

We have discussed various ways I/O devices can be accessed by a system. Now we have to build a protocol to transfer data between the system and an I/O device. When we talk about system, we generally mean the main memory. The data transfer process involves two distinct phases:

a data transfer phase and an end-notification phase. The data transfer phase transmits data between the memory and I/O device. This can be done by *programmed I/O* or *direct memory access*. The end-notification informs the processor that the data transfer has been completed. The processor gets this information either by an interrupt or through the programmed I/O mechanism. Typically, DMA-based data transfer uses an interrupt to indicate the end of data transfer, whereas the programmed I/O uses the other method. Thus, to understand I/O data transfer, we have to look at three basic techniques: programmed I/O, interrupt-driven I/O, and DMA.

Programmed I/O involves the processor in the I/O data transfer. Let us consider a supervisor–worker example. Assume that the supervisor gives a task to a worker. One way to know whether the task has been completed is to bug the worker periodically with the, "Did you finish?" question. Obviously, the manager would be wasting a lot of time. Programmed I/O is very similar. The processor repeatedly checks to see if a particular condition is true. Typically, it busy-waits until the condition is true. We used this busy-wait technique in our printer example on page 769. From this brief description, it should be clear that the programmed I/O mechanism wastes processor time.

In the supervisor–worker example, the supervisor can save a lot of time and energy if she stays put and lets the worker notify her when the task is done. Interrupt-driven I/O uses this concept. The processor assigns a task to an I/O controller and resumes its pending work. When the task is completed, the I/O controller notifies the processor by using an interrupt signal. Obviously, this is a better way of using the processor. However, an interrupt-driven mechanism requires hardware support, which is provided by all processors.

The last technique, DMA, relieves the processor of the low-level data transfer chore. We use DMA for bulk data transfer. For example, in an interrupt-driven I/O, the task assigned could be a DMA request to transfer data from a disk drive. Typically, a DMA controller oversees the data transfer. When the specified transfer is complete, the processor is notified by an interrupt signal.

We look at the interrupt mechanism and interrupt-driven I/O in the next chapter. In the remainder of this section, we describe the other two techniques.

19.4.1 Programmed I/O

The printer example given on page 769 explains how a typical programmed I/O operates. The heart of programmed I/O is a busy-wait loop. The processor busy-waits until a specific condition is satisfied, such as the printer "not busy" condition. This process is called *polling*.

We use the keyboard to illustrate how the programmed I/O works. We have already presented most of the details we need to write the keyboard program. Program 19.1 shows the program to read keys from the keyboard. Pressing the ESC key terminates the program.

The logic of the program is simple. To read a key, all we have to do is to wait for the PA7 bit to go low to indicate that a key is depressed (lines 34 to 36). Once we know that a key is down, we read the key scan code from PA6 to PA0 (line 38). The `and` statement on line 38 masks the most significant bit. Next we have to translate the scan code into the corresponding ASCII value. This translation is done by the `xlat` instruction on line 41. The `xlat` instruction uses

the `lcase_table` translation table given on lines 17 to 23.

After the key's ASCII value is displayed (line 47), we wait until the key is released. This loop is implemented by instructions on lines 50 to 53. Once the key is up, we clear the keyboard buffer using an interrupt service (lines 56 to 57). For now, ignore these two lines of code. We discuss these interrupt services in the next chapter.

Program 19.1 Programmed I/O example to read input from the keyboard

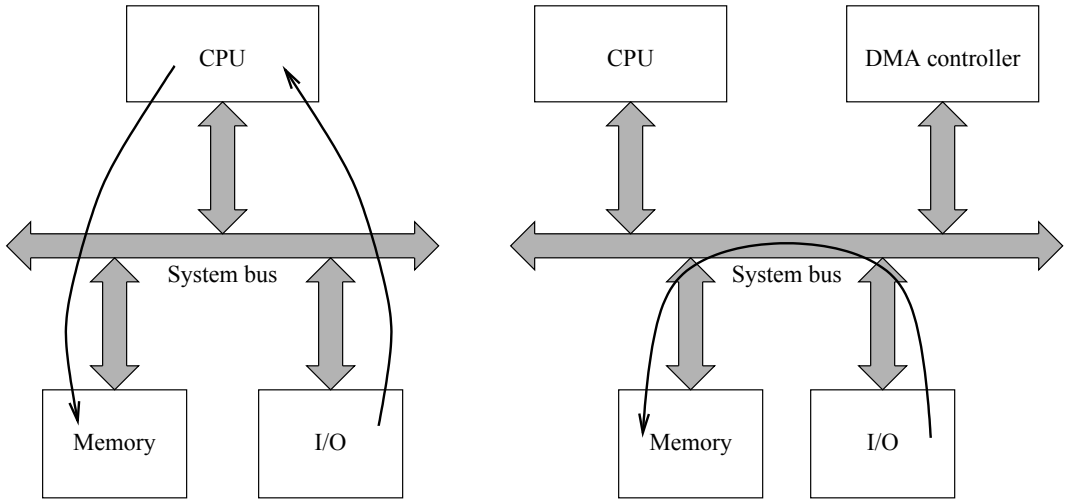
```
 1:  TITLE      Keyboard programmed I/O program      KBRD_PIO.ASM
 2:  COMMENT |
 3:          Objective: To demonstrate programmed I/O using keyboard.
 4:              Input: Key strokes from the keyboard.
 5:                     ESC key terminates the program.
 6:  |           Output: Displays the key on the screen.
 7:
 8:  ESC_KEY      EQU  1BH   ; ASCII code for ESC key
 9:  KB_DATA      EQU  60H   ; 8255 port PA
10:
11:  .MODEL SMALL
12:  .STACK 100H
13:  .DATA
14:  prompt_msg     DB  'Press a key. ESC key terminates the program.',0
15:  ; lowercase scan code to ASCII conversion table.
16:  ; ASCII code 0 is used for scan codes in which we are not interested.
17:  lcase_table    DB  01BH,'1234567890-=',08H,09H
18:                 DB  'qwertyuiop[]',0DH,0
19:                 DB  'asdfghjkl;',27H,60H,0,'\'
20:                 DB  'zxcvbnm,./',0,'*',0,' ',0
21:                 DB  0,0,0,0,0,0,0,0,0,0
22:                 DB  0,0,0,0,0,0,0,0,0,0
23:                 DB  0,0,0,0,0,0,0,0,0,0
24:  .CODE
25:  INCLUDE io.mac
26:
27:  main    PROC
28:          .STARTUP
29:          PutStr  prompt_msg
30:          nwln
31:  key_up_loop:
32:          ; Loops until a key is pressed i.e., until PA7 = 0.
33:          ; PA7 = 1 if a key is up.
34:          in      AL,KB_DATA   ; read keyboard status & scan code
35:          test    AL,80H       ; PA7 = 0?
36:          jnz     key_up_loop  ; if not, loop back
```

```
37:
38:          and     AL,7FH       ; isolate the scan code
39:          mov     BX,OFFSET lcase_table
40:          dec     AL           ; index is one less than scan code
41:          xlat
42:          cmp     AL,0         ; ASCII code of 0 => uninterested key
43:          je      key_down_loop
44:          cmp     AL,ESC_KEY   ; ESC key---terminate program
45:          je      done
46:  display_ch:
47:          putch   AL
48:          putch   ' '
49:
50:  key_down_loop:
51:          in      AL,KB_DATA    ; read keyboard status & scan code
52:          test    AL,80H        ; PA7 = 1?
53:          jz      key_down_loop ; if not, loop back
54:
55:          ; clear keyboard buffer
56:          mov     AX,0C00H
57:          int     21H
58:
59:          jmp     key_up_loop
60:  Done:
61:          ; clear keyboard buffer
62:          mov     AX,0C00H
63:          int     21H
64:
65:          .EXIT
66:  main    ENDP
67:          END   main
```

19.4.2 DMA

As we have seen, programmed I/O can be used to transfer data between I/O devices and memory. Programmed I/O overhead is small to interface with slower devices like the keyboard. However, for some I/O devices such as disks, data have to be transferred at a certain rate. For example, new drives with the Ultra ATA/100 data transfer protocol support a peak data transfer rate of 100 MB/s. Furthermore, no data during the transfer can be missed. For such devices, the CPU could be spending 10% to 20% of the time transferring data under the programmed I/O. Direct memory access is devised to free the processor of the data transfer responsibility.

DMA is implemented by using a DMA controller. The DMA controller acts as a slave to the processor and receives data transfer instructions from the processor. For example, to read a



(a) Programmed I/O transfer (b) DMA transfer

Figure 19.3 Data transfer from an I/O device to system memory: (a) in programmed I/O, data are read by the processor and then written to the memory; (b) in DMA transfer, the DMA controller generates the control signals to transfer data directly between the I/O device and memory.

block of data from an I/O device, the CPU sends the I/O device number, main memory buffer address, number of bytes to transfer, and the direction of transfer (I/O to memory or memory to I/O).

After the DMA controller has received the transfer instruction, it requests the bus. Once the DMA controller becomes the bus master, it generates all bus control signals to facilitate the data transfer. Figure 19.3 shows the difference between programmed I/O and DMA transfer. In programmed I/O, the system bus is used twice as shown in Figure 19.3a. The DMA transfer not only relieves the processor from the data transfer chore but also makes the transfer process more efficient by transferring data directly from the I/O device to memory.

DMA controllers typically support more than just one I/O device. For example, the 8237 DMA controller from Intel can support four independent devices. Each device is attached to a specific DMA channel. These channels are also called *I/O channels*. Each I/O channel has registers to keep track of the data buffer address in memory, a byte count to indicate the number of bytes to transfer, and the direction of the transfer.

The following steps are involved in a typical DMA operation:

1. The processor initiates the DMA controller by identifying the I/O device and supplying the memory address, byte count, and type of operation (input or output). This is referred to as *channel initialization*. Once initialized, the channel is ready for data transfer between the associated I/O device and memory.

2. When the I/O device is ready to transfer data, it informs the DMA controller. The DMA controller starts the transfer operation. This step consists of the following substeps.
   (a) Obtain the bus by going through the bus arbitration process described in Section 5.5;
   (b) Place the memory address and generate the appropriate read and write control signals;
   (c) Complete the transfer and release the bus for use by the processor or other DMA devices;
   (d) Update the memory address and count value;
   (e) If more bytes are to be transferred, repeat the loop from Step (a).
3. After completing the operation, the processor is notified. This notification is done by an interrupt mechanism (discussed in the next chapter). The processor can then check the status of the transfer (i.e., successful or failure).

The procedure in Step 2 is reasonable for slow devices. However, for fast devices that require bulk data transfer, we can improve the efficiency by avoiding a bus request for each word transfer. Instead, the DMA controller keeps the bus until the whole block of data is transferred. Next we illustrate how this can be done by means of an example.

An Example DMA Transfer

To illustrate the basic DMA operation, let's look at a sample data transfer. Figure 19.4 shows the interconnection among the four players: the CPU, memory, DMA controller, and the I/O device. In this figure, for simplicity, we show the address bus connecting the DMA controller to the memory and the data bus connecting the memory and I/O controller. However, note that these two buses, which are part of the standard system bus, also connect the processor and other components of the system.

Each DMA channel is initialized to support a specific I/O device. The processor does this initialization by writing appropriate commands into the DMA controller. After that the I/O channel is dedicated to service that specific I/O device. We illustrate the DMA operation with the aid of the timing diagram given in Figure 19.5. Let us assume that the DMA operation is a read operation (i.e., transferring data from the I/O device to memory) and the channel is initialized to transfer two words of data.

When the I/O device is ready to transfer data, it sends its request through the DMA request (DREQ) line to the DMA controller. On receiving this signal, the DMA controller raises the HOLD signal to indicate that it wants the bus for a DMA transfer. The processor releases the bus after completing its current instruction. The CPU then floats all the control signals it normally generates (such as memory read, memory write, I/O read, I/O write). The granting of the bus to the DMA controller is conveyed by asserting the hold acknowledge (HLDA) signal. This signifies that the DMA controller is in control of the bus. Then the DMA controller notifies the I/O device that the data transfer can begin by sending a DMA acknowledgment (DACK) signal to the I/O controller. For standard transfers, the DMA request can be removed after the DACK signal.

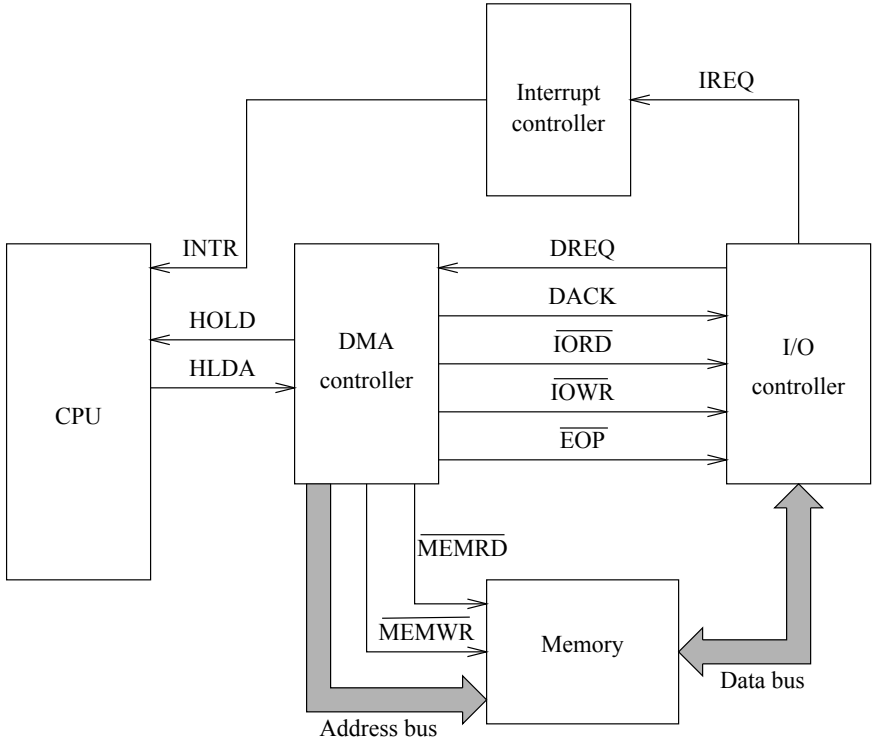


Figure 19.4 A simplified diagram giving the DMA controller details.

The DMA controller is responsible for generating the control signals I/O read ($\overline{\text{IORD}}$) and memory write ($\overline{\text{MEMWR}}$). The I/O device responds to the I/O read control signal and places the data on the data bus, which is written directly to the memory using the $\overline{\text{MEMWR}}$ signal. After the data word has been transferred, the word count is decremented. In addition, the memory address is incremented. If the word count is not zero, another data transfer cycle is initiated, as shown in Figure 19.5.

Once the two data transfers have been completed, the DMA controller indicates the termination of the operation by sending an "end of process" ($\overline{\text{EOP}}$) signal. It also removes the DACK signal, which in turn removes the HOLD signal, to return the bus to the processor. In this mode, the DMA transfer is done continuously without releasing the bus. For slow devices, this may unnecessarily block the bus from the processor. DMA transfer can also be done on a word-by-word basis. In this mode, the DMA controller requests the bus and releases it after transferring one word. The DMA controller decrements the count value and increments the buffer address, as before. When the device is ready to transfer the next word, it repeats the process by asserting the DREQ signal. This process is repeated until all the words are transferred, at which time the $\overline{\text{EOP}}$ signal is sent to the device to indicate that the data transfer has been completed and it should not send any more data.

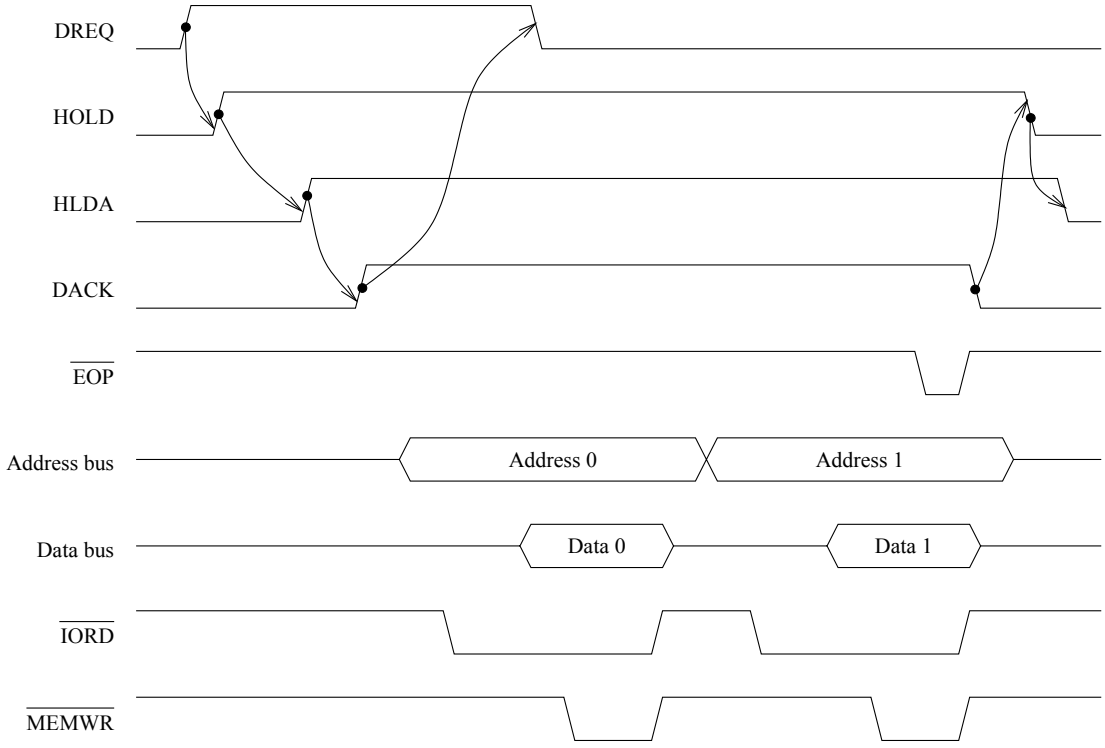


Figure 19.5 An example DMA transfer showing two transfer cycles.

An Example DMA Controller: Intel 8237

The Intel 8237 DMA controller chip has four DMA channels that can be programmed independently. Each channel uses a 16-bit address and a count value. Thus, it can address up to 64 K locations and can be initialized to transfer up to 64 K bytes. Usually, an external latch is used to specify the higher-order address bits in order to handle addresses greater than 16 bits.

The 8237 has the following internal registers:

- *Current Address Register:* Each channel has a 16-bit current address register. This register holds the address for the current DMA transfer. After each transfer, this address is automatically updated. The processor writes into this register during the initialization process. Since the 8237 has only an 8-bit data bus, the 16-bit address is written as two 8-bit writes.
- *Current Word Register:* This register maintains the word count. The actual number of transfers will be one more than the count value in this register. The word count is decremented after each transfer. When the register goes from zero to FFFFH, a terminal count (TC) signal is generated.

- *Command Register:* This 8-bit register controls the operation of the 8237. Using the command register, the processor can program the 8237 to assign fixed or rotating priorities for the DMA requests, whether the DREQ is high-active or low-active, and DACK should be high-active or low-active.
- *Mode Register:* Each channel has a 6-bit mode register that controls the DMA transfers on that channel. Some of the characteristics that can be programmed are the following:
  - Each channel can be programmed to either read or write a DMA transfer.
  - Each channel can be independently programmed to autoincrement or autodecrement the address after each data transfer.
  - Each channel can be programmed to autoinitialize the channel. For example, if the channel is initialized to transfer 1 KB data, the address and count registers are reinitialized after the DMA transfer is complete. This will eliminate the need for the CPU to initialize the channel after each DMA transfer.
- *Request Register:* This is a special feature of the 8237 to allow software-initiated DMA requests. A request register is associated with each channel. Normally, the DREQ signal initiates the DMA request. However, by setting the request bit in this register, we can start a DMA transfer (in addition to the standard DREQ-initiated transfers).
- *Mask Register:* Each channel has a mask register that can be set to disable the corresponding DMA request. All DREQ are disabled until this bit is cleared. When autoinitialization is not used, all further DREQs are disabled by automatically setting this bit at the end of the DMA operation when the $\overline{\text{EOP}}$ is generated.
- *Status Register:* This is an 8-bit register that provides the status of the four DMA channels to the CPU. Each of the four higher-order bits (bits 4 to 7) in this register indicate whether there is a DMA request pending on that channel. The lower-order four bits indicate whether the channel has reached the terminal count.
- *Temporary Register:* The 8237 supports memory-to-memory transfer of data. This register is used to hold the data during this kind of data transfer. To start this type of data transfer, the least significant bit in the command register should be set. This sets channels 0 and 1 to perform the block transfer of data in memory. The memory-to-memory transfer is initiated by setting the software DREQ for channel 0. The 8237 goes through the normal DMA operation. The data read from memory on channel 0 are placed in the temporary register. Then, channel 1 initiates a memory write to write the word from the temporary register. As usual, this cycle is repeated until the count reaches FFFFH, at which point, TC is set and the $\overline{\text{EOP}}$ is generated to terminate the transfer.

The A0 to A3 address lines are used to identify a register. Some address combinations are not used as the 8237 does not have 16 registers to use them all.

The 8237 supports four types of data transfer:

- *Single Cycle Transfer Mode:* In this mode, only a single data transfer takes place. As mentioned earlier, this mode is useful for slow devices.

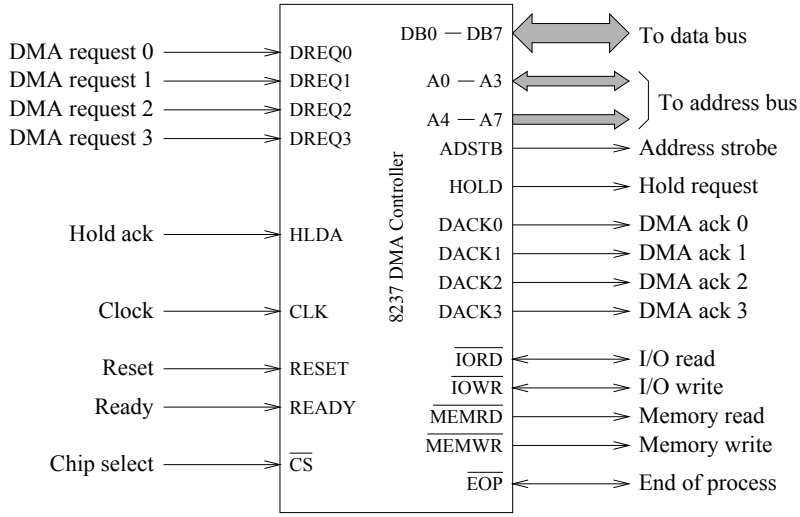


Figure 19.6 The Intel 8237 DMA controller.

- *Block Transfer Mode:* In this mode, the 8237 transfers data until the terminal count (TC) is generated or an external $\overline{\text{EOP}}$ is received.
- *Demand Transfer Mode:* This mode is somewhat similar to the block transfer mode. In this mode, the data transfer takes place until the TC, or an external $\overline{\text{EOP}}$ is received, or the DREQ signal goes inactive. This mode is useful for giving dynamic control to the I/O device to transfer the number of bytes it has.
- *Cascade Mode:* This mode is useful for expanding the number of DMA channels beyond the four provided by a single 8237 chip by adding more 8237 chips. A channel that is connected to another 8237 can use this mode so that it does not generate the control and address signals.

The logical signals of the 8237, which is a 40-pin chip, are shown in Figure 19.6. We have already discussed most of the signals before. The four DMA request lines (DREQ0 to DREQ3) and the corresponding acknowledgments (DACK0 to DACK3) serve the four DMA channels. The hold and hold acknowledgment signals are used to request a bus from the CPU; these two signals are also used to expand the number of DMA channels. The $\overline{\text{CS}}$, I/O read and write, along with the A0 to A3 address lines are used by the CPU to communicate with the 8237. The READY signal plays the same role as that in the CPU. This line can be used by a slow memory or I/O device to extend the memory read and write cycles. The $\overline{\text{EOP}}$ is a bidirectional signal. In a normal termination of a DMA transfer, the 8237 generates this signal to indicate the end of the data transfer. However, the 8237 also allows an external device to terminate the data transfer by pulling this line low.

When the 8237 is in the idle state, it monitors the DREQ lines every clock cycle to see if any channel is requesting service. It will also sample the $\overline{\text{CS}}$ line to see if the CPU is trying to communicate. If the $\overline{\text{CS}}$ is low, the 8237 allows the processor to program it and read its internal registers. In the program mode, the 8237 acts as an I/O device as far as the CPU is concerned. The CPU uses the I/O read ($\overline{\text{IOR}}$) and write ($\overline{\text{IOW}}$) signals along with the $\overline{\text{CS}}$ to communicate with the 8237. In addition, the lower four address lines (A0 to A3) are used to identify an 8237 I/O port (i.e., one of its internal registers). Thus, in the program mode, the $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ signals as well as A0 to A3 are input signals to the 8237. These signals are generated by the 8237 during the DMA transfer mode (i.e., these lines will be output signals).

If a DREQ line is active, the 8237 initiates a DMA transfer that essentially follows the actions described before (see Figure 19.5). In the DMA transfer mode, DMA acts as the bus master and generates the necessary I/O and memory control signals ($\overline{\text{IOR}}$, $\overline{\text{IOW}}$, $\overline{\text{MEMR}}$, and $\overline{\text{MEMW}}$). This is the reason why the I/O read and write signals are shown as bidirectional signals in Figure 19.6.

The original IBM PC used only a single 8237 chip to provide four DMA channels. Channel 0 was used for dynamic memory refresh, channel 2 for the floppy drive controller, and channel 3 for the hard disk controller. Channel 1 was available for another I/O device. Later models increased the number of channels by adding more 8237 chips.

19.5 Error Detection and Correction

Error detection is important in computer and communications systems. In computer systems, the wire distances are small compared to communication networks. Furthermore, the external disturbances are not as severe as in communication networks, such as wide area networks. In a computer system, we can divide the communication into two groups: within the box and with entities outside the box. Typically, the environment is well controlled within a box. In addition, the distances are small. For example, a memory read operation causes data to move on the motherboard. Even then, it is possible to receive data in error. The probability of an error increases when communicating to a device outside the system with a long wire.

Therefore, we should at least be able to detect errors in the received data stream. This process is called error detection. In addition, we would also like to be able to correct the error. We discuss some popular error detection and correction schemes in this section.

19.5.1 Parity Encoding

Parity is the simplest mechanism that adds rudimentary error detection capability. The basic principle is simple: add a bit, called the parity bit, such that the total number of 1s in the data and the parity bit is either even (for even parity) or odd (for odd parity). For example, if even parity is used with 7-bit ASCII data `1001100`, we append an extra parity bit of 1 to make the number of 1s even (four in this example). Thus, we transmit the codeword `11001100`, where the leftmost bit is the parity bit.

Suppose that, during the transmission, a bit is flipped. The receiver can detect this error because the number of 1s is odd. Of course, parity encoding will not detect all errors. For

example, if two bits flip, the receiver will not be able to detect the error. However, this simple parity scheme can detect all single-bit errors. The overhead is small; only a single extra bit is needed. In Section 19.5.3, we discuss a more comprehensive error detection scheme.

19.5.2 Error Correction

Let us focus on single-bit errors. Parity encoding will only tell us that there is an error. However, we don't know which bit is in error. If we know this information, we simply flip the bit to correct the error. How do we get this bit number? To incorporate error correction capability, we have to add more parity bits. We need to understand some simple theory to come up with an error-correcting scheme.

To correct single-bit errors in d data bits, we add p parity check bits to form a codeword c with $d + p$ bits. Next we show how we can compute these p check bits in order to facilitate single-bit error correction.

Hamming distance between two codewords is the number of bit positions in which the two codewords differ. For example, the Hamming distance between `11001100` and `10101000` is 3. The Hamming distance for the entire code is the smallest Hamming distance between any pair of codewords in the encoding scheme.

When we use even or odd parity encoding, we create a code with a Hamming distance of two. What this implies is that every other codeword in the coding sequence is illegal. Let us consider two data bits and a parity bit. Therefore, the codeword will have three bits for a total of eight codewords. Out of these, only four are legal codewords and the remaining are illegal, as shown below.

```
000
001   ⟸ Illegal codeword
011
010   ⟸ Illegal codeword
110
111   ⟸ Illegal codeword
101
100   ⟸ Illegal codeword
```

The above sequence is written such that the Hamming distance between successive codewords is one (including the first and the last). This ordering is also known as the *Gray code*. This is very similar to what we did in Karnaugh maps in Chapter 2. Thus, if a single-bit error occurs, a legal codeword is transformed into an illegal codeword. For example, when a single-bit error occurs, a `101` codeword becomes one of the following, depending on the error bit: `100`, `111`, or `001`. As you can see from the previous list, all these are illegal codewords.

With this parity scheme, we get detection capability, but we do not know how to correct the error. For example, if the received codeword is `100`, the legal codeword could be any of the following: `101`, `110`, or `000`.

Suppose that we have a code with a Hamming distance of 3. Then, we can see that a single-bit error leaves the error codeword closer to the original codeword. For example, we know that the codewords `11001100` and `10101000` have a Hamming distance of 3. Assume that a single-bit error occurs in transmitting `11001100` and is received as `11101100`. The received codeword is closer to `11001100` (Hamming distance of 1) than to `10101000` (Hamming distance is 2). Thus, we can say that the error is in the third bit position from left. This is essentially the principle used to devise error-correcting codes. We now describe a method that can correct single-bit errors.

The codeword consists of d data bits and p check bits. If we count the bit positions from left to right starting with 1, check bits occupy the bit positions that are a power of 2 (i.e., bit positions 1, 2, 4, 8, 16, . . .). Let us consider an example of encoding the 8-bit data `10101010`. The codeword with the check bits is shown below:

P1	P2	D7	P4	D6	D5	D4	P8	D3	D2	D1	D0
		1		0	1	0		1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12

The codeword is 12 bits long. The check bit values are derived as in the parity scheme discussed before. However, each check bit looks at a specific set of bit positions to compute its even parity value.

Check bit P_1 looks at bit positions 1, 3, 5, 7, 9, and 11.
Check bit P_2 looks at bit positions 2, 3, 6, 7, 10, and 11.
Check bit P_4 looks at bit positions 4, 5, 6, 7, and 12.
Check bit P_8 looks at bit positions 8, 9, 10, 11, and 12.

For our example, P_1 should be 1 because bit positions 3, 5, 7, 9, and 11 have an odd number of 1s (three 1s). Similarly, P_2 should be 1 as well because the number of 1s in bit positions 3, 6, 7, 10, and 11 is odd (again three). You can check that $P_4 = 1$ and $P_4 = 0$. This gives us the following codeword:

P1	P2	D7	P4	D6	D5	D4	P8	D3	D2	D1	D0
1	1	1	1	0	1	0	0	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12

What is the logic in selecting these specific bit positions for each parity bit? The idea is to have a unique set of check bits test the correctness of each bit. For example, bit 11 is checked by P_1, P_2, and P_8. Suppose this bit is in error. Then, only P_1, P_2, and P_8 are in error but not P_4. This particular pattern is not used for any other bit. This leads us to the following simple rule to identify the bit in error. The error bit position is the sum of the error parity bits with P_1 counted as 1, P_2 counted as 2, P_4 counted as 4, and so on. For our example, if bit 11 is in error, P_1, P_2, and P_8 check bits are in error. By adding the weights of these three check bits, we get

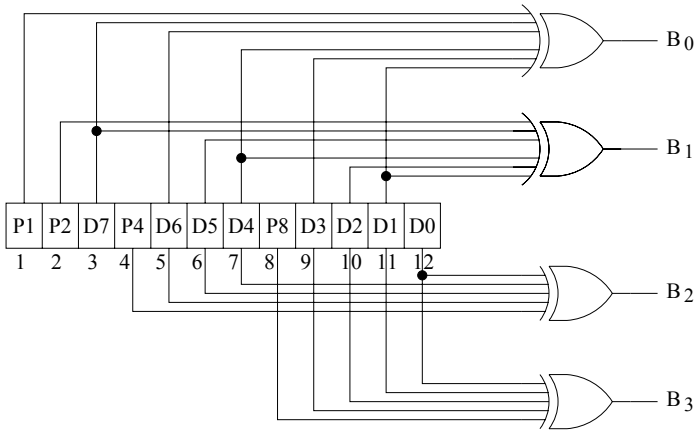


Figure 19.7 A simple circuit to identify the error bit position.

the error bit position as 11. We correct the bit by inverting the wrong bit. If there is no error, all check bits will be correct, which points to a bit position 0. Since we do not use the zero bit position, this combination is used to indicate error-free condition.

Figure 19.7 shows a simple digital logic circuit to detect errors. If there is an error, the 4-bit binary number $B = \mathrm{B_3B_2B_1B_0}$ will have a nonzero value. Furthermore, the binary number B would indicate the error bit position. This information can be used to correct the error in hardware.

Single-Error Correction and Double-Error Detection (SECDED)

High-performance systems use SECDED encoding on data transfers from memory. The previous encoding scheme gives us only single-error detection and correction capability. To get double-error detection capability, we have to add an additional parity bit that covers the entire codeword: data and parity. By convention, this parity bit is added as the leftmost bit P_0 that is not used by the error correction scheme. This gives us the capability to detect two errors. For our previous example, we use eight data bits and *five* check bits to get the SECDED capability.

19.5.3 Cyclic Redundancy Check

As we show in the next section, data transmission can be done either in parallel or serially. In parallel transmission, n bits are transmitted on n wires in parallel. On the other hand, serial transmission is done on a single wire, one bit at a time. The error correction scheme we have discussed in the last section is generally applied to parallel data transmission such as data transmitted from memory to processor/cache.

In parallel transmission, it is reasonable to assume that errors on each line are independent. If we assume that the error rate is 10^{-8}, then the possibility of two errors occurring simultane-

ously is 10^{-16}. Thus, in this case, the assumption that the single-bit error is the dominant type of error is reasonable. However, when we consider serial transmission of data, the assumption of independent errors in bits is invalid. At high bandwidths, even a transient noise can corrupt several bits in sequence. This type of error is called the *burst error.* Thus, in serial transmission, we have to handle burst errors. Furthermore, we often send a block of data rather than individual words. In these cases, the amount of overhead makes error correction very expensive. Therefore, only error detection schemes are used. Parity codes are not useful for detecting burst errors. Almost always, cyclic redundancy check codes are used to detect burst transmission errors.

Unlike the error-correcting codes, CRC codes use a fixed number of bits, typically 16 or 32 bits. Thus, the overhead is very small. Next we look at the basic theory that explains how and why the CRC code works.

The basic CRC calculation is based on integer division. When we divide a number (dividend) D by a divisor G, we get a quotient Q and a remainder R. That is,

$$\frac{D}{G} = Q + R.$$

For example, if we divide 13 by 5, we get $Q = 2$ and $R = 3$.

If we subtract the remainder R from D and then divide that number by G, we always get a zero remainder. In the previous example, we get 10 after subtracting the remainder 3 from 13. Thus, if we send this number, the receiver can divide it by G to check for errors. If the remainder is zero, there is no error. This is the basic principle used in CRC checking.

What Is the Divisor for the CRC Calculation? CRC uses a polynomial of degree n as the divisor. For example, one of the polynomials used by the universal serial bus is $x^{16} + x^{15} + x^2 + 1$. This polynomial identifies the 1 bit positions in the divisor. Thus, the USB polynomial represents the binary number `1100000000000101`. This polynomial is used for data packets. The USB also uses another shorter polynomial for token packets. It is of degree 5: $x^5 + x^2 + 1$. This polynomial represents the binary number 100101. Such polynomials are called *polynomial generators*. It should be noted that the number of bits in the binary number is one more than the degree of the polynomial. If we divide a binary number by a polynomial generator G of degree n, we get an n-bit remainder.

Subtraction Operation: CRC computation uses modulo-2 arithmetic, which means there will be no carry or borrow. This simplifies the subtract operation, which can be implemented using bit-wise exclusive-OR (XOR) operation.

CRC Generation

The CRC generator takes d-bit data D and a degree n polynomial G. It computes the CRC code and appends the n-bit remainder of the division to the message. This is analogous to subtracting the remainder in our example.

A little bit of theory would explain the CRC procedure. Let

$C =$ $(d + n)$-bit codeword,
$D =$ d-bit data,
$R =$ n-bit remainder (i.e., CRC code),
$G =$ degree n polynomial generator.

The goal is to generate C such that C/G would have no remainder. Remember that this is the condition that the receiver uses for error checking. Because we append n bits to the right, the codeword can be expressed as

$$C = D \times 2^n \oplus R. \tag{19.1}$$

We use $\oplus$ to represent the XOR operation. We generate R as

$$\frac{D \times 2^n}{G} = Q \oplus \frac{R}{G}, \tag{19.2}$$

where Q is the quotient and R is the remainder. Our interest is in R. We add this remainder R to generate the codeword C as in Equation 19.1.

When this codeword is received, it is divided by G,

$$\frac{C}{G} = \frac{D \times 2^n \oplus R}{G}.$$

By substituting Equation 19.2, we get

$$\frac{C}{G} = Q \oplus \frac{R}{G} \oplus \frac{R}{G} = Q \oplus \frac{R \oplus R}{G}.$$

Since $\oplus$ is the exclusive-OR operation, we know that $R \oplus R = 0$. Therefore,

$$\frac{C}{G} = Q$$

with zero remainder. This is used to check for error-free condition.

Example 19.1 *CRC computation using polynomial* $x^5 + x^2 + 1$.
Inputs: d-bit data `10100101` ($d = 8$ in this example) and a degree n polynomial generator $G = x^5 + x^2 + 1$ (i.e., 100101).

Output: n-bit CRC check bits.

Procedure: The d-bit message is appended n zeros on the right. In our example, the numerator is `10100101 00000`. This is equivalent to multiplying the message by 2^n. As mentioned, exclusive-OR is used for the subtract operation. The procedure is shown in Figure 19.8.

The message incorporating the CRC check bits is `10100101 01110`. The receiver performs exactly the same procedure to check for errors. An error in the received message results in a nonzero remainder. As shown in Figure 19.9, the remainder is zero for error-free messages. □

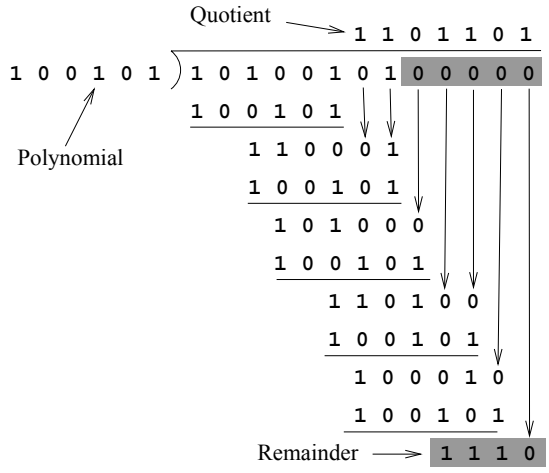


Figure 19.8 An example CRC calculation for `10100101`.

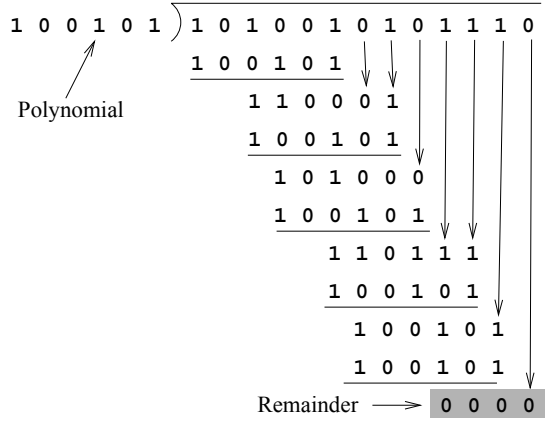


Figure 19.9 An error-free message results in a zero remainder.

A Serial CRC Generator Circuit

Generation of CRC can be done in a straightforward way by using shift registers and exclusive-OR gates. Figure 19.10 shows the logic circuit to generate the CRC code for the polynomial generator function $x^{16} + x^{15} + x^2 + 1$. We use a square to represent a 1-bit shift register. As you can see, the circuit uses a 16-bit shift register because the divisor is a degree 16 polynomial. The output from `X15` is passed through an XOR gate for the bit positions that have a 1 bit in the polynomial. In our example, bit positions 0, 2, and 15 receive the input through an XOR gate. Note that the shift register is cleared (i.e., initialized to zero) before each computation.

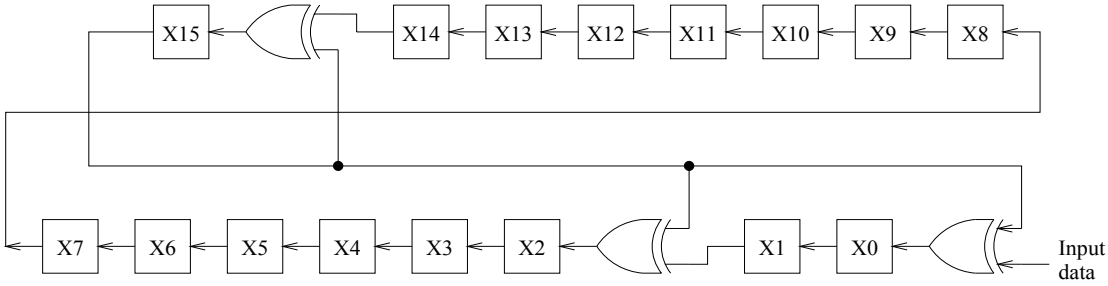


Figure 19.10 A serial CRC generator circuit for the polynomial generator function $x^{16} + x^{15} + x^2 + 1$.

The input data to the circuit consist of the actual data plus 16 zeros appended on the right (similar to what we did in Figure 19.8 to calculate the CRC). The data are fed to the circuit from left to right (i.e., the appended zero bits are fed last). Once the last bit is processed, the shift register would have the remainder.

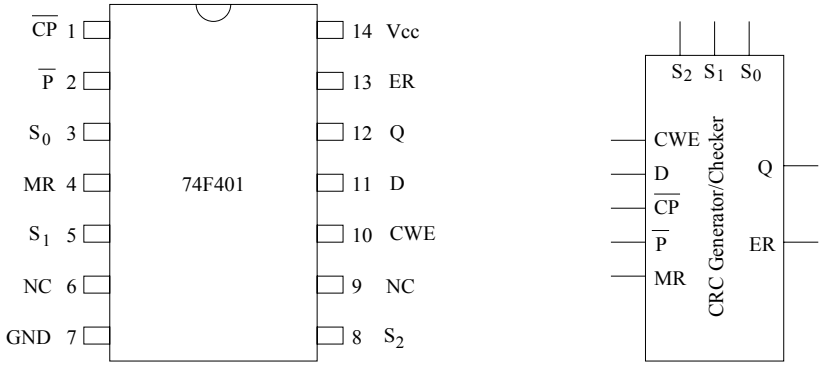
An Example CRC Generator and Checker Chip

The 74F401 chip supports CRC generation and checking of several standard 16-, 12-, and 8-degree polynomials. Figure 19.11 shows the connection diagram and the logic symbol. The three selection inputs S_2, S_1, S_0 are used to select the polynomial generator that should be used in CRC generation and checking. For example, setting these three bits to zero uses the polynomial $x^{16} + x^{15} + x^2 + 1$. It also supports seven more polynomials. It is a serial CRC generator and takes the data input on pin D with the clock on $\overline{\text{CP}}$. The input data are clocked on the high-to-low transition of the clock signal. The output Q provides the CRC code that can be appended to the data stream. The chip has an internal 16-bit shift register, which is implemented using D flip-flops. The reset (MR) and preset ($\overline{\text{P}}$) are used to clear and set the register, respectively. If the chip is used for error detection, the ER output indicates the error condition (ER = high).

The check word enable (CWE) input determines whether the X0 input to the internal CRC generator should be zero or the output of the XOR gate as shown in Figure 19.12a. The CWE input is used to control the data flow to the output as shown in Figure 19.12b. During the data phase, CWE = 1. This passes the input data to the CRC generator as well as to the output. At the end of the input data, CWE is made zero. This forces the data input of 74F401 to be zero (equivalent to the n zero bits we added on the right in our CRC calculation example). This also enables the CRC code from the Q output of the chip to pass on to the output.

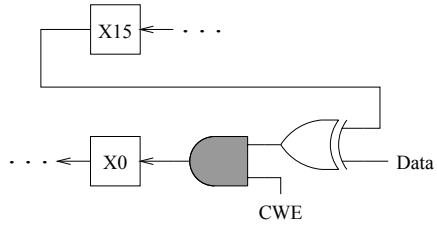
19.6 External Interface

We can interface I/O devices in one of two basic ways as shown in Figure 19.13: using a parallel or serial interface. In parallel transmission, several bits are transmitted on parallel wires as shown in Figure 19.14. For example, the parallel port we used to read from the keyboard

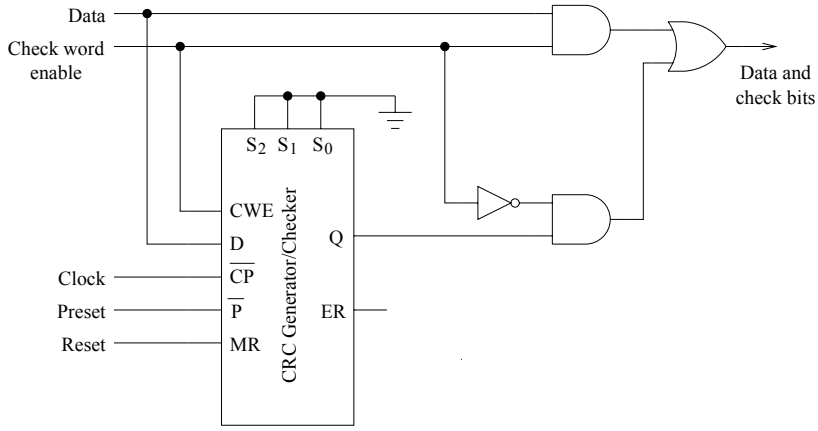


(a) Connection diagram (b) Logic symbol

Figure 19.11 Details of the 74F401 CRC generator and checker chip.



(a) Gated Code Word Enable input



(b) Check word generation

Figure 19.12 Using the 74F401 chip to generate the CRC for $x^{16} + x^{15} + x^2 + 1$.

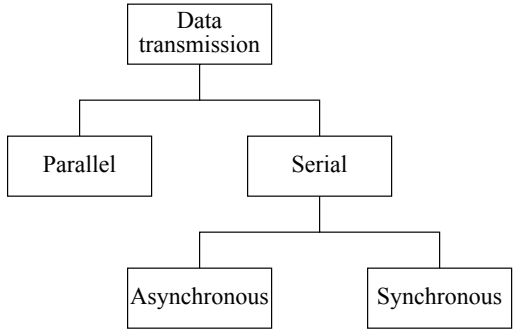


Figure 19.13 A taxonomy of data transmission techniques.

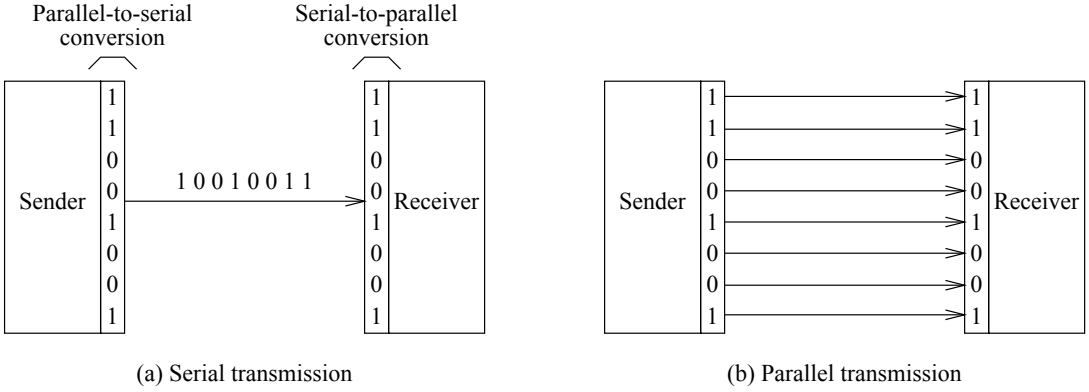


Figure 19.14 Two basic modes of data transmission.

transmits eight bits at a time (see Section 19.3.2 on page 772). On the other hand, in the serial transmission, only a single wire is used for data transmission.

Parallel transmission is faster: we can send n bits at a time on an n-bit wide parallel interface. That also means it is expensive compared to the serial interface. A further problem with the parallel interface, particularly at high data transfer rates, is that skew (some bits arrive early and out of sync with the rest) on the parallel data lines may introduce errorprone delivery of data. Because of these reasons, the parallel interface is usually limited to small distances. In contrast, the serial interface is cheaper and does not cause the data skew problems.

In this section, we look at some simple external bus standards to interface I/O devices. We look at the serial and parallel transmission techniques and standards. Sections 19.7 and 19.8 discuss two high-speed serial buses: the Universal Serial Bus and FireWire.

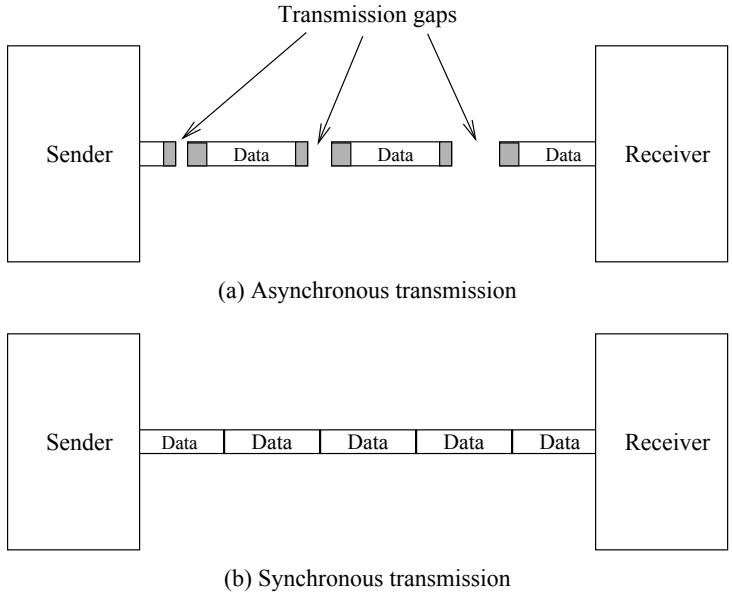


Figure 19.15 Asynchronous and synchronous transmission modes.

19.6.1 Serial Transmission

Serial transmission can be *asynchronous* or *synchronous*. In asynchronous mode, each byte is encoded for transmission in such a way that the sender and receiver clocks are not required to be in synchronization (see Figure 19.15*a*). In synchronous transmission, the two clocks should be synchronized before the data are transmitted. This synchronization is done in hardware using phase-locked-loops (PLLs). Once the clocks are synchronized, we can send a block of data, rather than just a byte as in the asynchronous transmission shown in Figure 19.15*b*.

In asynchronous transmission, each byte of data is enveloped by a start bit and one or more stop bits, as shown in Figure 19.16. The communication line is high when the line is idle. When transmitting a character, the start bit pulls the line low. This alerts the receiver that a byte is coming. It also identifies the bit boundary. Since the receiver knows the bit period, it samples the transmission line in the middle of the bit cell. It ignores the first bit, which is the start bit, and assembles a byte. The stop bit serves two purposes:

- Imagine what happens if we don't have a stop bit. Suppose the most significant bit of the byte is 0. Then, unless we force some idle time on the line, the start bit of the next byte will not cause the transition to identify the start of a new byte. The stop bit forces the line to go high between byte transmissions.
- Stop bits also give breathing time for the receiver to assemble the byte and hand it over to the receiver system before monitoring the line for the start bit. Typically, systems can use 1, $1\frac{1}{2}$, or 2 stop bits.

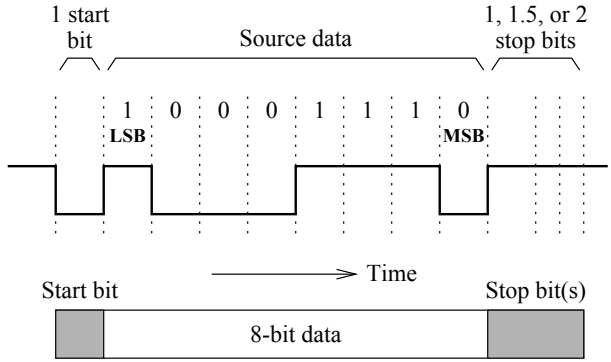


Figure 19.16 Asynchronous transmission uses start and stop bits.

Low-speed serial transmission is suitable for I/O devices like the keyboard. Since the characters are typed at a slow speed with large variable gaps, synchronous transmission is not useful.

In general, synchronous transmission is more efficient but it is expensive, as we need more hardware support for clock synchronization. However, it does not have the overhead associated with asynchronous transmission. In asynchronous transmission, for each eight bits, there are at least two additional bits of overhead, that is, 25% overhead, excluding the gaps between bytes. For this reason, high-speed serial transmission buses such as the USB and FireWire use the synchronous mode.

EIA-232 Serial Interface

As an example of a low-speed serial transmission standard, we describe the EIA-232 standard adopted by the Electronic Industry Association (EIA). This standard is also widely known by the name of its predecessor, RS-232. This is the serial interface port on your PC. The original standard uses a 25-pin DB-25 connector. However, a simplified version of the standard can use the familiar 9-pin connector DB-9. This interface is typically used to connect a modem to a computer system.

EIA-232 uses eight signals to facilitate serial transmission of data (see Figure 19.17). The original standard uses terminology like DCE (data circuit-termination equipment) and DTE (data terminal equipment). DCE is the unit that interfaces with the actual transmission network (e.g., modem). DTE is the unit that generates the binary data (e.g., computer). For our discussion purposes, we use the words modem and computer rather than DCE and DTE.

We use an example to illustrate the protocol. Assume that computer A wants to send data to computer B. Computer A generates the serial binary data and hands it over to its modem A. Modem A modulates these data so that they can go on the telephone line. The reverse process takes place at the receiving end. Modem B receives the modulated serial data and demodulates them (i.e., converts the signal back into the binary form) and passes them on to computer B.

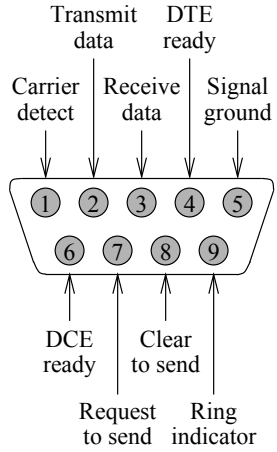


Figure 19.17 DB9 connector and its signals.

The transmission occurs in three phases just as in the way we proceed to make a phone call. When we want to make a telephone call, we dial the number, establish the connection, then talk to the person at the other end, and disconnect. The transmission protocol also uses these three phases: connection setup, data transmission, and connection termination.

- *Connection Setup:* This phase consists of the following substeps:

  1. Computer A asserts the DTE Ready (pin 4) signal to indicate to its modem that it wants to initiate a connection for data transfer. Then, it transmits the phone number of the receiver to the modem via the Transmit Data line (pin 2).

  2. When modem A contacts modem B, modem B alerts its computer of the incoming call via the Ring Indicator (pin 9) line. Computer B responds by asserting its DTE Ready line (pin 4) to indicate that it is ready to receive data. After receiving the green signal from computer B, modem B generates a carrier signal for data exchange and turns the DCE Ready signal to indicate that it ready to receive data from computer B.

  3. When modem A detects the carrier signal from the receiving modem, modem A alerts its computer via the Carrier Detect (pin 1) signal. It also turns the DCE Ready (pin 6) line to indicate to computer A that a circuit has been established. Modem A completes the circuit establishment by generating its own carrier signal for data transmission from A to B.

- *Data Transmission:* The data transmission phase uses handshaking by using the request-to-send (RTS) and clear-to-send (CTS) signals. When computer A is ready to send data, it activates the RTS (pin 7) signal. Modem A conveys its readiness by asserting CTS (pin

8). Computer A transmits the data via the Transmit Data (pin 2) line. Similar action takes place at the other end.

- *Connection Termination:* Once both sides have completed their data transmission session, both computers deactivate the RTS signals. This in turn causes the modems to turn off the carrier signals and their CTS signals. This terminates the connection.

19.6.2 Parallel Interface

As an example of a simple parallel interface, we look at the parallel printer interface with which you are familiar. It uses a 25-pin connector (known as DB-25). Although we present details about how a printer is interfaced using this parallel port, it can also be used by input devices like scanners. The pin assignment is shown in Table 19.3, which is derived from the interface developed by Centronics for their printers.

The interface is straightforward. It uses eight lines for the data, which are latched by using the strobe signal (pin 1). Data transfer uses simple handshaking by using the acknowledge (ACK) signal. After each byte, the computer waits to receive an acknowledgment before sending another byte of data. There are five lines to provide printer status and feedback: busy, out of paper, online/offline, autofeed, and fault. The printer can be initialized by using the INIT signal. This clears the printer buffer and resets the printer.

Small Computer System Interface (SCSI)

Another parallel interface, often used for high-speed transfer of data, is the Small Computer System Interface (SCSI, pronounced "scuzzy"). SCSI comes in two bus widths: 8 and 16 bits. The 8-bit wide SCSI is often called "narrow" SCSI and the 16-bit as "wide" SCSI.

SCSI is based on the disk interface developed in 1979 by Shugart Associates called SASI (Shugart Associates System Interface). This interface was adopted by the American National Standards Institute (ANSI) who released the first standard, SCSI 1, in 1986. It used an 8-bit wide bus and supported a transfer rate of 5 MB/s (see Table 19.4). SCSI 1 described the physical and electrical characteristics, but did not provide a standard command set. The industry developed a set of 18 basic commands. This set was referred to as the common command set (CCS).

The next version, SCSI 2 or Fast SCSI, used the CCS as the basis and increased the speed to 10 MB/s. The CCS has been expanded to include commands for other devices. It has also added command queuing so that commands can be executed efficiently.

Table 19.4 shows the roadmap for scaling SCSI to provide higher bandwidths. Today's Ultra SCSI provides about 20 MB/s. We can get double that if we use the Wide Ultra SCSI, which uses a 16-bit wide bus.

The data transfer rate is doubled from wide Ultra 2 to Ultra 160 by sending two bits of data per clock instead of one without increasing the clock frequency. That means that both the rising and falling edges of the REQ and ACK signals, which run at 40 MHz, are used to clock data.

SCSI is not a point-to-point interface as is the parallel interface. It is a true bus, which means we can add more than one device to it. In general, the 8-bit SCSI can have up to 8 devices and the 16-bit SCSI up to 16 devices. Each SCSI device is assigned a unique number to identify it

Table 19.3 Parallel printer interface signals

Pin #	Signal	Signal direction	Signal function
1	STROBE	PC $\Longrightarrow$ printer	Clock used to latch data
2	Data 0	PC $\Longrightarrow$ printer	Data bit 0 (LSB)
3	Data 1	PC $\Longrightarrow$ printer	Data bit 1
4	Data 2	PC $\Longrightarrow$ printer	Data bit 2
5	Data 3	PC $\Longrightarrow$ printer	Data bit 3
6	Data 4	PC $\Longrightarrow$ printer	Data bit 4
7	Data 5	PC $\Longrightarrow$ printer	Data bit 5
8	Data 6	PC $\Longrightarrow$ printer	Data bit 6
9	Data 7	PC $\Longrightarrow$ printer	Data bit 7 (MSB)
10	ACK	printer $\Longrightarrow$ PC	Printer acknowledges receipt of data
11	BUSY	printer $\Longrightarrow$ PC	Printer is busy
12	POUT	printer $\Longrightarrow$ PC	Printer is out of paper
13	SEL	printer $\Longrightarrow$ PC	Printer is online
14	AUTO FEED	printer $\Longrightarrow$ PC	Autofeed is on
15	FAULT	printer $\Longrightarrow$ PC	Printer fault
16	INIT	PC $\Longrightarrow$ printer	Clears printer buffer and resets printer
17	SLCT IN	PC $\Longrightarrow$ printer	TTL high level
18–25	Ground	N/A	Ground reference

on the bus and to direct the traffic. For narrow SCSI, the device id ranges from 0 to 7; for wide SCSI, 0 to 15 are used for device id.

SCSI supports both internal and external device connection. Narrow SCSI uses 50-pin connectors for both the internal and external interfaces. Wide SCSI uses 68-pin connectors to allow for the additional eight data lines. To allow other devices to be connected to the bus, each external SCSI device has an input and an output connector. This allows several devices to be chained by connecting the output of one device to the input of the next one.

SCSI allows a maximum cable length of 25 meters; but as the number of devices increases from 1, the length is reduced. Typically, if more than two devices are connected, the maximum cable length is reduced to about 12 meters.

Table 19.4 Types of SCSI

SCSI type	Bus width (bits)	Transfer rate MB/s
SCSI 1	8	5
Fast SCSI	8	10
Ultra SCSI	8	20
Ultra 2 SCSI	8	40
Wide Ultra SCSI	16	40
Wide Ultra 2 SCSI	16	80
Ultra 3 (Ultra 160) SCSI	16	160
Ultra 4 (Ultra 320) SCSI	16	320

Since more than one device can be on the bus, bus arbitration is needed to allocate the bus. SCSI uses a simple priority-based bus arbitration mechanism. If more then one device wants to be the bus master, the highest priority device gets the bus.

Details about the narrow SCSI signals are given in Table 19.5. To achieve better isolation, twisted pairs are used for the signal line. This is the reason for keeping one side of the connector (pins 1 through 25) more or less for the ground. For example, pin 26 and pin 1 are used for the twisted pair for data bit 0. From these signals, we can see some similarity with the parallel port interface. SCSI uses a single parity bit to provide error detection for each byte transferred. In addition, CRC is used to protect the integrity of the data.

SCSI uses a client-server model. It uses *initiator* and *target* instead of client and server. The initiator device issues commands to targets to perform a task. The target receives commands from initiators and performs the task requested. Typically, initiators are SCSI host adapters in computers and target devices are typically SCSI devices such as disk drives.

A SCSI host adapter sends a command to a selected target device on the SCSI bus by asserting a number of control signals. The target device acknowledges the selection and begins to receive data from the initiator. SCSI transfer proceeds in phases, which include the following operations: command, message in, message out, data in, data out, and status. The direction of transfer "In" and "Out" is from the initiator point of view. The target device is responsible for taking the bus between phases by correctly asserting the SCSI bus control signals.

SCSI uses asynchronous mode for all bus negotiations. It uses handshaking using the REQ and ACK signals for each byte of data. On a synchronous SCSI bus, the data are transferred synchronously. In the synchronous transfer mode, the REQ–ACK handshake is not used for each byte of data. A number of data bytes (e.g., eight) can be sent without waiting for the ACK by using REQ pulses. This increases the throughput and minimizes the adverse impact of the cable propagation delay. For more details on the SCSI standard, see the Web pointer in Section 19.11 on page 823.

Table 19.5 Narrow SCSI signals

Description	Signal	Pin	Pin	Signal	Description
Twisted pair ground	GND	1	26	D0	Data 0
Twisted pair ground	GND	2	27	D1	Data 1
Twisted pair ground	GND	3	28	D2	Data 2
Twisted pair ground	GND	4	29	D3	Data 3
Twisted pair ground	GND	5	30	D4	Data 4
Twisted pair ground	GND	6	31	D5	Data 5
Twisted pair ground	GND	7	32	D6	Data 6
Twisted pair ground	GND	8	33	D7	Data 7
Twisted pair ground	GND	9	34	DP	Data parity bit
Ground	GND	10	35	GND	Ground
Ground	GND	11	36	GND	Ground
Reserved		12	37		Reserved
No connection		13	38	TermPwr	Termination power (+5 V)
Reserved		14	39		Reserved
Ground	GND	15	40	GND	Ground
Twisted pair ground	GND	16	41	ATN	Attention
Ground	GND	17	42	GND	Ground
Twisted pair ground	GND	18	43	BSY	Busy
Twisted pair ground	GND	19	44	ACK	Acknowledge
Twisted pair ground	GND	20	45	RST	Reset
Twisted pair ground	GND	21	46	MSG	Message
Twisted pair ground	GND	22	47	SEL	Selection
Twisted pair ground	GND	23	48	C/D	Command/data
Twisted pair ground	GND	24	49	REQ	Request
Twisted pair ground	GND	25	50	I/O	Input/output

19.7 Universal Serial Bus

The Universal Serial Bus was originally developed in 1995 by a consortium of companies including Compaq, Hewlett Packard, Intel, Lucent, Microsoft, NEC, and Philips. The major goal of the USB was to define an external expansion bus that makes attaching peripherals to a computer as easy as hooking up a telephone to a walljack.

The current standard specification (USB 1.1) supports low-speed as well as full-speed devices. The USB can operate at 1.5 Mbps (low speed) and 12 Mbps (full speed). The low speed is sufficient for devices like the mouse and the keyboard. The 12 Mbps bandwidth supports LANs and other peripherals like the disk drives. The next version (USB 2.0) increases the bandwidth by a factor of 40 to 480 Mbps. This higher bandwidth makes the USB competitive with the SCSI as well as FireWire, as we show in the next section. With the increased bandwidth, the USB can support higher performance peripherals such as video-conferencing cameras, fast storage devices, and next-generation scanners, printers, and CD writers.

The USB 2.0 uses the same USB 1.1 connectors and full-speed cables without any changes. The transmission speed is negotiated on a device-by-device basis. If the higher speed is not supported by a peripheral, the link operates at a lower speed of 12 Mbps or 1.5 Mbps as determined by the peripheral. The USB version 2.0 is widely available in 2002.

19.7.1 Motivation

Before the USB, computer users faced several problems when attaching peripherals. Here we list some of these problems to show how the USB solves them.

Device-Specific Interfaces: PCs tended to have various device-specific interfaces. Some example connectors we will find on a PC include the PS/2, serial, parallel, monitor, microphone, speakers, modem, SCSI, and Ethernet. In most cases, each connection uses its own connector and cable, leading to cable clutter. In contrast to this scenario, USB uses a single connector type to connect any device.

Nonshareable Interfaces: Standard interfaces support only one device. For example, we can connect only one printer to the parallel interface. In contrast, the USB supports up to 127 devices per USB connection. For example, we can connect a keyboard, a mouse, and speakers to a single USB port using a single cable type.

I/O Address Space and Interrupt Request Problems: Adding a new peripheral device often causes I/O address and interrupt request (IRQ) conflicts. One may end up spending countless hours in debugging the conflict problem. In contrast, the USB does not require memory or address space. There is also no need for interrupt request lines. We show later how the USB handles traditional interrupts.

Installation and Configuration: Using the standard interfaces, adding a new peripheral device is often a time-consuming and frustrating experience for novice users. It may often involve

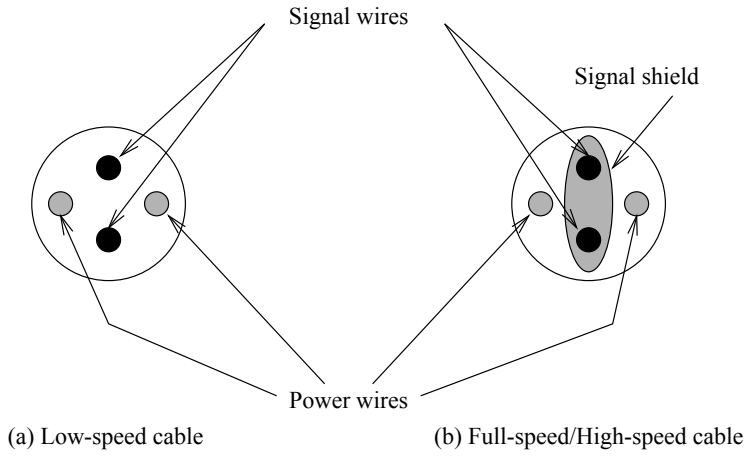


Figure 19.18 USB cables.

opening the box and installing expansion cards and configuring jumpers or DIP switches. In contrast, the USB supports true plug-and-play connectivity. It avoids unpleasant tasks such as setting jumpers and configuring the new device.

No Hot Attachment: We are too familiar with the dreaded sequence of restarts whenever we attach a new device. Attaching USB devices is easy: we don't have to turn off the computer and restart after installing the new device. We can hot plug the device and the system will automatically detect the device and configure it for immediate use.

19.7.2 Additional USB Advantages

In addition to these advantages, the USB also offers the following useful features:

- *Power Distribution:* The USB cable provides +5 V power to the device. Depending on the port, it can supply between 100 and 500 mA of current. This helps avoid the clutter of external power supplies. Small devices can operate by taking power right from the USB cable. Devices such as keyboards, mouse, wireless LANs, and even some floppy disk drives can be powered from the cable.

  The USB cable has four wires: two of these are used for power supply (+5 V) and the remaining two to carry the differential signal pair. The low-speed cables, also called subchannel cables, cannot exceed three meters. Furthermore, they do not require signal shielding as shown in Figure 19.18. The high- and full-speed cables shield the signal pair and can run up to five meters.
- *Control Peripherals:* The USB allows data to flow in both directions. This means the computer can control the peripheral device.

- *Expandable Through Hubs:* The USB provides increased expandability through hubs that are cheap and widely available. For example, we can buy a four-port hub for about $25 and seven-port hubs for about $50.
- *Power Conservation:* USB devices enter a suspend state if there is no activity on the bus for 3 ms. In the suspended state, devices draw only about 2.5 mA of current.
- *Error Detection and Recovery:* The USB uses CRC error-checking to detect transmission errors. In case of an error, the transaction is retried.

19.7.3 USB Encoding

The USB uses the NRZI (nonreturn to zero-inverted) encoding scheme. NRZI is often used in communication networks for encoding data. Figure 19.19 explains this encoding. Before describing the NRZI scheme, let's look at the simple NRZ encoding. In this encoding, a 0 is represented by a low level and a 1 by a high level as shown in Figure 19.19. Even though this scheme is simple to implement, it has two serious problems:

- Signal transitions do not occur if we are transmitting long strings of zeros or ones. Signal transitions are important for the receiver to recover data.
- In a noisy medium, it is difficult to detect zero- and one-levels. It is far easier to detect a transition, either from 0 to 1 or 1 to 0.

NRZI solves some of these problems. It will not use absolute levels to encode data; instead, it uses the presence or absence of signal transition to determine the bit value. In standard NRZI encoding, a signal transition occurs if the next bit is 1; the signal level stays the same if the next bit is 0. Note that a signal transition can be from low to high or vice versa. The USB NRZI scheme uses the complementary rule: a signal transition occurs if the next bit is zero as shown in Figure 19.19. In the rest of the discussion, we use the USB rule. Such encoding schemes are called *differential encoding* schemes.

NRZI encoding solves the two main problems associated with NRZ encoding. In NRZI encoding, signal level does not play any role. It only looks for signal transitions. Thus, it improves reliability of data transmission. Furthermore, it also solves the long strings of zeros. A long string of zeros forces the NRZI signal to alternate.

We still have a problem with long strings of ones. In this case, the signal level stays the same. To solve this problem, USB encoding uses bit stuffing. A zero is inserted after every six consecutive ones in the data before the data are NRZI encoded. This bit stuffing forces a transition in the NRZI data stream, as shown in Figure 19.20. This gives the receiver logic a signal transition at least once every seven bit times to guarantee data and clock recovery.

19.7.4 Transfer Types

The USB supports the following four transfer types: interrupt, isochronous, control, and bulk.

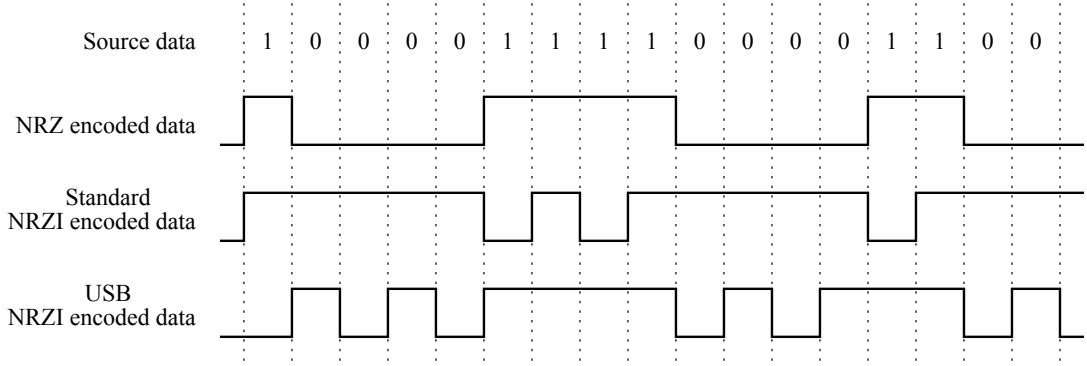


Figure 19.19 The USB encoding scheme.

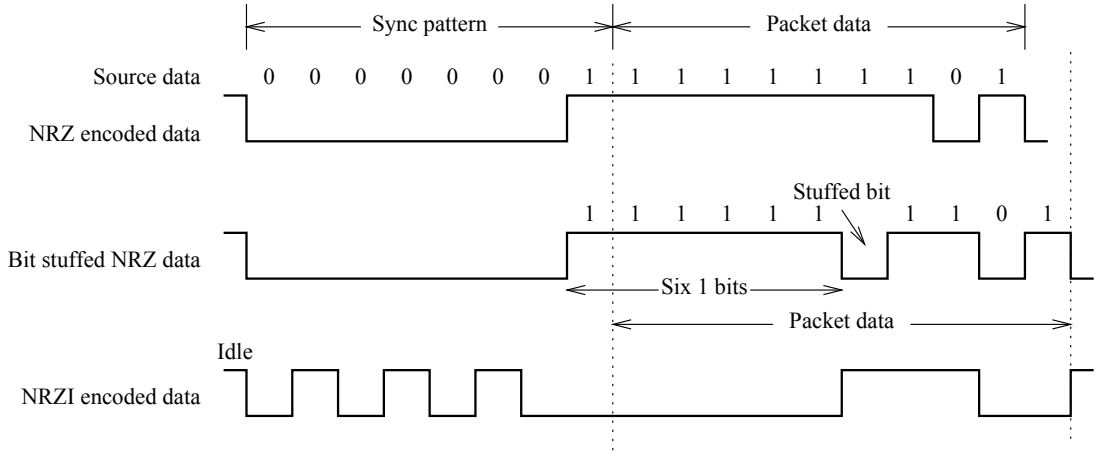


Figure 19.20 The USB uses bit stuffing to avoid the problems associated with long strings of ones.

Interrupt Transfer

The USB does not support interrupts in the traditional sense. For example, the keyboard generates an interrupt whenever you press or release a key. (We discuss keyboard interrupts in detail in the next chapter.) Instead, the USB uses polling, which is similar to the busy-waiting used by the programmed I/O. To get acceptable performance, we have to select an appropriate polling frequency. If the frequency is too high, we will waste the bandwidth. On the other hand, if we use too low a frequency, we may lose data. The frequency of USB interrupt transfers can be adjusted to meet the device requirements. In the USB 1.1, the polling interval can range from 1 to 255 ms, that is, from 1000 times to about 4 times a second. The USB uses an end-

point descriptor to specify the polling interval, in increments of 1 ms. Since the data integrity is important in this type of transfer, error detection and recovery are supported. If an error is detected, a retry is attempted.

Isochronous Transfer

Real-time applications that require a constant data transfer rate use this type of transfer. These applications need data in a timely manner. Examples include transmission of data to speakers and reading audio from a CD-ROM. To provide a constant data transfer rate, isochronous transfers are scheduled regularly. These transfers are real-time in nature, where timely delivery of data is more important. Isochronous transfers do not use error detection and recovery.

Control Transfer

Control transfers are used to configure and set up the USB devices. A control transfer involves two to three steps:

- *Setup Stage:* Control transfers always begin with a setup stage to convey the type of request made to the target device (e.g., reading the contents of the device descriptor).
- *Data Stage:* This is optional and only control transfers that require data transfers use this stage. For example, when the request is to read the device descriptor, the device will use this stage to send the descriptor.
- *Status Stage:* This final stage is always used to check the status of the operation.

Control transfers are provided a guaranteed 10% bus bandwidth allocation. However, if more bandwidth is available, it can be allocated to control transfers. Since data integrity is important, error detection and recovery are supported.

Bulk Transfer

Devices that do not have any specific transfer rate requirements use the bulk transfers. For example, in transferring a file to a printer, the data can be transferred at a slower rate without any problem. Bus bandwidth is allocated to bulk transfers on a low priority basis. In a fully allocated frame, it is possible that the other three types of transfers take 100% of the bandwidth. In that case, bulk transfers are deferred until the load on the USB system decreases. Bulk transfers support error detection and recovery. Recovery, as usual, is done by retransmission.

19.7.5 USB Architecture

The host hardware consists of a USB host controller and a root hub. The host controller initiates transactions over the USB, and the root hub provides the connection points. There are two types of host controllers: the open host controller (OHC) and the universal host controller (UHC).

Intel defined the UHC whereas Compaq, Microsoft, and National Semiconductor specified the OHC. These two controllers perform the same basic function. The UHC/OHC along with

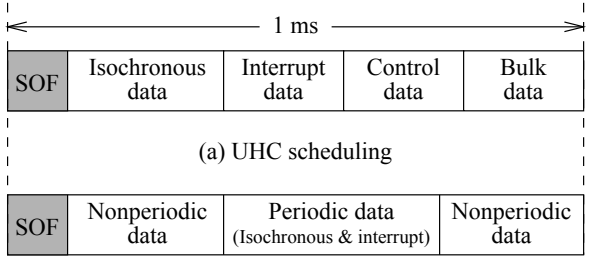


Figure 19.21 The UHC and OHC scheduling of the four transfer types (SOF = start of frame).

their drivers are responsible for scheduling and executing I/O request packets (IRPs) forwarded by the USB driver.

The main difference between the two controllers is the policy used to schedule the four types of transfers (interrupt, isochronous, control, and bulk transfers). Both controllers, however, use 1 ms frames to schedule the transfers.

The UHC schedules periodic transfers—isochronous and interrupt—first. These transfers are followed by the control and bulk transfers as shown in Figure 19.21a. The periodic transfers can take up to 90% of the bandwidth and the control transfers are allocated a guaranteed 10% bandwidth. As mentioned before, bulk transfers are scheduled only if there is bandwidth available after scheduling the other three transfers.

The OHC uses a slightly different scheduling policy. It reserves space for nonperiodic transfers (control and bulk) at the beginning of the frame such that these transfers are guaranteed 10% of the bandwidth (see Figure 19.21b). Next periodic transfers are scheduled to guarantee 90% of the bandwidth. If there is time left in the frame, nonperiodic transfers are scheduled as shown in Figure 19.21b.

The root hub receives the transactions generated by the host controller and transmits them on the USB. The root hub is responsible for the distribution of power, enabling and disabling the ports, device recognition, and status reporting when polled by the host software. The USB system can be expanded by using USB hubs, as shown in Figure 19.22.

Bus-powered devices can be divided into two groups: low- and high-power devices. Low-power devices should consume less than 100 mA of current. A high-powered device may use more than 100 mA but must use less than 500 mA. Similarly, a port can be a low-powered (supplies up to 100 mA) or full-powered (supplies up to 500 mA) port.

The bus can power a low-power device. Examples of such devices include the keyboard and mouse. There are some floppy disk drives that are bus powered! High-power devices can be designed to have their own power supply rather than drawing current from the USB bus. High-powered USB devices operate in three power modes: configured (500 mA), unconfigured (100 mA), and suspended (about 2.5 mA). When a device is connected, it is not configured and draws less than 100 mA. Thus, it can be connected to both low-powered and full-powered ports.

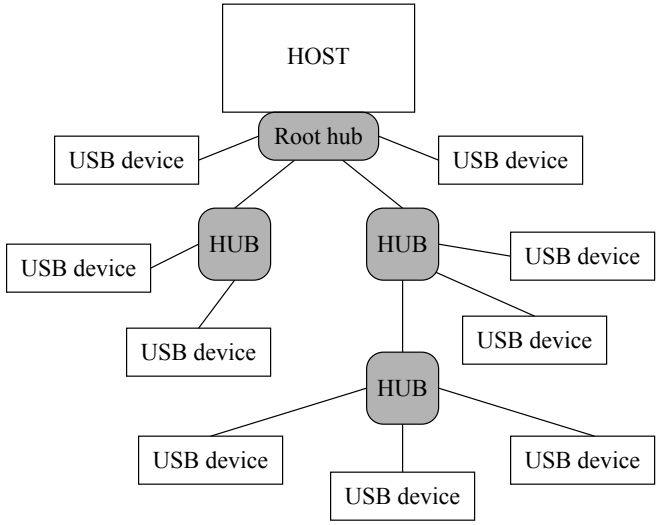


Figure 19.22 The USB can be expanded by using hubs: The hub port connected to the parent is called the upstream port and the port connected to a child is called the downstream port.

The port can then read the device's configuration register, which contains information about its current requirement, and configure the device if the port supports the required current. For example, if the device is a high-powered one requiring 500 mA, and the port is full-powered, the device is configured. Then it draws a full 500 mA. If, on the other hand, this device is connected to a low-powered port, the device is not configured and the user is informed of the situation.

USB hubs can be bus-powered or self-powered. A bus-powered hub does not require an extra power supply. The hub uses the power supplied by the USB. Bus-powered hubs can be connected to an upstream port that can supply a full 500 mA current. The downstream ports of the hub can only provide 100 mA of current. Furthermore, the number of ports is limited to four. Most four-port USB hubs work in dual power mode. In bus-powered mode, they support up to four low-powered devices. In self-powered mode, they support up to four high-powered USB devices or up to four downstream bus-powered USB hubs.

On the other hand, a self-powered hub uses its own power supply. Thus it is not restricted by the limitations of the bus-powered hub. Self-powered hubs can have more than four ports and supply a full 500 mA of current on each of their downstream USB ports. For example, we can get seven-port USB hubs, but these are not bus-powered.

19.7.6 USB Transactions

Transfers are done using one or more transactions. Each transaction consists of several packets. As shown in Figure 19.23, each application's data are encapsulated into several transactions.

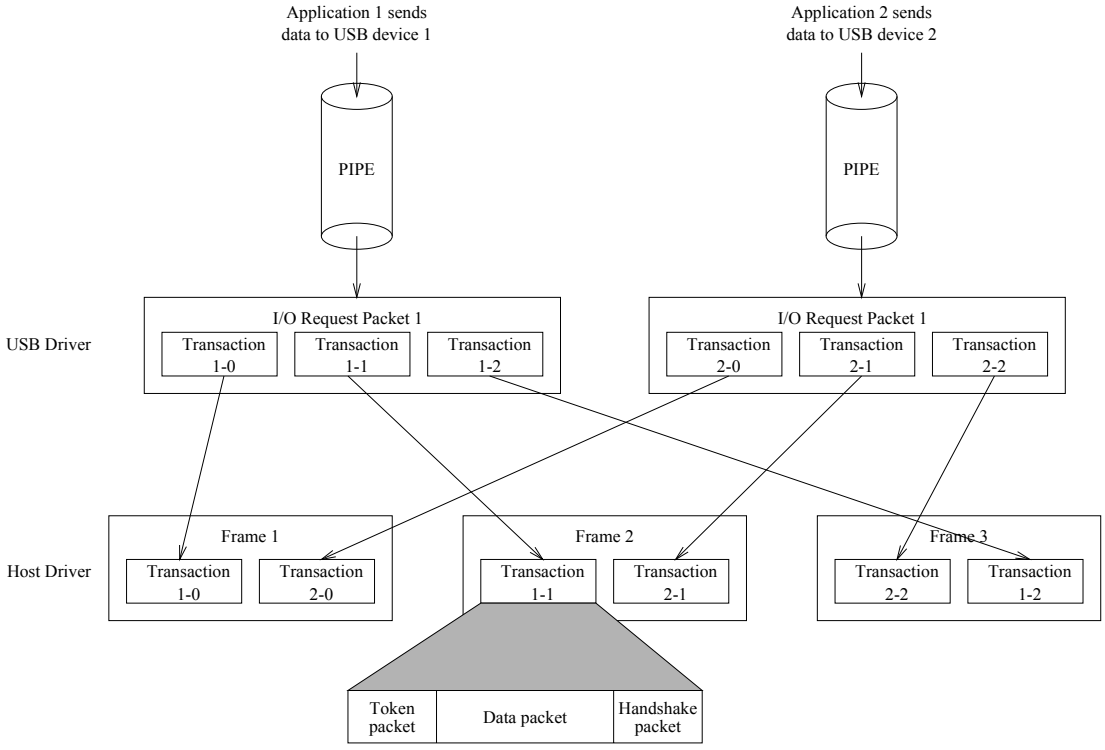


Figure 19.23 USB I/O request packet (IRP) frame.

The host driver multiplexes transactions from several devices based on the bandwidth allocation rules discussed before. At full speed, a frame is transmitted every 1 ms. Transactions may have between one and three phases:

- *Token Packet Phase:* All transactions begin with a token phase. It specifies the type of transaction and the target device address.
- *Data Packet Phase:* If the transaction type requires sending data, a data phase is included. A maximum of 1023 bytes of data are transferred during a single transaction.
- *Handshake Packet Phase:* Except for the isochronous data transfers, the other three types use error detection to provide guaranteed delivery. This phase provides feedback to the sender as to whether the data have been received without any errors. In case of errors, a retransmission of the transaction is done. Isochronous transfers do not use the handshake phase, as error detection is not done for these transfers.

The packet format is shown in Figure 19.24. A synchronization sequence precedes each packet. The receiver uses this sequence to synchronize to the incoming packet data rate. The

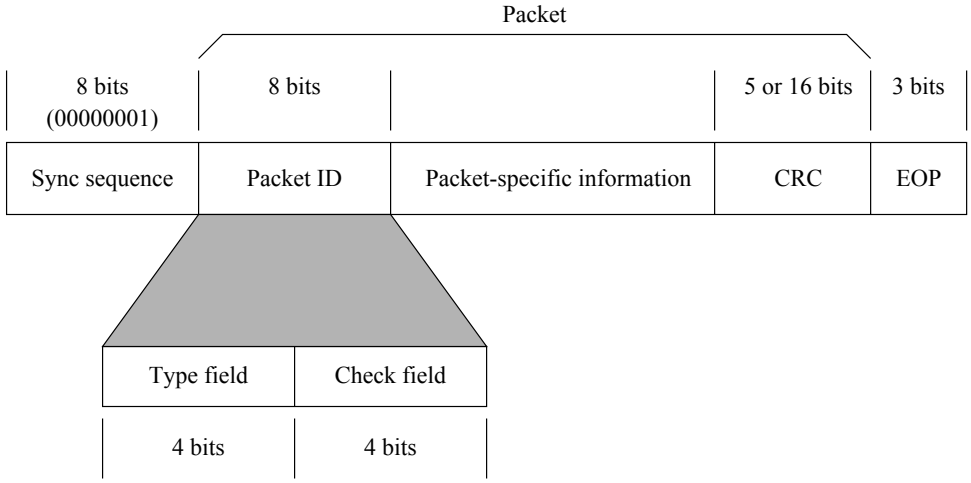


Figure 19.24 USB packet format.

synchronization sequence is an 8-bit value: 00000001 (see Figure 19.20). Each packet consists of a packet id, packet-specific data, and a CRC field.

The packet id consists of a 4-bit type field and a 4-bit check field. The type field identifies whether the packet is a token, data, or handshake packet. Since the type field is only 4 bits long, a 4-bit check field that is derived by complementing the type field is used to protect it. There is no need to go through the complicated CRC process to protect the type field.

A conventional CRC field protects the data part of a USB packet. The USB uses the CRC-5 and CRC-16. The generator polynomial for CRC-5 is $x^5 + x^2 + 1$. The generator polynomial for CRC-16 is the one we have seen before ($x^{16} + x^{15} + x^2 + 1$).

In token packets, a 5-bit CRC provides adequate protection, as this field is only 11 bits. In addition, using a 5-bit CRC aligns the packet to a byte boundary.

The end-of-packet (EOP) field indicates the end of each packet. This field is hardware encoded such that this encoding does not occur within the packet. The EOP is indicated by holding both signal lines low for two bit periods and leaving them idle for the third bit.

Let us now see how all these pieces fit together. Figure 19.25 shows how an IN transaction is composed. The IN transaction transfers data from a USB device to the system. Nonisochronous transactions consist of three packets, as shown in Figure 19.25a. Since isochronous transactions do not have the handshaking packet, it uses only two packets (Figure 19.25b).

The USB 2.0 specifies a microframe, which is 1/8th of the original USB's 1 ms frame (i.e., the USB 2.0 frame is 125 μs). This allows USB 2.0 devices to have small buffers even at the high data rates they support. For more details on the USB standard, see the Web pointer in Section 19.11 on page 823.

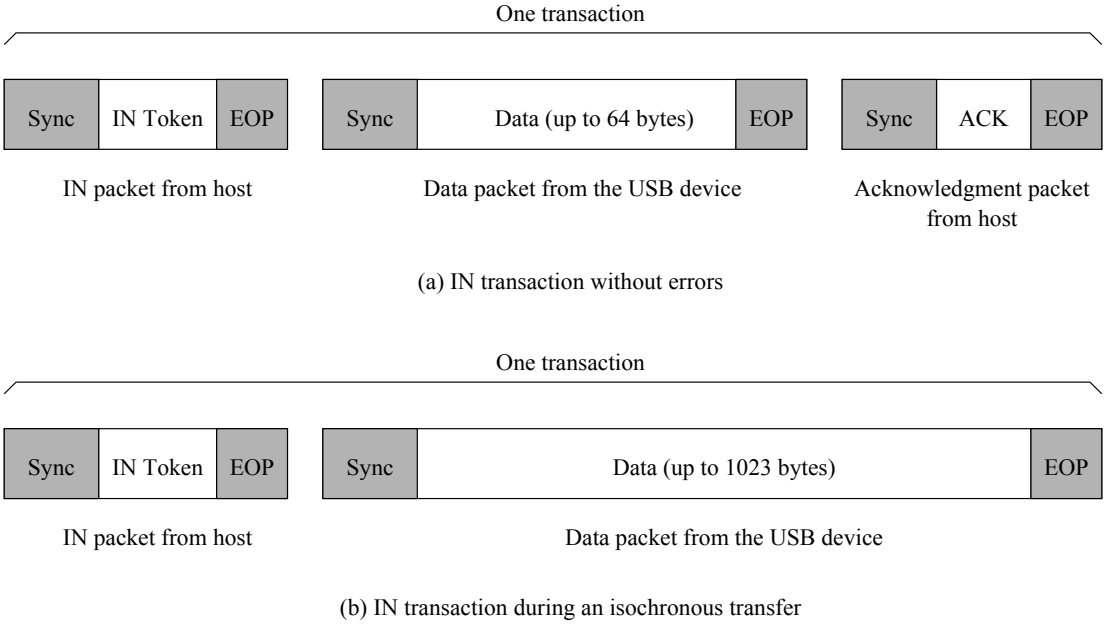


Figure 19.25 USB 1.1 transactions.

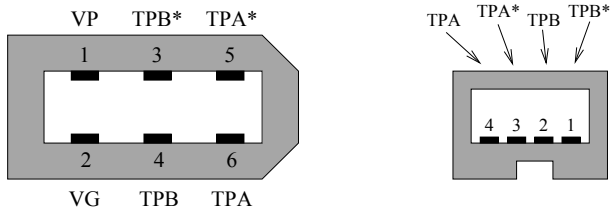
19.8 IEEE 1394

Apple originally developed this standard for high-speed peripherals. Apple called it *FireWire* because of its high-speed transmission (particularly when it was first developed in the 1980s). In fact, FireWire is a trademark of Apple. IEEE has standardized FireWire as IEEE 1394. Sony calls it *i.LINK*. The first version IEEE 1394-1995 was released in 1995. A slightly revised version was later released as 1394a. As of this writing, the next version 1394b is ready in draft form.

19.8.1 Advantages of IEEE 1394

IEEE 1394 shares many of the features we have seen with the USB. We sum up some of the more important ones here:

- *Speed:* IEEE 1394 offers substantially more bandwidth than the USB. When compared to the speed of 12 Mbps provided by USB 1.1, 1394 supports three speeds: 100, 200, and 400 Mbps. As we have noted before, USB 2.0 competes well in this range with its 480 Mbps rating. However, 1394 is also boosting the speeds to 3.2 Gbps! Since most implementations currently support USB 1.1 and 1394a, it is fair to say that 1394 provides substantial improvement in speed (more than 33 times). As a result, the USB interface is used for low- to medium-speed I/O devices, and the 1394 is used for high-speed devices. The 1394 is particularly entrenched in digital video transmission and editing applications.



(a) 6-pin connector (b) 4-pin connector

Figure 19.26 IEEE 1394 6-pin and 4-pin connectors.

- *Hot Attachment:* As with the USB, devices can be attached and removed dynamically without shutting down the system.
- *Peer-to-Peer Support:* The USB is processor-centric (i.e., the processor initiates the transfers). The USB typically supports host-to-peripheral applications. The IEEE 1394, on the other hand, supports peer-to-peer communication without involving the processor. For example, when a new device is attached, all devices on the bus reconfigure among themselves even if the bus is not connected to the system. The configuration process is described later.
- *Expandable Bus:* Devices can be connected in daisy-chain fashion to extend the bus. The bus can also be extended by using hubs as in the USB system.
- *Power Distribution:* As with the USB, the cable distributes power. However, 1394 supplies much higher power than the USB. The voltage can range between 8 and 33 V, and the current can be up to 1.5 amps. In contrast, USB power distribution is limited to 5 V and 0.5 amps. Therefore, 1394 cable can power more devices like the disk drives.
- *Error Detection and Recovery:* As with the USB, data transmission errors are detected using CRC. In case of errors, transaction retry is attempted.
- *Long Cables:* The 1394 supports cable lengths up to about four meters, which is in between the USB's cable lengths of three meters (low-speed) and five meters (full-speed).

19.8.2 Power Distribution

1394 uses two types of connectors and cables: 6- or 4-wire. The two connectors are not interchangeable as shown in Figure 19.26. The 4-wire cable does not provide power distribution. The 4-pin connector is compact and is used in most digital video devices such as camcorders, which have their own power.

Signal encoding in 1394 is different from that in the USB. The clock information is not embedded into the data signal. There are two twisted pairs to carry signals: one pair carries the data and the other the strobe signal. The strobe signal uses twisted pair A (TPA), and the data are transmitted over twisted pair B (TPB).

The data are encoded by using the simple NRZ scheme, as shown in Figure 19.27. The strobe signal is encoded such that it changes state if the successive bits are the same. That is,

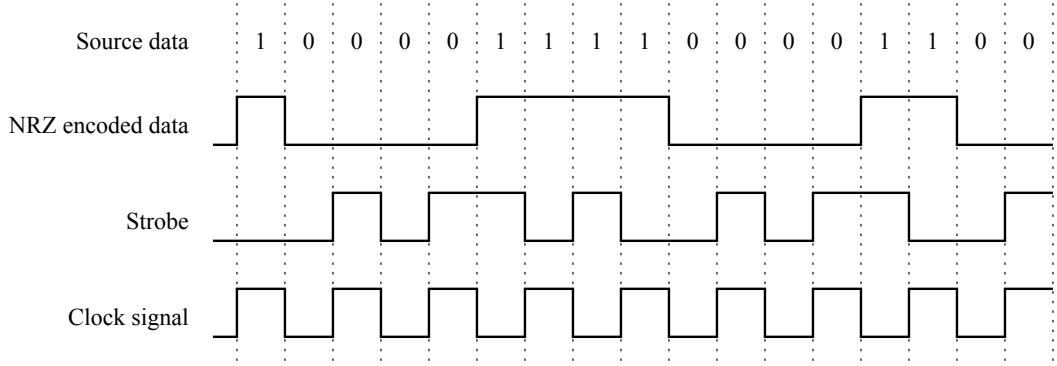


Figure 19.27 IEEE 1394 signal encoding: Data encoding uses the standard NRZ scheme. The strobe signal changes whenever the input data are the same in two successive cells.

the strobe signal captures the transitions that are missing from the data on TPB. There is no need to transmit the clock signal. These two signals together carry the clock information. The receiver can retrieve the clock by XORing the two signals (see Figure 19.27). The advantage of using this type of strobe signal as opposed to carrying the actual clock signal is that it requires far fewer transitions, which decreases the bandwidth requirement.

19.8.3 Transfer Types

Unlike the USB, 1394 supports only two types of transfers: *asynchronous* and *isochronous*. The asynchronous transfers are used for those applications that require correct delivery of data. Errors are not tolerated in these applications. For example, writing a file to a disk drive requires asynchronous transfer. As in the USB, the isochronous transfers are used for real-time applications such as audio and video. Here timely delivery is more important than correct delivery. Occasional errors do not affect the quality of the data. As a result of these differences, the two transfer types use different transfer protocols.

The asynchronous transfers use an acknowledgment to confirm delivery of the data, as shown in Figure 19.28*a*. The 1394 refers to the sender as the *requester* and the receiver as the *responder*. Asynchronous transfers can be of three types: read, write, and lock. The read and write actions allow data to be read or written. The lock transaction is used to perform the test-and-set type of action. It is up to the responder to implement the test-and-set type of atomic execution. Bus bandwidth is allocated on a per-cycle basis. Each cycle is 125 μs. All asynchronous transfers are collectively allocated a guaranteed bandwidth of 20%. A given node is not guaranteed any bandwidth for asynchronous transfers, but the bus arbitration uses a fair allocation scheme so that no node is starved of the bandwidth.

The isochronous transfers do not require an acknowledgment as shown in Figure 19.28*b*. Isochronous requesters are called isochronous talkers and responders are isochronous listeners. Isochronous transfers use a 6-bit channel number to identify the listener. In order to make

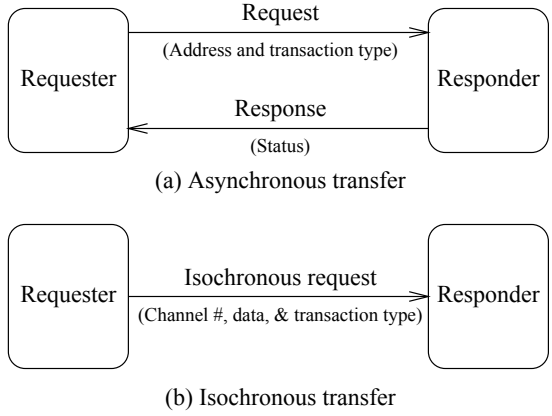


Figure 19.28 Asynchronous and isochronous communication models.

Table 19.6 Maximum data size for asynchronous and isochronous transfers

Bandwidth (Mbps)	Maximum data size (bytes)	
	Asynchronous transfers	Isochronous transfers
100	512	1024
200	1024	2048
400	2048	4096

sure that sufficient bandwidth is available for constant delivery of data for isochronous applications, each application should first request the needed bandwidth from the isochronous resource manager node.

During each cycle, up to 80% of the cycle time can be allocated to isochronous transfers. In each cycle, asynchronous and isochronous transfers cannot use more than the maximum payload size given in Table 19.6. If the isochronous traffic does not use the bandwidth, it is given to the asynchronous traffic. Thus, in the absence of isochronous traffic, asynchronous traffic throughput increases substantially.

19.8.4 Transactions

Asynchronous transactions typically follow a request and reply format. A sample transaction is shown in Figure 19.29. Each subaction begins with an arbitration phase to get the bus ownership. The winner of the arbitration uses the bus to send one request packet and after an acknowledge gap, the responder sends the acknowledgment packet. That completes one subaction. All

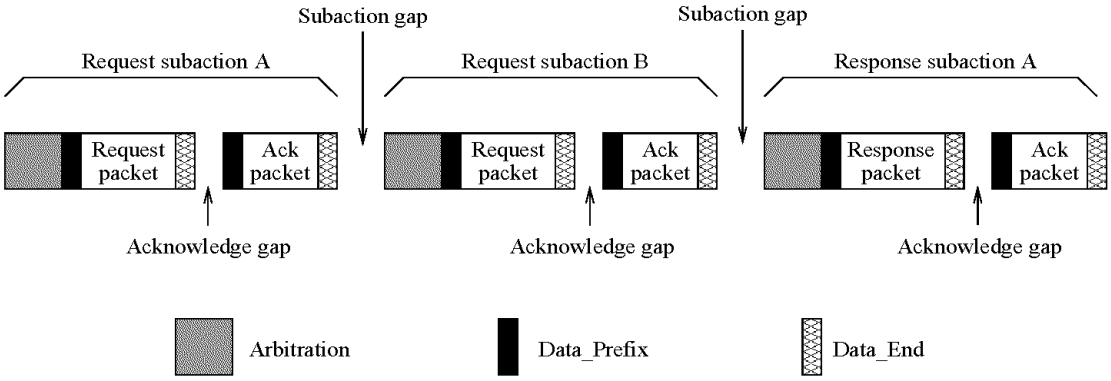


Figure 19.29 A sample sequence of asynchronous transactions.

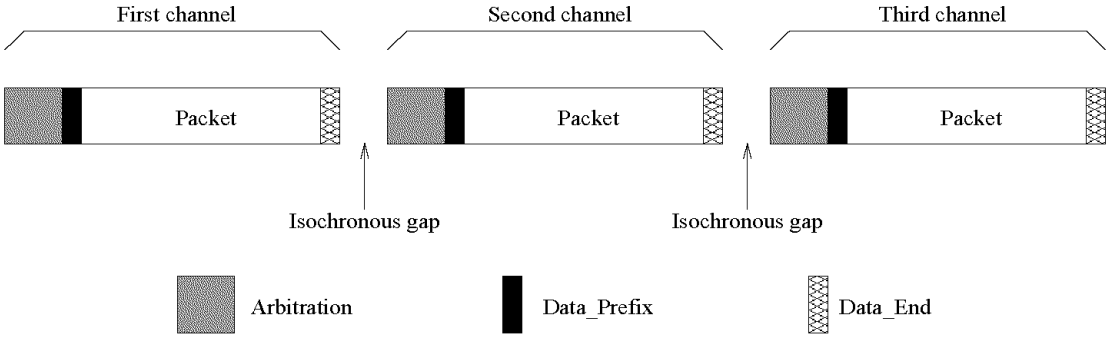


Figure 19.30 A sample isochronous transaction sequence.

nodes wait for a subaction gap before starting another arbitration. As shown in Figure 19.29, each packet is encapsulated between `Data_Prefix` and `Data_End`. The `Data_Prefix` field identifies the beginning of the data field and the end is indicated by `Data_End`.

The sample transaction sequence in Figure 19.29 shows a split transaction. That is, the request and response subactions of transaction A are split by a request subaction of transaction B. It is also possible to have concatenated transactions where the request and response are back to back. In this case, there is no need for the subaction gap.

Isochronous transactions are similar to their asynchronous counterparts, except that there is no acknowledge packet (see Figure 19.30). As are the asynchronous transactions, both concatenated and nonconcatenated isochronous transactions are supported. Figure 19.30 shows an example sequence of nonconcatenated transactions.

19.8.5 Bus Arbitration

Since 1394 supports peer-to-peer communication, we need a bus arbitration mechanism. The arbitration must respect the guaranteed bus bandwidth allocations to isochronous channels and a fairness-based allocation for asynchronous channels.

Asynchronous arbitration uses a fairness interval. During each fairness interval, all nodes that have a pending asynchronous transaction are allowed to obtain the bus ownership once. This is achieved by using an arbitration enable bit. Once an asynchronous channel acquires the bus ownership, its enable bit is cleared for the rest of the fairness interval.

Nodes that have pending isochronous transactions will go through an arbitration process during each cycle. Each node waits for 0.04 μs of bus idle time. This is referred to as the *isochronous gap*. The isochronous arbitration begins after this gap. The node that wins this arbitration uses its allocated slice for that cycle. The bus then returns to the idle state. After an isochronous gap, the remaining nodes again go through the arbitration process. This process repeats until all nodes with isochronous transactions have used their allocated bandwidth. Remember that these nodes will have to get their bandwidth allocations granted from the isochronous resource manager (IRM). Typically the root node serves as the IRM. This determination is done as part of the configuration process. IRM will not allocate the bandwidth unless the total allocated bandwidth is less than 80% of the total. The remaining bandwidth is used for asynchronous transactions. If there are no asynchronous transactions, the bus remains idle for the rest of the cycle (i.e., not given to the isochronous transactions).

19.8.6 Configuration

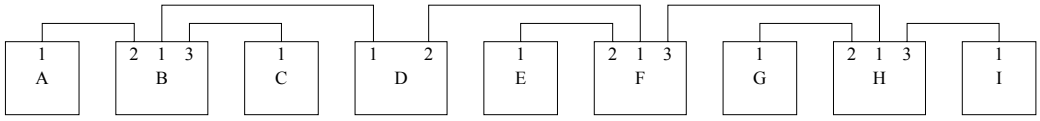
Bus configuration involves two main phases: *tree identification* and *self-identification*. As mentioned before, the configuration process does not require the host system. All devices connected to the bus configure themselves. The configuration process kicks in whenever the system is reset or a device is attached or removed. The first phase, the tree identification phase, is used to figure out the topology of the network. Once the topology has been identified, each node has to be assigned a unique id within the bus. This assignment is done by the self-identification phase. Next we describe these two phases by means of an example.

Tree Identification

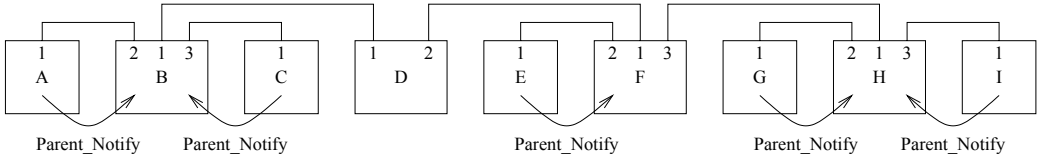
Two special signals are used for tree identification: `Parent_Notify` and `Child_Notify`. These two signals are hardware encoded as shown below:

`Parent_Notify`	TPA = 0	TPB = Z (high impedance state)
`Child_Notify`	TPA = 1	TPB = Z (high impedance state)

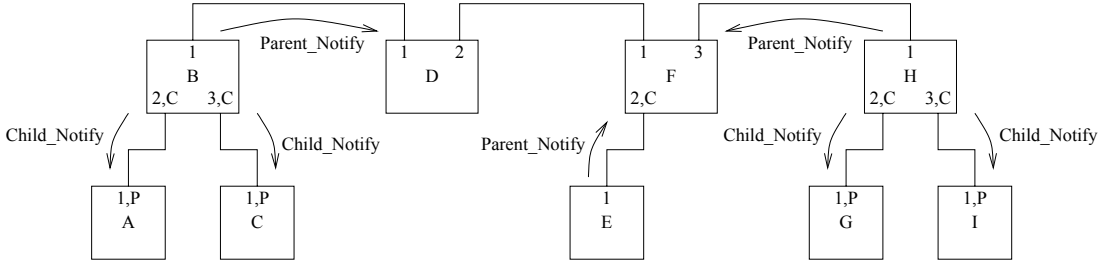
The tree identification process starts at the bottom (i.e., at leaf nodes) and propagates up to the root node. Since the topology is a hierarchy (i.e., tree), every node has a single parent except for the root node. A node C that knows that it is a child of another node P sends a `Parent_Notify` signal to node P. After receiving this signal, P marks C as its child. This process is repeated until all nodes identify their parents. The question is: How does a node identify that it is a child of another node?



(a) Original unconfigured network



(b) Leaf nodes send Parent_Notify signal to their parent nodes



(c) Topology starts to take shape with nodes A, C, E, G, and I identified as leaf nodes

Figure 19.31 The first few steps in the tree identification process.

- This is the simplest case. Leaf nodes, nodes that have only one connected port, know that they are the child nodes.
- If a node has n ports, it must receive the `Parent_Notify` signal on all but one port (say, port x) to know that it is a child of the node connected to port x.

Let's look at the example shown in Figure 19.31a. The tree identification process is started by the leaf nodes A, C, E, G, and I. These nodes send the `Parent_Notify` signal to the nodes connected to the only port they have (Figure 19.31b). Note that these nodes continue to send the `Parent_Notify` signals until the `Child_Notify` signal is received by them. This will automatically take care of any lost or corrupt signals.

Once the `Parent_Notify` signals from nodes A and C are received by branch B, it will send its own `Parent_Notify` signal to node D. Simultaneously, it sends `Child_Notify` signals to nodes A and C and marks ports 2 and 3 as child ports. Similarly, branch H also sends a `Parent_Notify` signal to branch F and `Child_Notify` messages to G and I (see Figure 19.31c). Branch H marks ports 2 and 3 as child ports.

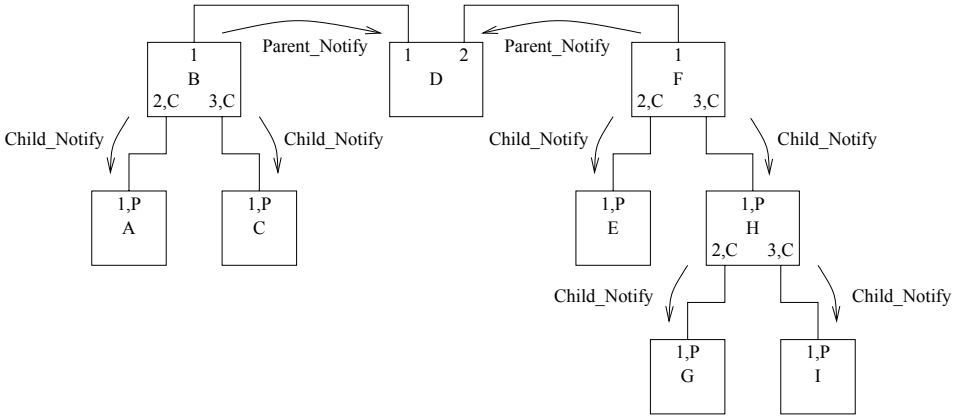


Figure 19.32 All leaf nodes have been identified.

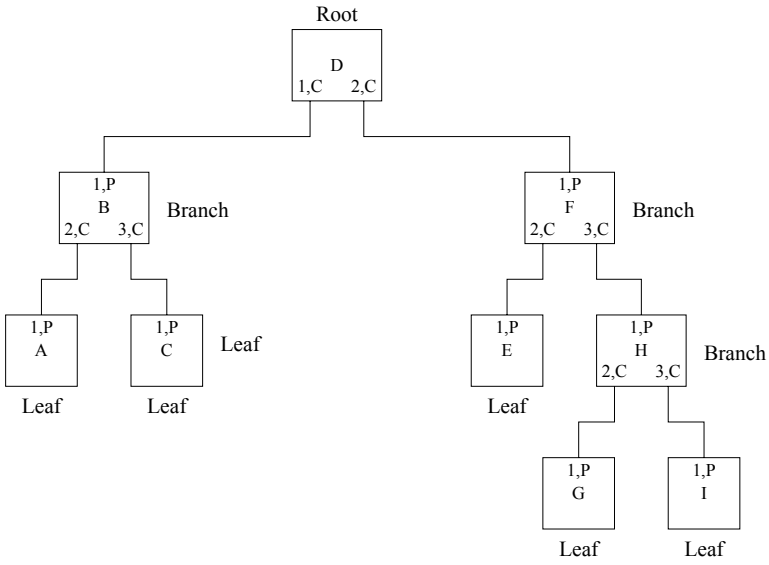


Figure 19.33 Final topology after the completion of the tree identification process.

At this stage, branch F is not qualified to send the `Parent_Notify` signal because it has not received the `Parent_Notify` signal on more than one port. In the next phase, branch F sends a `Parent_Notify` signal to branch D (Figure 19.32). Having already received a `Parent_Notify` signal from branch B, branch D considers itself the root as it has identified all its ports as child ports. The final topology is shown in Figure 19.33.

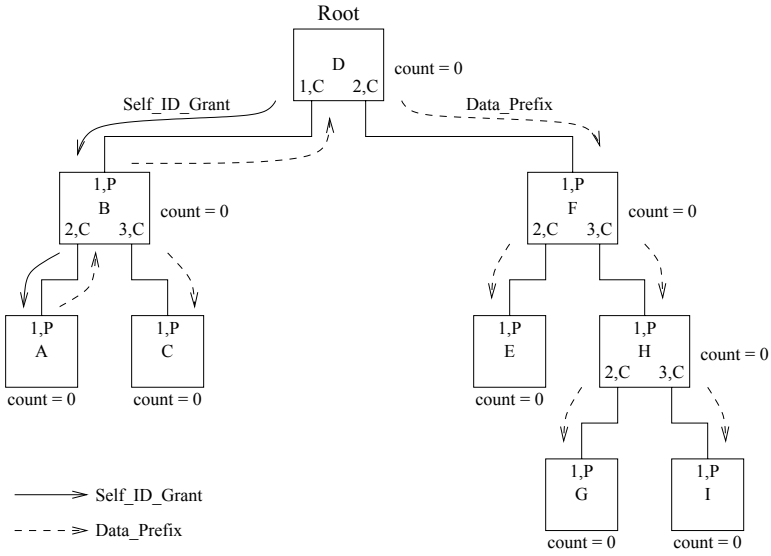


Figure 19.34 Initial network with count values initialized to zero.

Self-Identification

The self-identification phase configures the nodes. This configuration process assigns physical ids to the nodes. In addition, neighboring nodes exchange their transmission speed capabilities. Furthermore, the bus topology information gathered during the tree-identification phase is broadcast to all nodes. Note that, in the tree-identification phase, each node knows only about its own child and parent entities. By broadcasting this information, each node will have the topology of the bus.

During the self-identification phase, since the speeds of the nodes are not known, all nodes operate at the lowest speed of 100 Mbps. Self-identification is done using three special signals: `Self_ID_grant`, `Data_prefix`, and `Identification_done`. In contrast to the tree-identification process, the root node initiates the self-identification phase. Initially, all nodes reset their count value to zero (Figure 19.34). The root node issues a `Self_ID_grant` on its lowest numbered port and `Data_prefix` on all the other ports. Each branch node that receives the `Self_ID_grant` forwards it on its lowest numbered port and `Data_prefix` on all the other ports. At the end of this process, only node A receives the `Self_ID_grant` signal, as shown in Figure 19.34.

Node A assigns its count value zero as its physical id. Then node A broadcasts a self-id packet giving its id and its bus characteristics (e.g., 200 Mbps speed). This broadcast is initiated by node A by sending a self-id packet to node B (Figure 19.35). Node B forwards this on ports 1 and 3 to nodes C and A. Each branch node repeats this process to complete the broadcast. Thus, all nodes know that node A assigned itself zero as its physical id. Node A

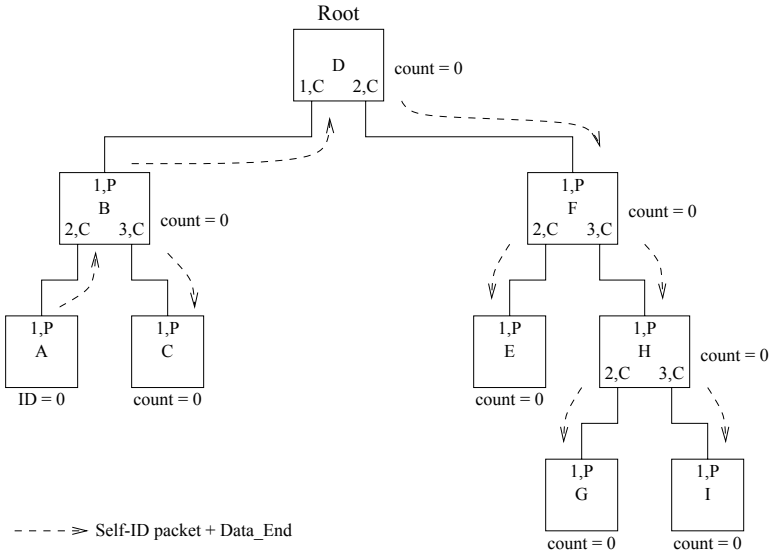


Figure 19.35 Node A receives the grant message and assigns itself physical node id zero.

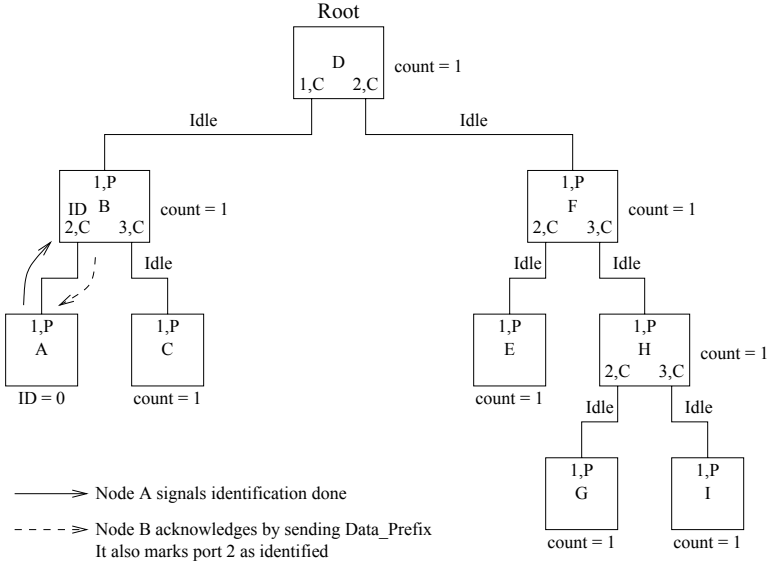


Figure 19.36 Node A completes the assignment process by sending the identification done signal and node B acknowledges by sending the `Data_prefix` signal. All other nodes, having received the `self_ID` message, update their count values to 1.

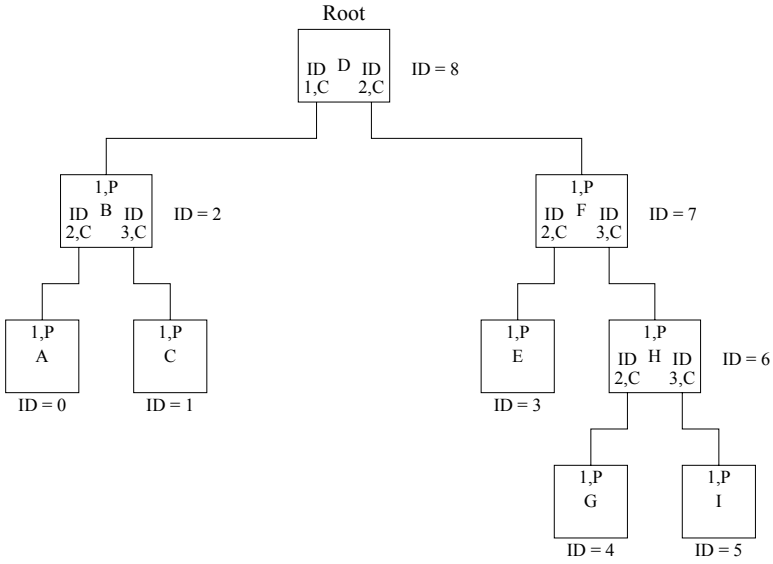


Figure 19.37 Final assignment of node ids.

then sends the `Identification_done` signal to node B, which acknowledges by sending a `Data_prefix` signal. Node B also marks this port as identified, as shown in Figure 19.36.

After the first self-id packet, the root node detects an idle period on the network and initiates the second round. During this round, node B forwards the `Self_ID_grant` signal to the lowest unidentified port (i.e., port 3 in the example). Node C assigns itself a physical id of 1 and broadcasts to all nodes like node A. This process is repeated until all nodes get a physical id. Figure 19.37 shows the final physical id assignment.

Self-id packets also carry information about the capability of a node to become the isochronous resource manager (IRM). If there is more than one contender for the IRM, the node with the highest physical id gets to be the IRM. Since the root node has the highest id, it often acts as the IRM for the network.

19.9 The Bus Wars

We have discussed several bus standards. Of these, the EIA-232 and parallel interfaces are certain to disappear in the near future. Their functionality is easily supported by the USB. Thus, we don't need special connectors for these two interfaces.

Of the other three standards, SCSI is the parallel interface, and the USB and IEEE 1394 use the serial interface. Each bus has future expansion plans to compete and survive. The SCSI Trade Association (STA) has plans for Ultra5 SCSI by 2003, with the bandwidth scaling up to 640 MB/s. SCSI is dominant for connecting disk drives and other storage devices, particularly for midrange to high-end users requiring external connectivity. For example, currently about 95% of high-end disk drives use a SCSI interface.

As we have seen in the USB section, USB 2.0 can compete well with SCSI 3 to connect a single disk drive. USB 2.0 provides about 60 MB/s data transfer rate. However, even the Ultra 2 SCSI provides a higher bandwidth and supports up to 16 devices. Even though in 1999 disk drives' transfer rate is 20 MB/s, it is expected to go up to 88 MB/s in 2003. At that time, USB 2.0 will not support even four drives as they transfer about 353 MB/s. A significant challenge for SCSI is to overcome its 12-meter cable length restriction cost effectively.

The USB 2.0 plays a key role in multimedia and Internet access applications, and wireless LANs. Even CD burners can exploit the USB 2.0's high speed. Windows 2000 and Millennium Edition support the USB 2.0. This can make the 2.0 successful just as Windows 98 played a key role in making the original USB interface a success.

IEEE 1394 is dominant in digital video applications. As mentioned, it will be useful in peer-to-peer applications, whereas the USB will be dominant in low-cost, host-to-peripheral applications. There is room for both interfaces. Current systems support both interfaces, particularly systems with digital video application support. There is also support for both standards from the industry. For example, Lucent produces a single controller that handles both USB and 1394.

Surely, we have not discussed all the bus types here. Our objective is to discuss the dominant and popular ones in order to give you an idea of their characteristics. Once you understand these basics, you can explore the other, less popular, bus standards.

19.10 Summary

We have looked at the I/O device interface in this chapter. I/O devices are usually interfaced to the system by means of an I/O controller. The I/O controller serves two important purposes: taking care of device-specific low-level details, and isolating the system bus from the electrical interface details. When we talk about an I/O device, we are not referring to the actual device, but to the associated controller.

I/O devices typically consist of several internal registers, including the command, data, and status registers. The processor communicates with an I/O device via these registers. For example, it can write to the command register to direct the associated I/O device to perform a specific task. The data register can be used for data transfer. The status register provides information on the status of the I/O device and the requested I/O operation.

The internal registers are accessed through I/O ports. I/O ports can be mapped into the memory address space or into a separate I/O address space. The former mapping is known as memory-mapped I/O. In isolated I/O, port addresses are mapped to a separate I/O address space. Processors like the Pentium support isolated I/O. Other processors like the PowerPC and MIPS support memory-mapped I/O. Isolated I/O requires separate I/O instructions in order to access the ports. Memory-mapped I/O, on the other hand, uses the standard memory read/write instructions to access I/O ports.

I/O data transfer involves two main phases: a data transfer phase to move data, and a transfer termination phase to indicate the end of the transfer operation. We can use either programmed

I/O or DMA to transfer data. The end of data transmission can be done with an interrupt or by using the programmed I/O method.

Programmed I/O involves the processor in the transfer process. Typically, the processor executes a busy-wait loop to transfer data. In DMA transfer, the processor initiates the transfer process by providing the necessary information to the DMA controller as to the type of operation, size of data to be transferred, and so on. The DMA controller is in charge of the actual data transfer process. Once the transfer is completed, the processor is notified. The DMA operation is more efficient than the programmed I/O-based data transfer. We have not discussed the interrupt mechanism in this chapter. The next chapter presents a detailed discussion of interrupts.

Data transmission can be done in either serial or parallel mode. Both modes are used in I/O interfaces. The parallel printer port is an example of the parallel port. The typical modem serial interface uses the EIA-232 standard. We have discussed the SCSI parallel interface, which is used to interface disk drives and other storage devices. We have also presented details on two external serial interfaces: the USB and IEEE 1394. The USB is very popular with low- to medium-speed I/O devices. It is processor-centric and typically supports host-to-peripheral applications. The IEEE 1394, which is also known as the FireWire, supports medium- to high-speed devices on a peer-to-peer basis. Most current systems provide these two interfaces.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Asynchronous transmission
- Bus arbitartion
- Cyclic redundancy check (CRC)
- Direct memory access
- Direction flag
- DMA acknowledge
- DMA controller
- DMA request
- EIA-232 serial interface
- Error correction
- Error detection
- FireWire
- Gray code
- Hold acknowledge
- IEEE 1394
- I/O address space
- I/O controller
- I/O port
- Isolated I/O
- Memory-mapped I/O
- Parallel interface
- Parity encoding
- Peripheral device
- Programmable peripheral interface
- Programmed I/O
- RS-232 serial interface
- SCSI bus
- SECDED
- Serial transmission
- Synchronous transmission
- Universal Serial Bus (USB)

19.11 Web Resources

Information on the CRC generator/checker chip 74F401 is available from Fairchild Semiconductor at `www.fairchildsemi.com/pf/74/74F401.html`.

Details on SCSI are available from the SCSI Trade Association at `www.scsita.org`.

For information on the USB standard, see `www.usb.org`.

Information on IEEE 1394 is available from the 1394 Trade Association at `www.1394TA.org`. Also, see the Apple site `www.apple.com/usb` for information on FireWire.

19.12 Exercises

19–1 Explain why I/O controllers are used to interface I/O devices to the system.

19–2 Describe the two I/O mapping methods: memory-mapped I/O and isolated I/O.

19–3 Discuss the differences between memory-mapped and isolated I/O.

19–4 We have said that the Pentium supports isolated I/O. Does this mean it does not support memory-mapped I/O?

19–5 Why do processors that support only memory-mapped I/O not provide separate I/O instructions?

19–6 What is the relationship between the key scan code and its ASCII value?

19–7 Pentium allows two formats—direct and indirect—for `in` and `out` instructions. What is the rationale for this?

19–8 Figure 19.2 on page 774 shows how the four ports of the 8255 PPI are mapped to addresses 60H through 63H. Modify this figure to map these four ports to E0H through E3H.

19–9 Is it possible to map the four 8255 I/O ports to addresses 62H to 65H? Explain.

19–10 What are the pros and cons of programmed I/O over DMA?

19–11 What is the purpose of the temporary register in the Intel 8237?

19–12 To correct a single-bit error, we add several parity bits to the data to derive a codeword. Each parity bit is responsible for checking certain bits of the codeword. Devise a way to remember these bit positions. To test your method, write down the bit positions checked by parity bits P_1 and P_4 in a 14-bit codeword.

19–13 How many parity bits p do you need to detect single-bit errors in an n-bit datum? That is, derive a general formula that tells you the value of p.

19–14 Compute the overhead of the error-correcting code discussed in this chapter. Express this overhead as a percentage of the data size. Use your judgment to figure out a way to present your data.

19–15 Compute the CRC code (i.e., remainder) for the data `1000101110100011`. Use the polynomial used in Example 19.1 on page 789. Write the codeword that would be transmitted.

19–16 Suppose a receiver received the codeword `101000110101110`. Verify that the codeword has been received incorrectly. Use the polynomial $x^5 + x^4 + x^2 + 1$.

19–17 Suppose a receiver received the codeword `101000010101110`. Verify that the code has been received correctly. Use the polynomial $x^5 + x^4 + x^2 + 1$.

19–18 We have given a CRC generator circuit in Figure 19.10 (page 791). Give a similar circuit for the polynomial $x^5 + x^2 + 1$. Trace your circuit to find the remainder generated for 10100101 (we have used the same data in Figure 19.8 on page 790). Verify that the remainder is 01110.

19–19 Implement a CRC generator circuit for the polynomial $x^5 + x^4 + x^2 + 1$. Trace your circuit for the input data `1010001101`. Make to sure to append five zeros to the input data. The remainder will be in the shift register after shifting the last bit. Verify that this remainder is correct by using the division process used in Figure 19.8 on page 790.

19–20 The 74F401 chip can be programmed to use one of several polynomials using the selection inputs S_2, S_1, S_0. To understand how this chip implements the CRC generator circuit, design a circuit that implements the following general polynomial:

$$G = \sum_{i=0}^{n} a_i x^i.$$

Hint: Use the XOR gate as a programmable inverter.

19–21 Design a digital logic circuit that uses the error detection circuit shown in Figure 19.7 to correct the error bit. *Hint:* Use a multiplexer and XOR gates to invert the error bit. Remember that the XOR gate can be used as a programmable inverter.

19–22 What is the purpose of the start bit in asynchronous transmission?

19–23 What is the purpose of the stop bit(s) in asynchronous transmission?

19–24 What are the motivating factors for proposing the USB and IEEE 1394 external interfaces?

19–25 USB uses the NRZI encoding as opposed to the simple NRZ encoding. Discuss the advantages of NRZI over NRZ encoding.

19–26 Since USB does not use interrupts in the traditional sense, how can external devices obtain interrupt services?

19–27 What are the differences between UHC and OHC?

19–28 On page 807, we have stated that seven-port USB hubs are not bus-powered but four-port hubs are. Explain why.

19–29 Explain why bus arbitration is required in IEEE 1394 but not in USB.

19–30 Explain how the IRM selection is done in IEEE 1394.

Chapter 20

Interrupts

Objectives

- To describe the interrupt mechanism of the Pentium;
- To explain software and hardware interrupts;
- To discuss DOS and BIOS interrupt services to interact with I/O devices such as the keyboard;
- To illustrate how user-defined interrupt handlers are written;
- To discuss how single-stepping is implemented in the Pentium;
- To describe the interrupt mechanism of PowerPC and MIPS processors.

Interrupts, like procedures, can be used to alter a program's flow of control to a procedure called the interrupt service routine or handler. For most of this chapter, we focus on Pentium interrupts.

Unlike procedures, which can be invoked by the `call` *instruction, interrupt service routines can be invoked either in software (called software interrupts), or by hardware (called hardware interrupts). Interrupts are introduced in the first section. Section 20.2 discusses a taxonomy of Pentium interrupts. The interrupt invocation mechanism of the Pentium is described in Section 20.3.*

Both DOS and BIOS provide several software interrupt service routines. Software interrupts are introduced in Section 20.4. This section also discusses the keyboard services of DOS and BIOS. Section 20.5 discusses exceptions. Exceptions are like interrupts, except that they are caused by an event within the processor such as an attempt to divide by zero.

Hardware interrupts are introduced in Section 20.6. Hardware interrupts deal with interrupt requests from the I/O devices. We use the keyboard to illustrate how interrupt handlers are written in Pentium-based systems. We briefly discuss the interrupt mechanisms of PowerPC and MIPS processors in Sections 20.7 and 20.8. The last section summarizes the chapter.

20.1 Introduction

The interrupt is a mechanism by which a program's flow of control can be altered. We have seen two other mechanisms that do the same: *procedures* and *jumps*. Jumps provide a one-way transfer of control, and procedures provide a mechanism to return control to the point of calling when the called procedure is completed.

Interrupts provide a mechanism similar to that of a procedure call. Causing an interrupt transfers control to a procedure, which is referred to as an *interrupt service routine.* An interrupt service routine is commonly called an interrupt *handler*. When the interrupt handler execution is done, the original program resumes execution as if it were not interrupted. This behavior is analogous to a procedure call. There are, however, some basic differences between procedures and interrupts that make interrupts almost indispensable.

One of the main differences is that interrupts can be initiated by both software and hardware. In contrast, procedures are purely software-initiated. The fact that interrupts can be initiated by hardware is the principal factor behind the power of the interrupt mechanism. This capability gives us an efficient way by which external devices (outside the CPU) can get the attention of the CPU.

Software-initiated interrupts—called simply *software interrupts*—are caused by executing a processor instruction. In the Pentium, software interrupts are caused by executing the `int` instruction. PowerPC and MIPS processors also have an instruction to generate interrupts. The PowerPC uses the *system call* (`sc`) instruction to cause interrupts. The MIPS processor also uses the *system call* (`syscall`) instruction for software interrupts. In fact, in Chapter 15, we used this instruction to invoke the SPIM simulator I/O services.

Thus, software interrupts, like procedure calls, are anticipated or planned events. For example, when you are expecting a response from the user (e.g., `Y` or `N`), you can initiate an interrupt to read a character from the keyboard. What if an unexpected situation arises that requires the immediate attention of the CPU? For example, you have written a program to display the first 90 Fibonacci numbers on the screen. While running the program, however, you have realized that your program never terminates because of a simple programming mistake (e.g., you forgot to increment the index variable controlling the loop). Obviously, you want to abort the program and return control to the operating system. As you know, in most cases this can be done by `ctrl-break`. For this example, `ctrl-break` certainly works. The important point is that this is not an anticipated event, so it cannot be programmed into the code. Strictly speaking, we can include code to handle all possible events, or at least most likely events, but such an alternative makes the program very inefficient.

The interrupt mechanism provides an efficient way to handle unanticipated events. Referring to the previous example, the `ctrl-break` could cause an interrupt to draw the attention of the CPU away from the user program. The interrupt service routine associated with `ctrl-break` can terminate the program and return control to the operating system.

We see two terms in this chapter: *interrupts* and *exceptions*. Interrupts were originally proposed as a way to handle unanticipated events such as requests for service from I/O devices. Later, the same mechanism was extended to handle internal events such as arithmetic overflow

and illegal instruction execution. As a result, different processors use different terminology. Processors like the Pentium distinguish between interrupts and exceptions. Other processors like the MIPS and PowerPC use these two interchangeably. Since the underlying mechanism is essentially the same, we use the term interrupt in our explanation.

As do procedures, interrupts involve transferring control to the interrupt handler and returning control to the interrupted location after executing the handler. To transfer control to the handler, the processor needs to know the type of interrupt. This type of information can be obtained in one of two ways. In a vectored interrupt mechanism, each interrupt type is assigned a specific address. When an interrupt occurs, the corresponding handler is invoked by transferring control to the type-specific handler. The Pentium and PowerPC use vectored interrupts.

An alternative way is to indicate the cause of the interrupt. In this case, all interrupts transfer control to a common interrupt handler. This handler looks at the cause register to identify the interrupt type and transfers control to the appropriate point in the operating system. As we show later, the MIPS processor uses this kind of mechanism to process interrupts.

For most of this chapter, we focus on the Pentium interrupt mechanism. We use several examples to illustrate interrupt-driven I/O. Once you know the Pentium interrupt processing mechanism, it is easy to see how the interrupt processing is done in the PowerPC and MIPS processors.

20.2 A Taxonomy of Pentium Interrupts

We have already identified two basic categories of interrupts: software-initiated and hardware-initiated (see Figure 20.1). The third category is called *exceptions*. Exceptions handle instruction faults. An example of an exception is the divide error fault, which is generated whenever divide by 0 is attempted. This error condition occurs during the `div` or `idiv` instruction if the divisor is 0. Later, we see an example of this fault in Section 20.5, which discusses exceptions in detail.

Software interrupts are written into a program by using the `int` instruction. The main use of software interrupts is in accessing I/O devices such as a keyboard, printer, display screen, disk drive, and the like. Software interrupts can be further classified into *system-defined* and *user-defined*. There are two types of system-defined software interrupts: interrupt services supported by DOS, and those supported by BIOS (basic input/output system). The BIOS is the lowest-level software that is stored in ROM. Note that DOS and other application software are loaded from disk.

The interrupt service routines provided by DOS and BIOS are not mutually exclusive. There are some services, such as reading a character from the keyboard, provided by both DOS and BIOS. In fact, DOS uses BIOS-supported routines to provide some services that control the system hardware (see Figure 20.2).

Hardware interrupts are generated by hardware devices to get the attention of the CPU. For example, when you strike a key, the keyboard hardware generates an external interrupt causing the CPU to suspend its present activity and execute the keyboard interrupt handler to process the key. After completing the keyboard handler, the CPU resumes what it was doing before the interruption.

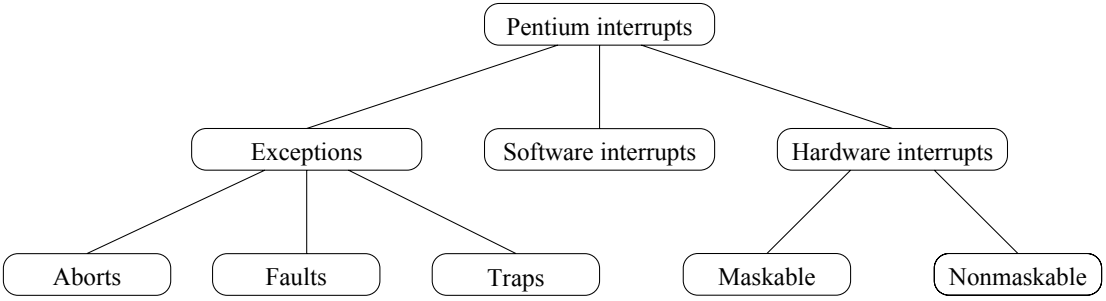


Figure 20.1 A taxonomy of Pentium interrupts.

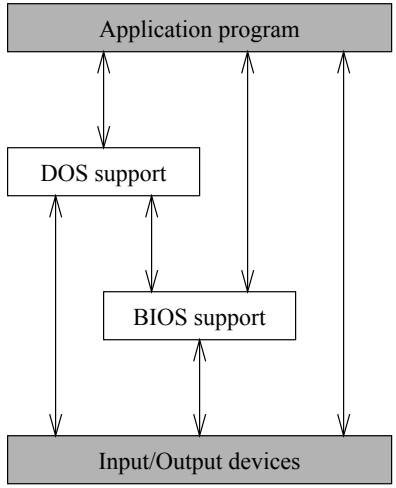


Figure 20.2 Various ways of interacting with I/O devices in a PC.

Hardware interrupts can be either *maskable* or *nonmaskable*. The nonmaskable interrupt (NMI) is always attended to by the CPU immediately. One example of an NMI is the RAM parity error indicating memory malfunction. Maskable interrupts can be delayed until execution reaches a convenient point. As an example, let us assume that the CPU is executing a `main` program. An interrupt occurs. As a result, the CPU suspends the `main` as soon as it finishes the current instruction of `main` and the control is transferred to ISR1. If ISR1 has to be executed without any interruption, the CPU can mask further interrupts until ISR1 is completed. Suppose that, while executing ISR1, another maskable interrupt occurs. Service to this interrupt would have to wait until ISR1 is completed.

20.3 Pentium Interrupt Processing

This section describes the interrupt processing mechanism of the Pentium in protected and real modes.

20.3.1 Interrupt Processing in Protected Mode

Unlike procedures, where a name is given to identify a procedure, a type number identifies interrupts. The Pentium supports 256 different interrupt types. Interrupt types range between 0 and 255. The interrupt type number is used as an index into a table that stores the addresses of interrupt handlers. This table is called the *interrupt descriptor table* (IDT). Like the global and local descriptor tables (GDT and LDT, as discussed in Chapter 7), each descriptor (or vector as they are often called) is essentially a pointer to an interrupt handler and requires 8 bytes. The interrupt type number is scaled by 8 to form an index into the IDT.

The IDT may reside anywhere in physical memory. The location of the IDT is maintained in an IDT register IDTR. The IDTR is a 48-bit register that stores 32 bits of IDT base address and a 16-bit IDT limit value. However, the IDT does not require more than 2048 bytes, as there can be at most 256 descriptors. In a system, the number of descriptors could be much smaller than the maximum allowed. In this case, the IDT limit can be set to the required size. If a descriptor is referenced that is outside the limit, the processor enters shutdown mode.

There are two special instructions to load (`lidt`) and store (`sidt`) the contents of the IDTR register. Both instructions take the address of a 6-byte memory as the operand. In the next subsection, we describe interrupt processing in real mode, which is the focus of this chapter.

20.3.2 Interrupt Processing in Real Mode

In real mode, the IDT is located at base address 0. Each vector takes only 4 bytes as opposed to 8 bytes in protected mode. Each vector consists of a CS:IP pointer to the associated handler: two bytes for specifying the code segment (CS), and two bytes for the offset (IP) within the code segment. Figure 20.3 shows the interrupt vector table layout in the memory.

Since each entry in the interrupt vector table is 4 bytes long, the interrupt type is multiplied by 4 to get the corresponding interrupt handler pointer in the table. For example, `int 2` can find the interrupt handler pointer at memory address $2 \times 4 = 00008$H. The first 2 bytes at the specified address are taken as the offset value and the next 2 bytes as the CS value. Thus, executing `int 2` causes the CPU to suspend its current program, calculate the address in the interrupt vector table (which is $2 \times 4 = 8$ for this example), read CS:IP values, and transfer control to that memory location.

Just as procedures do, interrupt handlers should end with a return statement to send control back to the interrupted program. The interrupt return (`iret`) is used for this purpose. A typical interrupt handler structure is shown below:

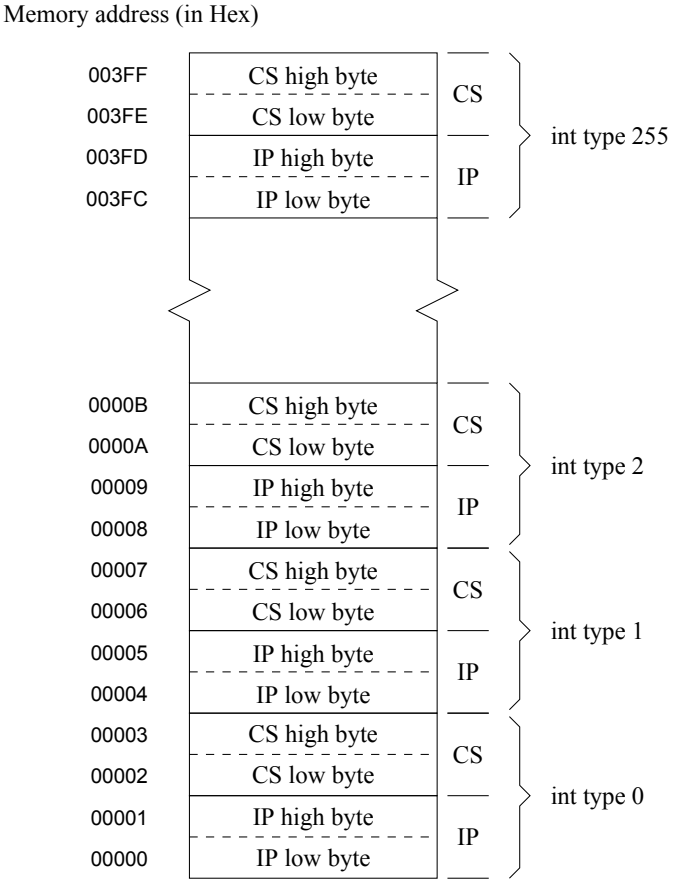


Figure 20.3 Interrupt vector table in memory (real mode).

```
;save the registers used in the ISR
sti     ; enable further interrupts
. . .
. . .
ISR body
. . .
. . .
; restore the saved registers
iret   ; return to the interrupted program
```

When an interrupt occurs, the following actions are taken:

1. Push the flags register onto the stack;
2. Clear the interrupt enable and trap flags;
3. Push the CS and IP registers onto the stack;
4. Load the CS with the 16-bit data at memory address (interrupt-type $\times$ 4 + 2);
5. Load the IP with the 16-bit data at memory address (interrupt-type $\times$ 4).

Note that EIP is used instead of IP for 32-bit segments. On receiving an interrupt, the flags register is automatically saved on the stack. The interrupt enable flag is cleared. This disables attending further interrupts until this flag is set. Usually, this flag is set in interrupt handlers unless there is a special reason to disable other interrupts. The interrupt flag can be set by `sti` and cleared by `cli` assembly language instructions. Neither instruction requires any operands.

The current CS and IP values are pushed onto the stack. In most cases, these values (i.e., CS:IP) point to the instruction following the current instruction. (See Section 20.5 for a discussion of a different behavior in the case of a fault.) If it is a software interrupt, CS:IP points to the instructions following the `int` instruction. The CS and IP registers are loaded with the address of the interrupt handler from the interrupt vector table.

The last instruction of an ISR is the `iret` instruction and serves the same purpose as `ret` for procedures. The actions taken on `iret` are as follows:

1. Pop the 16-bit value from the stack into the IP register;
2. Pop the 16-bit value from the stack into the CS register;
3. Pop the 16-bit value from the stack into the flags register.

In other words, the top three words from the stack are loaded into IP, CS, and flags registers.

20.4 Pentium Software Interrupts

Software interrupts are initiated by executing

```
int     interrupt-type
```

where `interrupt-type` is an integer in the range 0 through 255 (both inclusive). Thus, a total of 256 different types is possible. This is a sufficiently large number, as each interrupt type can be parameterized to provide several services. For example, all DOS services are provided by `int 21H`. There are more than 80 different services (called functions) provided by DOS. Registers are used to pass parameters and to return values. We discuss some of the `int 21H` services throughout this chapter.

DOS and BIOS provide several interrupt service routines to access I/O devices. The following sections describe the keyboard services and explain by means of examples how they can be used.

20.4.1 DOS Keyboard Services

DOS provides several interrupt services to interact with the keyboard. All DOS interrupt services are invoked by `int 21H` after setting up registers appropriately. The AH register should always be loaded with the desired function number. The following seven functions are provided by DOS to interact with the keyboard:

Function 01H: *Keyboard input with echo.*

Input: AH = 01H;
Returns: AL = ASCII code of the key entered.

This function can be used to read a character from the keyboard buffer. If the keyboard buffer is empty, this function waits until a character is typed. The received character is echoed to the display screen. If the character is a `ctrl-break`, an interrupt 23H is invoked to abort the program.

Function 06H: *Direct console I/O.* There are two subfunctions associated with this function: keyboard input or character display. The DL register is used to specify the desired subfunction.

Subfunction: *Keyboard input.*

Inputs: AH = 06H,
DL = FFH;
Returns: ZF = 0 if a character is available,
In this case, the ASCII code of the key
is placed in the AL register.
ZF = 1 if no character is available.

If a character is available, the zero flag (ZF) is cleared (i.e., ZF = 0), and the character is returned in the AL register. If no character is available, this function does not wait for a character to be typed. Instead, control is returned immediately to the program and the zero flag is set (i.e., ZF = 1). The input character is not echoed. No `ctrl-break` check is done by this function.

Subfunction: *Character display.*

Inputs: AH = 06H;
DL = Character to be displayed
(it should not be FFH);
Returns: Nothing.

The character in the DL register is displayed on the screen.

Function 07H: *Keyboard input without echo or* `ctrl-break` *check.*

Input: AH = 07H;
Returns: AL = ASCII code of the key entered.

This function waits for a character from the keyboard and returns it in AL as described in function 01H. The difference between this function and function 01H is that this function does not echo the character, and no `ctrl-break` service is provided. This function is usually used to read the second byte of an extended-keyboard character (see page 834).

Function 08H: *Keyboard input without echo.*

Input: AH = 08H;
Returns: AL = ASCII code of the key entered.

This function provides similar service to that of function 07H except that it performs a `ctrl-break` check. As a result, this function is normally used to read a character from the keyboard when echoing is not needed.

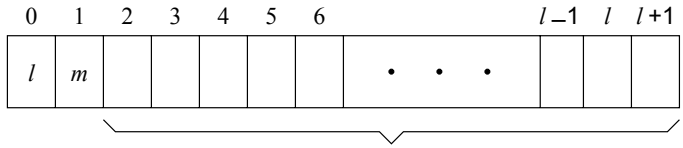
Function 0AH: *Buffered keyboard input.*

Inputs: AH = 0AH;
DS:DX = Pointer to the input buffer
(first byte of the input buffer
should have the buffer size);
Returns: Character string in the input buffer.

This function can be used to input a character string (terminated by the carriage return) into a buffer within the calling program. Before calling this function, DS:DX should be loaded with a pointer to the input buffer and the first byte of this buffer must contain a nonzero value representing the string length to be read including the carriage return.

The input character string is placed in the buffer starting at the third byte of the buffer. Characters are read until either the Enter key is pressed or the buffer is filled to one less than its length. When the Enter key is pressed to terminate the input, 0DH is stored in the buffer and the number of characters in the buffer (excluding the carriage return character) is placed in the second byte of the input buffer.

When the input buffer is filled to one less than its length before encountering the Enter key, all keys except Enter and Backspace keys are rejected, and this rejection is indicated by a beep.



Input buffer for character string

l = maximum number of characters (given as input),
m = indicates the actual number of characters in the input buffer excluding the carriage return (returned by the function).

Function 0BH: *Check keyboard buffer.*

Input: AH = 0BH;
Returns: AL = 00H if the keyboard buffer is empty,
AL = FFH if the keyboard buffer is not empty.

This function can be used to check the status of the keyboard buffer. It returns 00H in AL if the keyboard buffer is empty, and returns FFH in AL if the buffer has at least one character. A `ctrl-break` check is done by this function. The keyboard buffer is not modified in any way.

Function 0CH: *Clear keyboard buffer.*

Inputs: AH = 0CH,
AL = 01H, 06H, 07H, 08H, or 0AH;
Returns: Depends on the AL contents (see below).

This function can be used to clear the keyboard buffer to discard any typeahead input entered by the user. If AL is 01H, 06H, 07H, 08H, or 0AH, then an appropriate DOS function is performed following the buffer flush. If AL contains any other value, nothing is done after clearing the buffer.

Extended Keyboard Keys

The IBM PC keyboard has several keys that are not part of the ASCII character set. These keys include the function keys, cursor arrows, Home, End, and so on. These keys are called *extended keys*. When an extended key is pressed, the first byte placed in the keyboard buffer is 00H and the second byte is the keyboard scan code for the key. Table 19.1 on page 773 lists the keyboard scan codes for the extended keys. In contrast, when an ASCII key is pressed, the first byte in the keyboard buffer (which is 30 bytes long to store 15 typeahead keys with two bytes for each key) is the ASCII code for the key, and the second byte is the scan code of the key.

To read a character from the keyboard using DOS functions, extended keys require two function calls, as shown in the following procedure:

```
 Read the next character code into AL using function 08H
if (AL ≠ 0)
then
     AL = ASCII code (ASCII character)
else      {extended key}
     read the scan code of the extended key into AL using
          function 07H
     AL = scan code (extended key character)
end if
```

Example 20.1 *Get string procedure.*
In this example, we look at the `GetStr` procedure that we have used to read a string from the keyboard. The `GetStr` is a macro (see the `io.mac` file listing) that can receive up to two parameters: a pointer to a buffer to store the input string, and an optional buffer length value. The macro, after checking the parameters, places the buffer pointer in AX and the buffer length in CX and calls the procedure `proc_GetStr`. This procedure actually reads a string from the keyboard using the buffered keyboard input function 0AH. The pseudocode for the procedure is as follows:

```
proc_GetStr ()
    save registers used in the procedure
    if (CX < 2)
    then
        CX := 2
    else
        if (CX > 81)
        then
            CX := 81
        end if
    end if
    use function 0AH to read input string into
        temporary buffer str_buffer
    copy input string from str_buffer to
        user buffer and append NULL
    restore registers
    return
end proc_GetStr
```

Program 20.1 gives the assembly language code for the this procedure. It follows the pseudocode in a straightforward manner. It uses the `DOScall` macro, which is defined below:

```
DOScall   MACRO   function_number
          mov     AH,function_number
          int     21H
          ENDM
```

Program 20.1 Procedure to read a string from the keyboard

```
 1:  ;-------------------------------------------------------------
 2:  ; Get string (of maximum length 80) from keyboard.
 3:  ;     AX <-- pointer to a buffer to store the input string
 4:  ;     CX <-- buffer size = string length + 1 for NULL
 5:  ; If CX < 2,  CX := 2 is used to read at least one character.
 6:  ; If CX > 81, CX := 81 is used to read at most 80 characters.
 7:  ;-------------------------------------------------------------
 8:  proc_GetStr  PROC
 9:        push       DX      ; save registers
10:        push       SI
11:        push       DI
12:        push       ES
13:        mov        DX,DS   ; set up ES to point to DS
14:        mov        ES,DX   ;  for string instruction use
15:        mov        DI,AX   ; DI = buffer pointer
16:        ; check CX bounds
17:        cmp        CX,2
18:        jl         set_CX_2
19:        cmp        CX,81
20:        jle        read_str
21:        mov        CX,81
22:        jmp        SHORT read_str
23:  set_CX_2:
24:        mov        CX,2
25:  read_str:
26:        ; use temporary buffer str_buffer to read the string
27:        ;  in using function 0AH of int 21H
28:        mov        DX,OFFSET str_buffer
29:        mov        SI,DX
30:        mov        [SI],CL  ; first byte = # of chars. to read
31:        DOScall    0AH
32:        inc        SI       ; second byte = # of chars. read
33:        mov        CL,[SI] ; CX = # of bytes to copy
34:        inc        SI       ; SI = input string first char.
35:        cld                 ; forward direction for copy
36:        rep        movsb
37:        mov        BYTE PTR [DI],0   ; append NULL character
38:        pop        ES       ; restore registers
39:        pop        DI
40:        pop        SI
41:        pop        DX
42:        ret
43:  proc_GetStr  ENDP
```

20.4.2 BIOS Keyboard Services

BIOS provides keyboard service routines under `int 16H`. Here we describe three common routines that are useful in accessing the keyboard. As with the DOS functions, the AH register should contain the function code before executing an interrupt of type 16H. One difference between DOS and BIOS functions is that if you use DOS services, the keyboard input can be redirected.

Function 00H: *Read a character from the keyboard.*

```
Input:     AH  =  00H;
Returns:   if AL ≠ 0 then
             AL  =  ASCII code of the key entered,
             AH  =  Scan code of the key entered,
           if AL  =  0 then
             AH  =  Scan code of the extended key entered.
```

This BIOS function can be used to read a character from the keyboard. If the keyboard buffer is empty, it waits for a character to be entered. As with the DOS keyboard function, the value returned in AL determines if the key represents an ASCII character or an extended key character. In both cases, the scan code is placed in the AH register and the ASCII and scan codes are removed from the keyboard buffer.

A Problem

The problem is that 00H represents NULL in ASCII. To solve this problem, returning the NULL key ASCII code (00H) is interpreted as reading an extended key. Then how will you recognize the NULL key? This is a special case and the only ASCII key that is returned as an extended key character. Thus, if AL = 0 and AH = 3 (the scan code for the @ key), then the contents of AL should be treated as the ASCII code for the NULL key.

Here is a simple routine to read a character from the keyboard, which is a modified version of the routine given on page 834:

```
Read the next character code using function 00H of int 16H
if (AL ≠ 0)
then
    AL = ASCII code of the key entered
else      {AL = 0 which implies extended key with one exception}
    if  (AH = 3)
    then
        AL = ASCII code of NULL
    else
        AH = scan code of an extended key
    end if
end if
```

Table 20.1 Bit assignment for shift and toggle keys

Bit number	Key assignment
0	`Right shift` key depressed
1	`Left shift` key depressed
2	`Control` key depressed
3	`Alt` key depressed
4	`Scroll lock` switch is on
5	`Number lock` switch is on
6	`Caps lock` switch is on
7	`Ins lock` switch is on

Function 01H: *Check keyboard buffer.*

Input: AH = 01H;
Returns: ZF = 1 if the keyboard buffer is empty;
ZF = 0 if there is at least one character available. In this case, the ASCII and scan codes are placed in the AL and AH registers as in function 00H. The codes, however, are not removed from the keyboard buffer.

This function can be used to take a peek at the next character without actually removing it from the buffer. It provides similar functionality to the DOS function 0BH (see page 834). Unlike the DOS function, the zero flag (ZF) is used to indicate whether the keyboard buffer is empty. Since it does not actually remove the key codes from the keyboard buffer, it allows us to look ahead at the next character.

Function 02H: *Check keyboard status.*

Input: AH = 02H;
Returns: AL = status of the shift and toggle keys.

The bit assignment is shown in Table 20.1. In this table, a bit with a value of 1 indicates the presence of the condition.

This function can be used to test the status of the four shift keys (right shift, left shift, ctrl, and alt) and four toggle switches (scroll lock, number lock, caps lock, and ins).

Example 20.2 *A BIOS keyboard example.*
In this example, we write a program that reads a character string from the keyboard and displays the input string along with its length. The string input is terminated either by pressing both shift

keys simultaneously, or by entering 80 characters, whichever occurs first. This is a strange termination condition (requiring the depression of both shift keys), but it is useful to illustrate the flexibility of the BIOS keyboard functions.

As the `main` procedure is straightforward to understand, we focus on the mechanics of the `read_string` procedure. On first attempt, we might write this procedure as follows:

```
read_string()
    get maximum string length str_len and
        string pointer from the stack
    repeat
        read keyboard status (use int 16H function 2)
        if (both shift keys depressed)
        then
            goto end_read
        else
            read keyboard key (use int 16H function 0)
            copy the character into the string buffer
            increment buffer pointer
            display the character on the screen
        end if
    until (string length = str_len)
end_read:
    append NULL character to string input
    find and return the string length
    return
end read_string
```

Unfortunately, this procedure will not work properly. In most cases, the only way to terminate the string input is by actually entering 80 characters. Pressing both shift keys will not terminate the string input unless a key is entered while holding both shift keys down. Why? The problem with the above code is that the `repeat` loop briefly checks the keyboard status (takes only a few microseconds). It then waits for you to type a key. When you enter a key, it reads the ASCII code of the key and initiates another `repeat` loop iteration. Thus, every time you enter a key, the program checks the status of the two shift keys within a few microseconds after a key has been typed. Therefore, `read_string` will almost never detect the condition that both shift keys are depressed (with the exception noted).

To correct this problem, we have to modify the procedure as follows:

```
read_string()
    get maximum string length str_len and
        string pointer from the stack
read_loop:
```

```
        repeat
            read keyboard status (use int 16H function 2)
            if (both shift keys depressed)
            then
                goto end_read
            else
                check keyboard buffer status (use int 16H function 1)
                if (a key is available)
                then
                    read keyboard key (use int 16H function 0)
                    copy the character into the string buffer
                    increment buffer pointer
                    display the character on screen
                end if
            end if
        until (string length = str_len)
    end_read:
        append NULL character to string input
        find and return the string length
        return
    end read_string
```

With the modification, the procedure's `repeat` loop spends most of the time performing the following two actions:

1. Read keyboard status (using `int 16H` function 2).
2. Check if a key has been pressed (using `int 16H` function 1).

Since function 1 does not wait for a key to be entered, the procedure properly detects the string termination condition (i.e., depression of both shift keys simultaneously).

Program 20.2 `funnystr.asm` demonstrates the use of BIOS functions to read a string from the keyboard

```
 1:  COMMENT |       A string read program       FUNNYSTR.ASM
 2:          Objective: To demonstrate the use of BIOS keyboard
 3:                     functions 0, 1, and 2.
 4:                Input: Prompts for a string.
 5:  |            Output: Displays the input string and its length.
 6:
 7:  STR_LENGTH  EQU  81
 8:  .MODEL SMALL
 9:  .STACK   100H
10:  .DATA
```

```
11:  string      DB  STR_LENGTH DUP (?)
12:  prompt_msg  DB  'Please enter a string (< 81 chars): ',0
13:  string_msg  DB  'The string entered is ',0
14:  length_msg  DB  ' with a length of ',0
15:  end_msg     DB  ' characters.',0
16:
17:  .CODE
18:  INCLUDE  io.mac
19:  main    PROC
20:          .STARTUP
21:          PutStr  prompt_msg
22:          mov     AX,STR_LENGTH-1
23:          push    AX              ; push max. string length
24:          mov     AX,OFFSET string
25:          push    AX              ; and string pointer parameters
26:          call    read_string     ; to call read_string procedure
27:          nwln
28:          PutStr  string_msg
29:          PutStr  string
30:          PutStr  length_msg
31:          PutInt  AX
32:          PutStr  end_msg
33:          nwln
34:          .EXIT
35:  main    ENDP
36:  ;-----------------------------------------------------------
37:  ; String read procedure using BIOS int 16H. Receives string
38:  ; pointer and the length via the stack. Length of the string
39:  ; is returned in AX.
40:  ;-----------------------------------------------------------
41:  read_string     PROC
42:          push    BP
43:          mov     BP,SP
44:          push    BX
45:          push    CX
46:          mov     CX,[BP+6]    ; CX = length
47:          mov     BX,[BP+4]    ; BX = string pointer
48:  read_loop:
49:          mov     AH,2         ; read keyboard status
50:          int     16H          ; status returned in AL
51:          and     AL,3         ; mask off most significant 6 bits
52:          cmp     AL,3         ; if equal both shift keys depressed
53:          jz      end_read
54:          mov     AH,1         ; otherwise, see if a key has been
55:          int     16H          ; struck
```

```
56:          jnz     read_key      ; if so, read the key
57:          jmp     read_loop
58:  read_key:
59:          mov     AH,0          ; read the next key from keyboard
60:          int     16H           ; key returned in AL
61:          mov     [BX],AL       ; copy to buffer and increment
62:          inc     BX            ;  buffer pointer
63:          PutCh   AL            ; display the character
64:          loop    read_loop
65:  end_read:
66:          mov     BYTE PTR[BX],0  ; append NULL
67:          sub     BX,[BP+4]     ; find the input string length
68:          mov     AX,BX         ; return string length in AX
69:          pop     CX
70:          pop     BX
71:          pop     BP
72:          ret     4
73:  read_string     ENDP
74:          END     main
```

20.5 Pentium Exceptions

Pentium exceptions are classified into *faults*, *traps*, and *aborts* depending on the way they are reported and whether the instruction that is interrupted is restarted. Faults and traps are reported at instruction boundaries. Faults use the boundary before the instruction during which the exception was detected. When a fault occurs, the system state is restored to the state before the current instruction so that the instruction can be restarted. The divide error, for instance, is a fault detected during the `div` or `idiv` instruction. The processor, therefore, restores the state to correspond to the one before the divide instruction that caused the fault. Furthermore, the instruction pointer is adjusted to point to the divide instruction so that, after returning from the exception handler, the divide instruction is reexecuted.

Another example of a fault is the *segment-not-present* fault. This exception is caused by a reference to data in a segment that is not in memory. Then, the exception handler must load the missing segment from the disk and resume program execution starting with the instruction that caused the exception. In this example, it clearly makes sense to restart the instruction that caused the exception.

Traps, on the other hand, are reported at the instruction boundary immediately following the instruction during which the exception was detected. For instance, the overflow exception (interrupt 4) is a trap. Therefore, no instruction restart is done. User-defined interrupts are also examples of traps.

Aborts are exceptions that report severe errors. Examples include hardware errors and inconsistent values in system tables.

There are several interrupts predefined by the Pentium. These are called *dedicated* inter-

Table 20.2 The first five Pentium dedicated interrupts

Interrupt type	Purpose
0	Divide error
1	Single-step
2	Nonmaskable interrupt (NMI)
3	Breakpoint
4	Overflow

rupts. These include the first five interrupts as shown in Table 20.2. The NMI is a hardware interrupt that is discussed in Section 20.6. A brief description of the remaining four interrupts is given here.

Divide Error Interrupt: The Pentium generates a type 0 interrupt whenever executing a divide instruction—either `div` (divide) or `idiv` (integer divide)—results in a quotient that is larger than the destination specified. The default interrupt handler displays a *divide overflow* message and terminates the program.

Single-Step Interrupt: Single-stepping is a useful debugging tool to observe the behavior of a program instruction by instruction. To start single-stepping, the trap flag (TF) bit in the flags register should be set (i.e., TF = 1). When TF is set, the CPU automatically generates a type 1 interrupt after executing each instruction. Some exceptions do exist, but we do not bother about them here.

The interrupt handler for a type 1 interrupt can be used to display relevant information about the state of the program. For example, the contents of all registers could be displayed. Shortly we present an example program that initiates and stops single-stepping (see Example 20.3).

To end single-stepping, the TF should be cleared. The Pentium, however, does not have any instructions to directly manipulate the TF bit. Instead, we have to resort to an indirect means. This is illustrated in the next example.

Breakpoint Interrupt: If you have used a debugger, which you should have by now, you already know the usefulness of inserting breakpoints while debugging a program. The type 3 interrupt is dedicated to the breakpoint processing. This type of interrupt can be generated by using the special single-byte form of `int 3` (opcode CCH). Using the `int 3` instruction automatically causes the assembler to encode the instruction into the single-byte version. Note that the standard encoding for the `int` instruction is two bytes long.

Inserting a breakpoint in a program involves replacing the program code byte by CCH while saving the program byte for later restoration to remove the breakpoint. The standard 2-byte version of `int 3` can cause problems in certain situations, as there are instructions that require only a single byte to encode.

Overflow Interrupt: The type 4 interrupt is dedicated to handling overflow conditions. There are two ways by which a type 4 interrupt can be generated: either by `int 4` or by `into`. Like the breakpoint interrupt, `into` requires only one byte to encode, as it does not require the specification of the interrupt type number as part of the instruction. Unlike `int 4`, which unconditionally generates a type 4 interrupt, `into` generates a type 4 interrupt only if the overflow flag is set. We do not normally use `into`, as the overflow condition is usually detected and processed by using the conditional jump instructions `jo` and `jno`.

Example 20.3 *A single-step example.*
As an example of an exception, we write an interrupt handler to single-step a piece of code (let us call it *single-step code*). During single-stepping, we display the contents of the AX and BX registers after the execution of each instruction in the single-step code. The objectives in writing this program are to demonstrate how interrupt handlers can be defined and installed and to show how the TF can be manipulated.

To put the CPU in the single-step mode, we have to set the TF. Since there are no instructions to manipulate TF directly, we have to use an indirect means: first we use `pushf` to push flags onto the stack, then manipulate the TF bit, and finally, use `popf` to restore the modified flags word from the stack to the flags register. The code on lines 42 to 46 of Program 20.3 essentially performs this manipulation to set TF. The TF bit can be set by

```
or    AX,100H
```

Of course, we can also manipulate this bit directly on the stack itself. To clear the TF bit, we follow the same procedure and instead of `or`ing, we use

```
and    AX,0FEFFH
```

on line 57. We use two services of `int 21H` to get and set interrupt vectors.

Function 35H: *Get interrupt vector.*

Inputs:	AH	=	35H,
	AL	=	Interrupt type number;
Returns:	ES:BX	=	Address of the specified interrupt handler.

Function 25H: *Set interrupt vector.*

Inputs:	AH	= 25H,
	AL	= Interrupt type number,
	DS:DX	= Address of the interrupt handler;
Returns:	Nothing.	

The remainder of the code is straightforward:
Lines 27 to 30: We use function 35H to get the current vector value for `int 1`. This vector value is restored before exiting the program.
Lines 33 to 39: The vector of our interrupt handler is installed by using function 25H.
Lines 62 to 68: The original `int 1` vector is restored using function 25H.

A Note: It is not necessary to restore the old vector before we exit the program. DOS does this for us as part of the function 4CH call. Thus, it is not necessary to read and store the old interrupt vector value in our program. However, DOS does not restore interrupt vectors for all interrupts. As an example, it does not restore the original vector for `int 09H`. Thus, it is good practice to save and restore interrupt vectors within your program. We follow this practice in our examples.

Program 20.3 An example to illustrate the installation of a user-defined ISR

```
 1:  TITLE   Single-step program      STEPINTR.ASM
 2:  COMMENT |
 3:          Objective: To demonstrate how ISRs can be defined
 4:                     and installed.
 5:              Input: None.
 6:             Output: Displays AX and BX values for
 7:  |                  the single-step code.
 8:
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
12:  old_offset  DW  ?    ; for old ISR offset
13:  old_seg     DW  ?    ;   and segment values
14:  start_msg   DB  'Starts single-stepping process.',0
15:  AXequ       DB  'AX = ',0
16:  BXequ       DB  ' BX = ',0
17:
18:  .CODE
19:  INCLUDE io.mac
20:
21:  main    PROC
22:          .STARTUP
```

```
23:         PutStr  start_msg
24:         nwln
25:
26:         ; get current interrupt vector for int 1H
27:         mov     AX,3501H        ; AH = 35H and AL = 01H
28:         int     21H             ; returns the offset in BX
29:         mov     old_offset,BX   ;   and the segment in ES
30:         mov     old_seg,ES
31:
32:         ;set up interrupt vector to our ISR
33:         push    DS              ; DS is used by function 25H
34:         mov     AX,CS           ; copy current segment to DS
35:         mov     DS,AX
36:         mov     DX,OFFSET sstep_ISR  ; ISR offset in DX
37:         mov     AX,2501H        ; AH = 25H and AL = 1H
38:         int     21H
39:         pop     DS              ; restore DS
40:
41:         ; set trap flag to start single-stepping
42:         pushf
43:         pop     AX              ; copy flags into AX
44:         or      AX,100H         ; set trap flag bit (TF = 1)
45:         push    AX              ; copy modified flag bits
46:         popf                    ;   back to flags register
47:
48:         ; from now on int 1 is generated after executing
49:         ;  each instruction. Some test instructions follow.
50:         mov     AX,100
51:         mov     BX,20
52:         add     AX,BX
53:
54:         ; clear trap flag to end single-stepping
55:         pushf
56:         pop     AX              ; copy flags into AX
57:         and     AX,0FEFFH       ; clear trap flag bit (TF = 0)
58:         push    AX              ; copy modified flag bits
59:         popf                    ;   back to flags register
60:
61:         ; restore the original ISR
62:         mov     DX,old_offset
63:         push    DS
64:         mov     AX,old_seg
65:         mov     DS,AX
66:         mov     AX,2501H
67:         int     21H
```

```
68:          pop     DS
69:
70:          .EXIT
71:  main    ENDP
72:  ;---------------------------------------------------------
73:  ;Single-step interrupt service routine replaces int 01H.
74:  ;---------------------------------------------------------
75:  sstep_ISR  PROC
76:          sti                 ; enable interrupt
77:          PutStr  AXequ       ; display AX contents
78:          PutInt  AX
79:          PutStr  BXequ       ; display BX contents
80:          PutInt  BX
81:          nwln
82:          iret
83:  sstep_ISR  ENDP
84:          END   main
```

20.6 Pentium Hardware Interrupts

We have seen how interrupts can be caused by the software instruction `int`. Since these instructions are placed in a program, such software interrupts are called *synchronous* events. Hardware interrupts, on the other hand, are of hardware origin and *asynchronous* in nature. These interrupts are typically used by I/O devices to alert the CPU that they require its attention.

Hardware interrupts can be further divided into either *maskable* or *nonmaskable* interrupts (see Figure 20.1 on page 828). A nonmaskable interrupt can be triggered by applying an electrical signal on the NMI pin of the Pentium. This interrupt is called nonmaskable because the CPU always responds to this signal. In other words, this interrupt cannot be disabled under program control. The NMI causes a type 2 interrupt.

Most hardware interrupts are of maskable type. To generate a hardware interrupt, an electrical signal should be applied to the INTR (interrupt request) input of the Pentium. The Pentium recognizes the INTR interrupt only if the interrupt enable flag (IF) bit is set to 1 (see Figure 7.4 on page 259). Thus, these interrupts can be masked (i.e., disabled) by clearing the IF bit. Note that we can use `sti` and `cli`, respectively, to set and clear this bit in the flags register.

20.6.1 How Does the CPU Know the Interrupt Type?

Recall that every interrupt should be identified by its type (a number between 0 and 255), which is used as an index into the interrupt vector table to obtain the corresponding interrupt handler address. This interrupt invocation procedure is common to all interrupts, whether caused by software or hardware.

In response to a hardware interrupt request on the INTR pin, the Pentium initiates an interrupt acknowledge sequence. As part of this sequence, it sends out an interrupt acknowledge

(INTA) signal, and the interrupting device is expected to place the interrupt type number on the data bus. This number is used to identify the interrupt type. In the Pentium, the interrupt type number is byte, which is placed on the lower eight data lines, to identify an interrupt out of the 256 interrupts.

20.6.2 How Can More Than One Device Interrupt?

From the above description, it is clear that all interrupt requests from external devices should be input via the INTR pin of the Pentium. Although it is straightforward to connect a single device, computers typically have more than one I/O device requesting interrupt service. For example, the keyboard, hard disk, and floppy disk all generate interrupts when they require the attention of the CPU.

When more than one device interrupts, we have to have a mechanism to prioritize these interrupts (in case they arrive simultaneously) and forward only one interrupt request at a time to the CPU while keeping the other interrupt requests pending their turn. This mechanism can be implemented by using a special chip, the Intel 8259 programmable interrupt controller, which is described next.

20.6.3 8259 Programmable Interrupt Controller

The Intel 8259 programmable interrupt controller (PIC) chip can be used to accommodate more than one interrupting device. The 8259 PIC can service interrupts from up to eight hardware devices. These interrupts are received on lines IRQ0 through IRQ7, as shown in Figure 20.4.

Internally, the 8259 has an 8-bit interrupt command register (ICR) and another 8-bit interrupt mask register (IMR). The ICR is used to program the 8259, and the IMR is used to enable or disable specific interrupt requests. The 8259 can be programmed to assign priorities to IRQ0 to IRQ7 requests in several ways. BIOS initializes the 8259 to assign fixed priorities: the default mode called the fully nested mode. In this mode, the incoming interrupt requests IRQ0 through IRQ7 are prioritized with the IRQ0 receiving the highest priority and the IRQ7 receiving the lowest priority.

Also part of this initialization is the assignment of interrupt type numbers. To do this, only the lowest type number need be specified. BIOS uses 08H as the lowest interrupt type (for the request coming on the IRQ0 line). The 8259 automatically assigns the next seven numbers to the remaining seven IRQ lines in increasing order, with IRQ7 generating an interrupt of type 0FH.

All communication between the CPU and the 8259 occurs via the data bus. The 8259 PIC is an 8-bit device requiring two ports for the ICR and IMR. These are mapped to the I/O address space, as shown in Table 20.3. Table 20.4 shows the mapping of the IRQ input to various devices in the system.

Note that the CPU recognizes external interrupt requests generated by the 8259 only if the IF flag is set. Thus, by clearing the IF flag, we can disable all eight external interrupts as a group. However, to selectively disable external interrupts, we have to use the IMR. Each bit

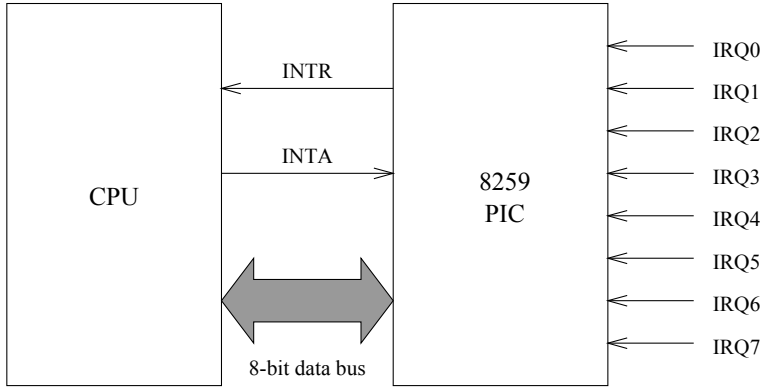


Figure 20.4 Intel 8259 programmable interrupt controller.

Table 20.3 8259 port address mapping

8259 register	Port address
ICR	20H
IMR	21H

in the IMR enables (if the bit is 0) or disables (if the bit is 1) its associated interrupt. Bit 0 is associated with IRQ0, bit 1 with IRQ1, and so on. For example, we can use the code

```
mov     AL,0FEH
out     21H,AL
```

to disable all external interrupts except the system timer interrupt on the IRQ0 line.

When several interrupt requests are received by the 8259, it serializes these requests according to their priority levels. For example, if a timer interrupt (IRQ0) and a keyboard interrupt (IRQ1) arrive simultaneously, the 8259 forwards the timer interrupt to the CPU, as it has a higher priority than the keyboard interrupt. Once the timer interrupt handler is completed, the 8259 forwards the keyboard interrupt to the CPU for processing. To facilitate this, the 8259 should know when an interrupt handler is completed. The end of an interrupt handler execution is signaled to the 8259 by writing 20H into the ICR. Thus, the code fragment

```
mov     AL,20H
out     20H,AL
```

can be used to indicate end-of-interrupt (EOI) to 8259. This code fragment appears before the `iret` instruction of an interrupt handler.

Table 20.4 Mapping of I/O devices to external interrupt levels

IRQ #	Interrupt type	Device
0	08H	System timer
1	09H	Keyboard
2	0AH	Used for expansion
3	0BH	Serial port (COM1)
4	0CH	Serial port (COM2)
5	0DH	Hard disk
6	0EH	Floppy disk
7	0FH	Printer

20.6.4 A Pentium Hardware Interrupt Example

We now illustrate how a hardware interrupt routine can be written. As an example, we write a type 9 interrupt routine to replace the BIOS supplied `int 09` routine to read input from the keyboard. In the last chapter, we looked at a similar example using the programmed I/O technique (see Section 19.4.1 on page 775).

Example 20.4 *A keyboard example to illustrate interrupt-driven I/O.*
Our objective is to write a replacement ISR for `int 09H`. A type 9 interrupt is generated via the IRQ1 line of the 8259 PIC by the keyboard every time a key is depressed or released.

The logic of the main procedure can be described as follows:

```
main()
    save the current int 9 vector
    install our keyboard ISR
    display "Interrupt handler installed" message
    repeat
        read_kb_key()
            {this procedure waits until a key is pressed
             and returns the ASCII code of the key in AL}
        if (key ≠ Esc key)
        then
            if (key = carriage return key)
            then
                display newline
            else
                display the key
```

```
            end if
        else
            goto done   {If Esc key, we are done}
        end if
    until (FALSE)
done:
    restore the original int 09H vector
    return to DOS
end main
```

The `read_kb_key` procedure waits until a value is deposited in the keyboard buffer `keyboard_data`. The pseudocode is as follows:

```
read_kb_key()
    while (keyboard_data = –1)
    end while
    AL := keyboard_data
    keyboard_data := –1
    return
end read_kb_key
```

The keyboard interrupt handler `kbrd_ISR` is invoked whenever a key is pressed or released. As described in Section 19.3, the scan code of the key can be read from PA0 to PA6, and the key state can be read from PA7. PA7 is 0 if the key is depressed; PA7 is 1 if the key is released. After reading the key scan code in Program 20.4 (lines 107 and 108), the keyboard should be acknowledged. This is done by momentarily setting and clearing the PB7 bit (lines 111 to 116). If the key is the left shift or right shift key, bit 0 of `keyboard_flag` is updated. If it is a normal key, its ASCII code is obtained. The code on lines 154 and 155 will send an end-of-interrupt to the 8259 to indicate that the interrupt service is completed. The pseudocode of the ISR is given below:

```
kbrd_ISR()
    read key scan code from KB_DATA (port 60H)
    set PB7 bit to acknowledge using KB_CTRL (port 61H)
    clear PB7 to reset acknowledge
    process the key
    send end-of-interrupt (EOI) to 8259
    iret
end kbrd_ISR
```

Program 20.4 A keyboard ISR to replace BIOS `int 09H` keyboard ISR

```
 1:  TITLE   Keyboard interrupt service program   KEYBOARD.ASM
 2:  COMMENT |
 3:          Objective: To demonstrate how the keyboard works.
 4:              Input: Key strokes from the keyboard. Only left
 5:                     and right shift keys are recognized.
 6:                     ESC key restores the original keyboard ISR
 7:                     and terminates the program.
 8:  |            Output: Displays the key on the screen.
 9:
10:  ESC_KEY      EQU  1BH   ; ASCII code for ESC key
11:  CR           EQU  0DH   ; ASCII code for carriage return
12:  KB_DATA      EQU  60H   ; 8255 port PA
13:  KB_CTRL      EQU  61H   ; 8255 port PB
14:  LEFT_SHIFT   EQU  2AH   ; left shift scan code
15:  RIGHT_SHIFT  EQU  36H   ; right shift scan code
16:  EOI          EQU  20H   ; end-of-interrupt byte for 8259 PIC
17:  PIC_CMD_PORT EQU  20H   ; 8259 PIC command port
18:
19:  .MODEL SMALL
20:  .STACK 100H
21:  .DATA
22:  install_msg    DB  'New keyboard ISR installed.',0
23:  keyboard_data  DB  -1    ; keyboard buffer
24:  keyboard_flag  DB  0     ; keyboard shift status
25:  old_offset     DW  ?     ; storage for old int 09H vector
26:  old_segment    DW  ?
27:  ; lowercase scan code to ASCII conversion table.
28:  ; ASCII code 0 is used for unnecessary scan codes.
29:  lcase_table    DB  01BH,'1234567890-=',08H,09H
30:                 DB  'qwertyuiop[]',CR,0
31:                 DB  'asdfghjkl;',27H,60H,0,'\'
32:                 DB  'zxcvbnm,./',0,'*',0,' ',0
33:                 DB  0,0,0,0,0,0,0,0,0,0
34:                 DB  0,0,0,0,0,0,0,0,0,0
35:                 DB  0,0,0,0,0,0,0,0,0,0
36:  ; uppercase scan code to ASCII conversion table.
37:  ucase_table    DB  01BH,'!@#$%^&*()_+',08H,09H
38:                 DB  'QWERTYUIOP{}',0DH,0
39:                 DB  'ASDFGHJKL:','"','~',0,'|'
40:                 DB  'ZXCVBNM<>?',0,'*',0,' '
41:                 DB  0,0,0,0,0,0,0,0,0,0
42:                 DB  0,0,0,0,0,0,0,0,0,0,0,0,0,0
43:  .CODE
44:  INCLUDE io.mac
```

```
45:
46:  main    PROC
47:          .STARTUP
48:          PutStr  install_msg
49:          nwln
50:
51:          ; save int 09H vector for later restoration
52:          mov     AX,3509H     ; AH = 35H and AL = 09H
53:          int     21H          ; DOS function 35H returns
54:          mov     old_offset,BX   ; offset in BX and
55:          mov     old_segment,ES  ; segment in ES
56:
57:          ;set up interrupt vector to our keyboard ISR
58:          push    DS           ; DS is used by function 25H
59:          mov     AX,CS        ; copy current segment to DS
60:          mov     DS,AX
61:          mov     DX,OFFSET kbrd_ISR  ; ISR offset in DX
62:          mov     AX,2509H     ; AH = 25H and AL = 09H
63:          int     21H
64:          pop     DS           ; restore DS
65:
66:  repeat:
67:          call    read_kb_key  ; read a key
68:          cmp     AL,ESC_KEY   ; if ESC key
69:          je      done         ; then done
70:          cmp     AL,CR        ; if carriage return
71:          je      newline      ; then display new line
72:          PutCh   AL           ; else display character
73:          jmp     repeat
74:  newline:
75:          nwln
76:          jmp     repeat
77:  done:
78:          ; restore original keyboard interrupt int 09H vector
79:          mov     DX,old_offset
80:          push    DS
81:          mov     AX,old_segment
82:          mov     DS,AX
83:          mov     AX,2509H
84:          int     21H
85:          pop     DS
86:
87:          .EXIT
88:  main    ENDP
89:  ;---------------------------------------------------------
```

```
 90:  ;This procedure waits until a valid key is entered at the
 91:  ; keyboard. The ASCII value of the key is returned in AL.
 92:  ;------------------------------------------------------------
 93:  read_kb_key  PROC
 94:          cmp     keyboard_data,-1  ; -1 is an invalid entry
 95:          je      read_kb_key
 96:          mov     AL,keyboard_data
 97:          mov     keyboard_data,-1
 98:          ret
 99:  read_kb_key  ENDP
100:  ;------------------------------------------------------------
101:  ;This keyboard ISR replaces the original int 09H ISR.
102:  ;------------------------------------------------------------
103:  kbrd_ISR  PROC
104:          sti                   ; enable interrupt
105:          push    AX            ; save registers used by ISR
106:          push    BX
107:          in      AL,KB_DATA    ; read keyboard scan code and the
108:          mov     BL,AL         ;  key status (down or released)
109:          ; send keyboard acknowledge signal by momentarily
110:          ;  setting and clearing PB7 bit
111:          in      AL,KB_CTRL
112:          mov     AH,AL
113:          or      AL,80H
114:          out     KB_CTRL,AL    ; set PB7 bit
115:          xchg    AL,AH
116:          out     KB_CTRL,AL    ; clear PB7 bit
117:
118:          mov     AL,BL         ; AL = scan code + key status
119:          and     BL,7FH        ; isolate scan code
120:          cmp     BL,LEFT_SHIFT   ; left or right shift key
121:          je      left_shift_key  ;  changed status?
122:          cmp     BL,RIGHT_SHIFT
123:          je      right_shift_key
124:          test    AL,80H          ; if not, check status bit
125:          jnz     EOI_to_8259     ; if key released, do nothing
126:          mov     AH,keyboard_flag ; AH = shift key status
127:          and     AH,1            ; AH = 1 if left/right shift is ON
128:          jnz     shift_key_on
129:          ; no shift key is pressed
130:          mov     BX,OFFSET lcase_table ; shift OFF, use lowercase
131:          jmp     SHORT get_ASCII       ;  conversion table
132:  shift_key_on:
133:          mov     BX,OFFSET ucase_table ; shift key ON, use uppercase
134:  get_ASCII:                            ;  conversion table
```

```
135:          dec     AL           ; index is one less than scan code
136:          xlat
137:          cmp     AL,0         ; ASCII code of 0 => uninterested key
138:          je      EOI_to_8259
139:          mov     keyboard_data,AL  ; save ASCII code in keyboard buffer
140:          jmp     SHORT EOI_to_8259
141:
142:  left_shift_key:
143:  right_shift_key:
144:          test    AL,80H       ; test key status bit (0=down, 1=up)
145:          jnz     shift_off
146:  shift_on:
147:          or      keyboard_flag,1     ; shift bit (i.e., LSB) := 1
148:          jmp     SHORT EOI_to_8259
149:  shift_off:
150:          and     keyboard_flag,0FEH  ; shift bit (i.e., LSB) := 0
151:          jmp     SHORT EOI_to_8259
152:
153:  EOI_to_8259:
154:          mov     AL,EOI              ; send EOI to 8259 PIC
155:          out     PIC_CMD_PORT,AL     ; indicating end of ISR
156:          pop     BX                  ; restore registers
157:          pop     AX
158:          iret
159:  kbrd_ISR  ENDP
160:          END   main
```

20.7 Interrupt Processing in the PowerPC

The Pentium's interrupt processing involves the stack. For example, the flags register contents and the return address value are pushed onto the stack as part of transferring control to the interrupt handler. As the PowerPC and MIPS are RISC processors, they use registers to store the return address as well as the system state. We describe the PowerPC interrupt processing in this section. MIPS processor details are presented in the next section.

The PowerPC maintains the system state information in the machine state register (MSR). Each MSR bit indicates a specific state of the system's context. Here is a sample of the kind of state information contained in the MSR.

- *POW:* The power management enable bit indicates whether the system is operating in normal mode (POW = 0) or reduced power mode (POW = 1).
- *EE:* The external interrupt enable bit is similar to the IF flag in the Pentium. If this bit is zero, the processor delays recognition of external interrupts. The processor responds to the external interrupts when this bit is one.

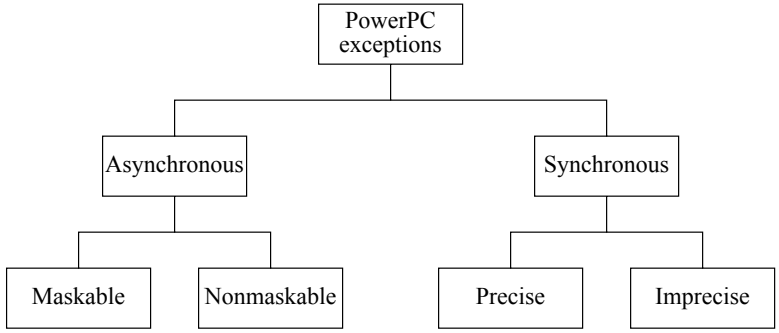


Figure 20.5 PowerPC exception classification.

- *PR:* If the privilege level bit is zero, the processor can execute both user- and supervisor-level instructions. If it is 1, only the user-level instructions are executed.
- *SE:* If the single-step enable bit is zero, the processor executes instructions normally. If this bit is 1, it single-steps as in the Pentium.
- *IP:* This exception prefix bit indicates whether an exception vector is prepended with Fs or 0s. In a sense, this bit indicates the base address of the vector table. If this bit is 0, exceptions are vectored to physical addresses $000nnnnn$H, where $nnnnn$ is the offset of the exception vector. On the other hand, if IP = 1, the physical address FFF$nnnnn$ is used. We explain shortly the offset associated with various exceptions.
- *IR:* This bit indicates whether the instruction address translation is disabled (0) or enabled (1).
- *DR:* This bit indicates whether the data address translation is disabled (0) or enabled (1).
- *LE:* This bit indicates whether the processor runs in little-endian (1) or big-endian mode (0). Recall that the PowerPC supports both endian modes. This bit can be used to dynamically change the mode.

The PowerPC exception classification is shown in Figure 20.5. Asynchronous exceptions are caused by events external to the processor, and synchronous interrupts are caused by instructions. To relate to our discussion of the Pentium interrupt classification, asynchronous exceptions are similar to hardware interrupts and synchronous exceptions are similar to software interrupts. As in the Pentium, asynchronous exceptions can be divided into maskable and nonmaskable types.

Synchronous exceptions are either precise or imprecise. Normally, we expect an exception to be associated with the instruction that caused it. In pipelined computers, establishing this association is not straightforward. Thus, we talk about precise and imprecise exceptions. A *precise exception* is an exception that is associated with the instruction that caused it. *Imprecise exceptions*, however, do not make this association. The PowerPC uses imprecise exceptions for floating-point instructions. All other instruction-caused exceptions are precise exceptions.

PowerPC exception processing is similar to that of the Pentium. Recall that Pentium interrupt processing involves the following four main actions:

- Push the machine state information (i.e., flags register) onto the stack;
- Disable further interrupts;
- Push the return address onto the stack;
- Load the interrupt handler address.

Instead of using the stack, the PowerPC uses two 32-bit save/restore registers—SRR0 and SRR1—to save the return address and the machine state. The SRR0 register is used to save the return address and SRR1 is used to save the machine state. Machine state information is copied from the MSR register. Note that, except for the POW bit, all the other bits of MSR listed before are stored in SRR1.

Each exception type has a fixed offset value, just as the table index value in the Pentium does. However, the Pentium uses indirect addressing to transfer control to the interrupt handler. In the PowerPC, exception offsets are used directly. You can place a jump instruction at that location to transfer control to the actual interrupt handler. The offset of each interrupt is separated by 256 bytes, as shown in Table 20.5. Thus, a jump is needed only if the interrupt handler is longer. The area 01000 to 02FFFH is reserved for implementation-specific exception vectors.

The IP bit in MSR specifies whether the leading hex digits are Fs or 0s. For example, external interrupt uses the vector offset 00000500H (if IP = 0) or FFF00500 (if IP = 1). In most systems, IP is set to 1 during system initialization and cleared to 0 after completing the initialization process.

Returning from an exception handler, again, is very similar to the Pentium's return mechanism. The PowerPC uses the `rfi` (return from interrupt) instruction to return control. The `rfi` instruction copies the contents of SRR1 into MSR and transfers control to the instruction at the address in SRR0.

20.8 Interrupt Processing in the MIPS

As mentioned in Section 20.1, the MIPS does not use the vectored interrupt mechanism. When an interrupt occurs, the processor suspends processing instructions in the normal flow and enters the kernel mode. It then disables interrupts and transfers control to a handler located at a fixed address. The handler saves the context of the processor including the program counter, the current operating mode (user or supervisor), and status of the interrupts (enabled or disabled). As in the PowerPC, the return address is stored in a register called the EPC (exception program counter). This information is used to restore the context after returning from the handler. As you can see, this process looks similar to that of the PowerPC.

The registers needed to process interrupts are not part of the main processor. Instead, these registers are located in coprocessor 0 (CP0). Register 14 of coprocessor 0 is used as the EPC. The contents of the EPC register can be copied into processor registers using the `mfc0` (move from coprocessor 0) instruction.

Table 20.5 PowerPC exceptions ("Offset" values in hex)

Exception type	Offset	Description
System reset	00100	System reset exception (similar to Reset in the Pentium)
Machine check	00200	An external interrupt caused by bus parity error, access to invalid physical address, etc.
DSI	00300	Invalid data memory access exception (e.g., memory protection violation, address cannot be translated)
ISI	00400	Invalid instruction memory access exception (e.g., instruction fetch from a no-execute segment, instruction fetch violates memory protection)
External interrupt	00500	External interrupt as in Pentium
Alignment	00600	Alignment exception (e.g., operand is not aligned)
Program	00700	Program exception (e.g., illegal instruction, privileged instruction execution is attempted in user-level)
System call	00C00	System call exception occurs when a system call (SC) instruction is executed (similar to the `int` instruction of the Pentium)
Trace	00D00	Trace exception (for debugging purposes)

Since the MIPS processor does not use vectored interrupts, it uses one of the coprocessor registers to provide the interrupt type information. Register 13 of the coprocessor is used as the Cause register. The Cause register has a five-bit field to identify the interrupt type. Table 20.6 shows some of the MIPS interrupt types.

The Cause register also contains information on pending interrupts. There are eight bits to indicate the pending interrupts. Out of these, two bits are used for software interrupts.

The Status register (register 12 of the coprocessor) has an eight-bit interrupt mask field. These eight bits control the interrupt enable and disable status of the eight interrupt conditions in the Cause register. The Status register also has a reverse-endian bit that can be used to invert the endien. Note that the processor is configured as little- or big-endian at system reset.

On external interrupts and exceptions, the MIPS processor jumps to a location at address 80000180H. However, not all exceptions use this handler. For example, reset, soft reset, and nonmaskable interrupt cause the MIPS processor to jump to location BFC00000H. The processor registers `$k0` and `$k1` are reserved for OS kernel use. Typically, the Cause and EPC registers of coprocessor 0 are loaded into these registers, as shown below:

Table 20.6 Some example MIPS exceptions

Type number	Description
0	External interrupt
8	Syscall exception
9	Breakpoint exception
12	Arithmetic overflow exception
13	Trace exception

```
mfc0    $k0,$13     # copy Cause register into $k0
mfc0    $k1,$14     # copy EPC register into $k1
```

The interrupt handler can then examine the Cause register contents (in `$k0`) to jump to an appropriate point in the operating system.

One difference from the other two processors is that the MIPS processor saves the interrupted instruction address as opposed to the one following it. As a result, we need to add an offset value of 4 before returning control from the handler, as shown below:

```
addiu    $k1,$k1,4
rfe
jr       $k1
```

This code also shows how an interrupt handler returns control. Unlike the Pentium's `iret` instruction, the MIPS return instruction only restores the context of the processor before the handler was invoked. To actually transfer control back, we have to use the `jr` instruction as in procedures.

20.9 Summary

Interrupts provide a mechanism to transfer control to an interrupt service routine. The mechanism is similar to that of a procedure call. However, although procedures can be invoked only by a procedure call in software, interrupts can be invoked by both hardware and software. In this chapter, we focused mainly on the Pentium processor interrupt mechanism.

Software interrupts are often used to support access to the system I/O devices. In PCs, both BIOS and DOS provide a high-level interface to the hardware via software interrupts. Hardware interrupts are used by I/O devices to interrupt the CPU to service their requests.

All Pentium interrupts, whether hardware- or software-initiated, are identified by an interrupt type number that is between 0 and 255. This interrupt number is used to access the interrupt vector table to get the associated interrupt vector. Hardware interrupts can be disabled by manipulating the interrupt flag using `sti` and `cli` instructions. Masking of individual external

interrupts can be done by manipulating the IMR register of the 8259 programmable interrupt controller.

In PCs, there are three ways an application program can access I/O devices. DOS and BIOS provide software interrupt support routines to access I/O devices. In the third method, an application program accesses the I/O devices directly via I/O ports. This involves low-level programming using `in` and `out` instructions. Such direct control of I/O devices requires detailed knowledge about the devices. We used several examples to illustrate how these methods are used to interact with I/O devices.

We briefly introduced the interrupt mechanisms of the PowerPC and MIPS processors. As does the Pentium, the PowerPC uses vectored interrupts. The MIPS processor, on the other hand, uses a Cause register to identify the cause of the interrupt.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Aborts
- Asynchronous exceptions
- Breakpoint interrupt
- Dedicated interrupts
- Exceptions
- Extended keys
- Faults
- Hardware interrupts
- Hardware interrupts—maskable
- Hardware interrupts—nonmaskable
- Imprecise exceptions
- Interrupt acknowledge
- Interrupt descriptor table
- Interrupt flag
- Interrupt handler
- Interrupt service routine
- Overflow interrupt
- Precise exceptions
- Programmable interrupt controller
- Single-step interrupt
- Software interrupts
- Synchronous exceptions
- System-defined interrupts
- Taxonomy of interrupts
- Trap flag
- Traps
- User-defined interrupts
- Vectored interrupts

20.10 Exercises

20–1 What is the difference between a procedure and an interrupt service routine?

20–2 What is the difference between the interrupt handling mechanisms of the Pentium and PowerPC?

20–3 In invoking an interrupt handler in the Pentium, the flags register is automatically saved on the stack. However, a procedure call does not automatically save the flags register. Explain the rationale for this difference.

20–4 How would you categorize the interrupts generated by the keyboard?

20–5 Describe how the extended keyboard keys are handled.

20–6 Explain how one can disable all maskable hardware interrupts efficiently.

20–7 Describe another way to disable all maskable hardware interrupts. (It doesn't have to be as efficient as that in the last exercise.)

20–8 Write a piece of code to disable all maskable hardware interrupts except the timer and keyboard interrupts. Refer to the interrupt table on page 850.

20–9 We have stated that the

```
into
```

instruction generates a type 4 interrupt. As you know, we can also generate this type of interrupt using the

```
int     4
```

instruction. What is the difference between these two instructions?

20–10 Suppose that the Pentium is currently executing the keyboard interrupt service routine shown below:

```
keyboard_ISR    PROC
        sti
         .
         .
        ISR body
         .
         .
        iret
keyboard_ISR    ENDP
```

Assume that, while in the middle of executing the keyboard ISR, a timer interrupt occurs. Describe the activities of the CPU until it completes processing the keyboard interrupt service routine.

20–11 What happens in the scenario described in the last question if the `sti` instruction is deleted from the keyboard interrupt handler?

20–12 Discuss the advantages and disadvantages of the three ways an application program can interact with I/O devices (see Figure 20.2).

20–13 Describe the actions taken (until the beginning of the execution of the interrupt handler) by the Pentium in response to `int 10H`. You can assume real mode of operation.

20–14 Is there any difference between how an interrupt handler is invoked if the interrupt is caused by the `int` instruction or hardware interrupt or exception?

20–15 What is the difference between the DOS keyboard function 0BH and the BIOS keyboard function 01H?

20–16 Discuss the tradeoffs associated with interrupts and polling (described in Section 19.4.1 on page 775).

20.11 Programming Exercises

20–P1 Write a divide error exception handler to replace the system-supplied one. This handler should display the message, "A divide error has occurred" and then replace the result with the maximum possible value. You can use registers for the dividend and divisor of the `div` instruction. Test your divide error interrupt handler by making the divisor zero. Also, experiment with the interrupt handler code so that you can verify that the `div` instruction is restarted because divide error is considered a fault. For example, if your interrupt handler does not change the value of the divisor (i.e., leave it as 0), your program will not terminate, as it repeatedly calls the divide error exception handler by restarting the divide instruction. After observing this behavior, modify the interrupt handler to change the divisor to a value other than 0 in order to proceed with your test program.

20–P2 The `into` instruction generates an overflow interrupt (interrupt 4) if the overflow flag is set. Overflow interrupt is a trap, and therefore the interrupt instruction is not restarted. Write an interrupt handler to replace the system-supplied one. Your interrupt handler should display the message, "An overflow has occurred" and then replace the result with zero. As a part of the exercise, test that `into` does not generate an interrupt unless the overflow flag is set.

20–P3 Convert `toupper.asm` given in Chapter 9 into an interrupt handler for interrupt 100. You can assume that DS:BX points to a null-terminated string. Write a simple program to test your interrupt handler.

APPENDICES

Appendix A

Computer Arithmetic

Objectives

- To present various number systems and conversions among them;
- To introduce signed and unsigned number representations;
- To discuss floating-point number representation;
- To describe IEEE 754 floating-point representation.

This appendix examines how data are represented internally in a computer system. Representing numbers is a two-step process: we have to select a number system to use, and then we have to decide how numbers in the selected number system can be represented for internal storage.

To facilitate our discussion, we first introduce several number systems, including the decimal system that we use in everyday life. Section A.2 discusses conversion of numbers among the number systems. We then proceed to discuss how integers—both unsigned (Section A.3) and signed (Section A.4)—and floating-point numbers (Section A.5) are represented. Character representation is discussed in the next appendix. We conclude with a summary.

A.1 Positional Number Systems

The number systems that we discuss here are based on positional number systems. The decimal number system that we are already familiar with is an example of a positional number system. In contrast, the Roman numeral system is not a positional number system.

Every positional number system has a *radix* or *base*, and an *alphabet*. The base is a positive number. For example, the decimal system is a base-10 system. The number of symbols in the alphabet is equal to the base of the number system. The alphabet of the decimal system is 0 through 9, a total of 10 symbols or digits.

In this appendix, we discuss four number systems that are relevant in the context of computer systems and programming. These are the *decimal* (base-10), *binary* (base-2), *octal* (base-8), and

hexadecimal (base-16) number systems. Our intention in including the familiar decimal system is to use it to explain some fundamental concepts of positional number systems.

Computers internally use the binary system. The remaining two number systems—octal and hexadecimal—are used mainly for convenience to write a binary number even though they are number systems on their own. We would have ended up using these number systems if we had 8 or 16 fingers instead of 10.

In a positional number system, a sequence of digits is used to represent a number. Each digit in this sequence should be a symbol in the alphabet. There is a weight associated with each position. If we count position numbers from right to left starting with zero, the weight of position n in a base b number system is b^n. For example, the number 579 in the decimal system is actually interpreted as

$$5 \times (10^2) + 7 \times (10^1) + 9 \times (10^0) .$$

(Of course, $10^0 = 1$.) In other words, 9 is in unit's place, 7 in 10's place, and 5 in 100's place. More generally, a number in the base b number system is written as

$$d_n d_{n-1} \ldots d_1 d_0 ,$$

where d_0 represents the least significant digit (LSD) and d_n represents the most significant digit (MSD). This sequence represents the value

$$d_n b^n + d_{n-1} b^{n-1} + \cdots + d_1 b^1 + d_0 b^0 . \tag{A.1}$$

Each digit d_i in the string can be in the range $0 \leq d_i \leq (b-1)$. When we are using a number system with $b \leq 10$, we use the first b decimal digits. For example, the binary system uses 0 and 1 as its alphabet. For number systems with $b > 10$, the initial letters of the English alphabet are used to represent digits greater than 9. For example, the alphabet of the hexadecimal system, whose base is 16, is 0 through 9 and A through F, a total of 16 symbols representing the digits of the hexadecimal system. We treat lowercase and uppercase letters used in a number system such as the hexadecimal system as equivalent.

The number of different values that can be represented using n digits in a base b system is b^n. Consequently, since we start counting from 0, the largest number that can be represented using n digits is $(b^n - 1)$. This number is written as

$$\underbrace{(b-1)(b-1)\ldots(b-1)(b-1)}_{\text{total of } n \text{ digits}} .$$

The minimum number of digits (i.e., the length of a number) required to represent X different values is given by $\lceil \log_b X \rceil$, where $\lceil\ \rceil$ represents the ceiling function. Note that $\lceil m \rceil$ represents the smallest integer that is greater than or equal to m.

A.1.1 Notation

The commonality in the alphabet of several number systems gives rise to confusion. For example, if we write 100 without specifying the number system in which it is expressed, different interpretations can lead to assigning different values, as shown below:

Number		Decimal value
100	$\xrightarrow{\text{binary}}$	4
100	$\xrightarrow{\text{decimal}}$	100
100	$\xrightarrow{\text{octal}}$	64
100	$\xrightarrow{\text{hexadecimal}}$	256

Thus, it is important to specify the number system (i.e., specify the base). We use the following notation in this text: A single letter—uppercase or lowercase—is appended to the number to specify the number system. We use D for decimal, B for binary, Q for octal, and H for hexadecimal number systems. When we write a number without one of these letters, the decimal system is the default number system. Using this notation, 10110111B is a binary number and 2BA9H is a hexadecimal number.

Decimal Number System

We use the decimal number system in everyday life. This is a base-10 system presumably because we have 10 fingers and toes to count. The alphabet consists of 10 symbols, digits 0 through 9.

Binary Number System

The binary system is a base-2 number system that is used by computers for internal representation. The alphabet consists of two digits, 0 and 1. Each binary digit is called a bit (standing for *b*inary dig*it*). Thus, 1021 is not a valid binary number.

In the binary system, using n bits, we can represent numbers from 0 through $(2^n - 1)$ for a total of 2^n different values. We need m bits to represent X different values, where

$$m = \lceil \log_2 X \rceil .$$

For example, 150 different values can be represented by using

$$\lceil \log_2 150 \rceil = \lceil 7.229 \rceil = 8 \text{ bits} .$$

In fact, using 8 bits, we can represent $2^8 = 256$ different values (i.e., from 0 through 255D).

Octal Number System

This is a base-8 number system with the alphabet consisting of digits 0 through 7. Thus, 181 is not a valid octal number. The octal numbers are often used to express binary numbers in a

compact way. For example, we need 8 bits to represent 256 different values. The same range of numbers can be represented in the octal system by using only

$$\lceil \log_8 256 \rceil = \lceil 2.667 \rceil = 3 \text{ digits}.$$

For example, the number 230Q is written in the binary system as 10011000B, which is difficult to read and errorprone. In general, we can reduce the length by a factor of 3. As we show in the next section, it is straightforward to go back to the binary equivalent, which is not the case with the decimal system.

Hexadecimal Number System

This is a base-16 number system. The alphabet consists of digits 0 through 9 and letters A through F. In this text, we use capital letters consistently, even though lowercase and uppercase letters can be used interchangeably. For example, FEED is a valid hexadecimal number, whereas GEFF is not.

The main use of this number system is to conveniently represent long binary numbers. The length of a binary number expressed in the hexadecimal system can be reduced by a factor of 4. Consider the previous example again. The binary number 10011000B can be represented as 98H. Debuggers, for example, display information—addresses, data, and so on—in hexadecimal representation.

A.2 Number Systems Conversion

When we are dealing with several number systems, there is often a need to convert numbers from one system to another. In the following, we look at how we can perform these conversions.

A.2.1 Conversion to Decimal

To convert a number expressed in the base-b system to the decimal system, we merely perform the arithmetic calculations of Equation A.1 given on page 866; that is, multiply each digit by its weight, and add the results. Note that these arithmetic calculations are done in the decimal system. Let's look at a few examples next.

Example A.1 *Conversion from binary to decimal.*
Convert the binary number 10100111B into its equivalent in the decimal system.

$$\begin{aligned} 10100111B &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 \\ &\quad + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 167D. \end{aligned}$$

Example A.2 *Conversion from octal to decimal.*
Convert the octal number 247Q into its equivalent in the decimal system.

$$\begin{aligned} 247Q &= 2 \cdot 8^2 + 4 \cdot 8^1 + 7 \cdot 8^0 \\ &= 167D. \end{aligned}$$

Example A.3 *Conversion from hexadecimal to decimal.*
Convert the hexadecimal number A7H into its equivalent in the decimal system.

$$\begin{aligned} A7H &= \mathrm{A} \cdot 16^1 + 7 \cdot 16^0 \\ &= 10 \cdot 16^1 + 7 \cdot 16^0 \\ &= 167D. \end{aligned}$$

We can obtain an iterative algorithm to convert a number to its decimal equivalent by observing that a number in base b can be written as

$$\begin{aligned} d_1 d_0 &= d_1 \times b^1 + d_0 \times b^0 \\ &= (d_1 \times b) + d_0, \\ d_2 d_1 d_0 &= d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 \\ &= ((d_2 \times b) + d_1)b + d_0, \\ d_3 d_2 d_1 d_0 &= d_3 \times b^3 + d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 \\ &= (((d_3 \times b) + d_2)b + d_1)b + d_0. \end{aligned}$$

The following algorithm summarizes this process.

Algorithm: Conversion from base b to the decimal system
Input: A number $d_{n-1}d_{n-2}\ldots d_1 d_0$ in base b
Output: Equivalent decimal number
Procedure: The digits of the input number are processed from left to right one digit at a time.

```
Result = 0;
for (i = n − 1 downto 0)
    Result = (Result × b ) + d_i
end for
```

We now show the workings of this algorithm by converting 247Q into decimal.

Initial value: Result = 0
After iteration 1: Result = (0 × 8) + 2 = 2D;
After iteration 2: Result = (2 × 8) + 4 = 20D;
After iteration 3: Result = (20 × 8) + 7 = 167D.

This is the correct answer, as shown in Example A.2.

A.2.2 Conversion from Decimal

Theoretically, we could use the same procedure to convert a number from the decimal system into a target number system. However, the arithmetic calculations (multiplications and additions) should be done in the target system base. For example, to convert from decimal to hexadecimal, the multiplications and additions involved should be done in base 16, not in base 10. Since we are not used to performing arithmetic operations in nondecimal systems, this is not a pragmatic approach.

Luckily, there is a simple method that allows such base conversions while performing the arithmetic in the decimal system. The procedure is as follows:

> *Divide the decimal number by the base of the target number system and keep track of the quotient and remainder. Repeatedly divide the successive quotients while keeping track of the remainders generated until the quotient is zero. The remainders generated during the process, written in reverse order of generation from left to right, form the equivalent number in the target system.*

This conversion process is shown in the following algorithm:

Algorithm: Decimal to base b conversion
Input: A number $d_{n-1}d_{n-2}\cdots d_1d_0$ in decimal
Output: Equivalent number in the target base b number system
Procedure: Result digits are obtained from left to right. In the following, MOD represents the modulo operator and DIV the integer divide operator.

```
Quotient = decimal number to be converted
while (Quotient ≠ 0)
    next most significant digit of result = Quotient MOD b
    Quotient = Quotient DIV b
end while
```

Example A.4 *Conversion from decimal to binary.*
Convert the decimal number 167 into its equivalent binary number.

		Quotient	Remainder
167/2	=	83	1
83/2	=	41	1
41/2	=	20	1
20/2	=	10	0
10/2	=	5	0
5/2	=	2	1
2/2	=	1	0
1/2	=	0	1

The desired binary number can be obtained by writing the remainders generated in the reverse order from left to right. For this example, the binary number is 10100111B. This agrees with the result of Example A.1 on page 868. □

To understand why this algorithm works, let M be the decimal number that we want to convert into its equivalent representation in the base-b target number system. Let $d_n d_{n-1} \ldots d_1 d_0$ be the equivalent number in the target system. Then

$$\begin{aligned} M &= d_n d_{n-1} \ldots d_1 d_0 \\ &= d_n \cdot b^n + d_{n-1} \cdot b^{n-1} + \cdots + d_1 \cdot b^1 + d_0 \cdot b^0. \end{aligned}$$

Now, to get d_0, divide M by b.

$$\begin{aligned} \frac{M}{b} &= (d_n \cdot b^{n-1} + d_{n-1} \cdot b^{n-2} + \cdots + d_1) + \frac{d_0}{b} \\ &= Q_1 + \frac{d_0}{b}. \end{aligned}$$

Since d_0 is less than b, it represents the remainder of M/b division. To obtain the d_1 digit, divide Q_1 by b. Our algorithm merely formalizes this procedure.

Example A.5 *Conversion from decimal to octal.*
Convert the decimal number 167 into its equivalent in octal.

		Quotient	Remainder
167/8	=	20	7
20/8	=	2	4
2/8	=	0	2

Therefore, 167D is equivalent to 247Q. From Example A.2 on page 868, we know that this is the correct answer. □

Example A.6 *Conversion from decimal to hexadecimal.*
Convert the decimal number 167 into its equivalent in hexadecimal.

		Quotient	Remainder
167/16	=	10	7
10/16	=	0	A

Therefore, 167D = A7H, which is the correct answer (see Example A.3 on page 869). □

A.2.3 Conversion Among Binary, Octal, and Hexadecimal

Conversion among binary, octal, and hexadecimal number systems is relatively easier and more straightforward. Conversion from binary to octal involves converting three bits at a time, whereas binary to hexadecimal conversion requires converting four bits at a time.

Table A.1 3-bit binary to octal conversion

3-bit binary	Octal digit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binary/Octal Conversion

To convert a binary number into its equivalent octal number, form 3-bit groups starting from the right. Add extra 0s at the left-hand side of the binary number if the number of bits is not a multiple of 3. Then replace each group of 3 bits by its equivalent octal digit using Table A.1. With practice, you don't need to refer to the table, as you can easily remember the replacement octal digit. Why three bit groups? Simply because $2^3 = 8$.

Example A.7 *Conversion from binary to octal.*
Convert the binary number 10100111 to its equivalent in octal.

$$\begin{aligned} 10100111\mathrm{B} &= \overbrace{\mathbf{0}10}^{2}\,\overbrace{100}^{4}\,\overbrace{111}^{7}\,\mathrm{B} \\ &= 247\mathrm{Q}\,. \end{aligned}$$

Notice that we have added a leftmost 0 (shown in bold) so that the number of bits is 9. Adding 0s on the left-hand side does not change the value of a number. For example, in the decimal system, 35 and 0035 represent the same value. □

We can use the reverse process to convert numbers from octal to binary. For each octal digit, write the equivalent 3-bit group from Table A.1. You should write exactly 3 bits for each octal digit even if there are leading 0s. For example, for octal digit 0, write the three bits 000.

Example A.8 *Conversion from octal to binary.*
The following two examples illustrate conversion from octal to binary:

Table A.2 4-bit binary to hexadecimal conversion

4-bit binary	Hex digit	4-bit binary	Hex digit
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

$$105\text{Q} = \overbrace{001}^{1}\overbrace{000}^{0}\overbrace{101}^{5}\text{B},$$
$$247\text{Q} = \overbrace{010}^{2}\overbrace{100}^{4}\overbrace{111}^{7}\text{B}.$$

If you want an 8-bit binary number, throw away the leading 0 in the binary number. □

Binary/Hexadecimal Conversion

The process for conversion from binary to hexadecimal is similar except that we use 4-bit groups instead of 3-bit groups because $2^4 = 16$. For each group of 4 bits, replace it by the equivalent hexadecimal digit from Table A.2. If the number of bits is not a multiple of 4, pad 0s at the left.

Example A.9 *Binary to hexadecimal conversion.*
Convert the binary number 1101011111 into its equivalent hexadecimal number.

$$\begin{aligned} 1101011111\text{B} &= \overbrace{\mathbf{00}11}^{3}\overbrace{0101}^{5}\overbrace{1111}^{F}\text{B} \\ &= 35\text{FH}\,. \end{aligned}$$

As in the octal to binary example, we have added two 0s on the left to make the total number of bits a multiple of 4 (i.e., 12). □

The process can be reversed to convert from hexadecimal to binary. Each hex digit should be replaced by exactly four binary bits that represent its value (see Table A.2). An example follows:

Example A.10 *Hex to binary conversion.*
Convert the hexadecimal number B01D into its equivalent binary number.

$$\text{B01DH} = \overbrace{1011}^{B}\,\overbrace{0000}^{0}\,\overbrace{0001}^{1}\,\overbrace{1101}^{D}\text{B}.$$

□

As you can see from these examples, the conversion process is simple if we are working among binary, octal, and hexadecimal number systems. With practice, you will be able to do conversions among these number systems almost instantly.

If you don't use a calculator, division by 2 is easier to perform. Since conversion from binary to hex or octal is straightforward, an alternative approach to converting a decimal number to either hex or octal is to first convert the decimal number to binary and then from binary to hex or octal.

$$\text{Decimal} \Longrightarrow \text{Binary} \Longrightarrow \text{Hex or Octal}.$$

The disadvantage, of course, is that for large numbers, division by 2 tends to be long and thus may lead to simple errors. In such a case, for binary conversion you may want to convert the decimal number to hex or the octal number first and then to binary.

$$\text{Decimal} \Longrightarrow \text{Hex or Octal} \Longrightarrow \text{Binary}.$$

A final note: You don't normally require conversion between hex and octal numbers. If you have to do this as an academic exercise, use binary as the intermediate form, as shown below:

$$\text{Hex} \Longrightarrow \text{Binary} \Longrightarrow \text{Octal},$$
$$\text{Octal} \Longrightarrow \text{Binary} \Longrightarrow \text{Hex}.$$

A.3 Unsigned Integer Representation

Now that you are familiar with different number systems, let us turn our attention to how integers (numbers with no fractional part) are represented internally in computers. Of course, we know that the binary number system is used internally. Still, there are a number of other details that need to be sorted out before we have a workable internal number representation scheme.

We begin our discussion by considering how unsigned numbers are represented using a fixed number of bits. We then proceed to discuss the representation for signed numbers in the next section.

The most natural way to represent unsigned (i.e., nonnegative) numbers is to use the equivalent binary representation. As discussed in Section A.1.1, a binary number with n bits can represent 2^n different values, and the range of the numbers is from 0 to $(2^n - 1)$. Padding of 0s on the left can be used to make the binary conversion of a decimal number equal exactly N bits. For example, to represent 16D we need $\lceil \log_2 16 \rceil = 5$ bits. Therefore, 16D = 10000B. However, this can be extended to a byte (i.e., $N = 8$) as

```
00010000B
```

or to the word size (i.e., $N = 16$) as

```
00000000 00010000B
```

A problem arises if the number of bits required to represent an integer in binary is more than the N bits we have. Clearly, such numbers are outside the range of numbers that can be represented using N bits. Recall that using N bits, we can represent any integer X such that

$$0 \leq X \leq 2^N - 1 .$$

A.3.1 Arithmetic on Unsigned Integers

In this section, the four basic arithmetic operations—addition, subtraction, multiplication, and division—are discussed.

Addition

Since the internal representation of unsigned integers is the binary equivalent, binary addition should be performed on these numbers. Binary addition is similar to decimal addition except that we are using the base-2 number system.

When you are adding two bits x_i and y_i, you generate a result bit z_i and a possible carry bit C_{out}. For example, in the decimal system when you add 6 and 7, the result digit is 3, and there is a carry. The following table, called a *truth table*, covers all possible bit combinations that x_i and y_i can assume. We use these truth tables to derive digital logic circuit designs to perform addition. For more details, see our discussion in Section 3.5 on page 95.

Input bits		Output bits	
x_i	y_i	z_i	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This truth table describes the functionality of what is called a *half-adder* to add just two input bits. Such an adder is sufficient only to add the least significant two bits of a binary number. For other bits, there may be a third bit: carry-out generated by adding the bits just right of the current bit position.

This addition involves three bits: two input bits x_i and y_i, as in the half-adder, and a carry-in bit C_{in} from bit position $(i - 1)$. The required functionality is shown in Table A.3, which corresponds to that of the *full-adder*.

Table A.3 Truth table for binary addition

Input bits			Output bits	
x_i	y_i	C_{in}	z_i	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Given this truth table, it is straightforward to perform binary addition. For each three bits involved, use the truth table to see what the output bit value is and if a carry bit is generated. The carry bit C_{out} generated at bit position i will go as the carry-in C_{in} to bit position $(i + 1)$. Here is an example:

Example A.11 *Binary addition of two eight-bit numbers.*

$$\begin{array}{rl} & \texttt{001110} \leftarrow \mathrm{C_{in}} \\ \texttt{174D} = & \texttt{10101110B} \\ \texttt{75D} = & \texttt{01001011B} \\ \hline \texttt{249D} = & \texttt{11111001B} \end{array}$$

□

An overflow is said to have occurred if there is a carry-out of the leftmost bit position, as shown in the following example:

Example A.12 *Binary addition with overflow.*
Addition of 174D and 91D results in an overflow, as the result is outside the range of the numbers that can be represented by using eight bits.

$$\begin{array}{rcl} & \begin{array}{c}\text{indicates}\\ \text{overflow}\\ \downarrow\end{array} & \\ & \mathbf{1}\texttt{1111110} \leftarrow \mathrm{C_{in}} & \\ \texttt{174D} = & \texttt{10101110B} & \\ \texttt{91D} = & \texttt{01011011B} & \\ \hline \texttt{265D} \neq & \texttt{00001001B} & \end{array}$$

Table A.4 Truth table of binary subtraction

Input bits			Output bits	
x_i	y_i	B_{in}	z_i	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The overflow condition implies that the sum is not in the range of numbers that can be represented using eight bits, which is 0 through 255D. To represent 265D, we need nine bits. You can verify that `100001001B` is the binary equivalent of 265D. □

Subtraction

The subtraction operation is similar to the addition operation. The truth table for binary subtraction is shown in Table A.4. The inputs are two input bits x_i and y_i, and a borrow-in B_{in}. The subtraction operation generates a result bit z_i and a borrow-out B_{out}. Table A.4 shows the two output bits when $x_i - y_i$ is performed.

Example A.13 *Binary subtraction of two eight-bit numbers.*
Perform binary subtraction of 110D from 201D.

```
        1111110 ← Bin
201D = 11001001B
110D = 01101110B
------------------
 91D = 01011011B
```

□

If borrow is produced out of the leftmost bit position, an underflow is said to have occurred indicating that the result is too small to be represented. Since we are considering only non-negative integers, any negative result causes an underflow, as shown in the following example:

Example A.14 *Binary subtraction with underflow.*
Subtracting 202D from 201D results in an underflow, as the result is negative.

```
        indicates
        underflow
            ↓
          1 1111110 ← B_in
201D =    11001001B
202D =    11001010B
-1D  ≠    11111111B      (= 255D)
```

Since the result -1 is too small, it cannot be represented. In fact, the result 111111111B represents -1D in the 2's complement representation of signed numbers, as we show in Section A.4.4. □

In practice, the subtract operation is treated as the addition of the negated second operand. That is, 50D − 40D is treated as 50D + (−40D). Then, of course, we need to discuss how the signed numbers are represented. This is the topic of the next section. Now, however, let us look at how multiplication and division operations are done on unsigned binary numbers. This information is useful if you want to write multiplication/division routines in assembly language. For example, the Pentium does not support multiplying two 64-bit numbers. Although it is unlikely that you will write such a routine, discussion of multiplication and division gives the basic concepts involved.

Multiplication

Let us now consider unsigned integer multiplication. Multiplication is more complicated than either addition or subtraction operations. Multiplying two n-bit numbers could result in a number that requires $2n$ bits to represent. For example, multiplying two 16-bit numbers could produce a 32-bit result.

To understand how binary multiplication is done, it is useful to recall decimal multiplication from when you first learned multiplication. Here is an example:

Example A.15 *Decimal multiplication.*

```
                 123   ← multiplicand
             ×   456   ← multiplier
123 × 6 ⇒        738
123 × 5 ⇒       615
123 × 4 ⇒      492
Product ⇒      56088
```

We started with the least significant digit of the multiplier, and the partial product 123 × 6 = 738 is computed. The next higher digit (5) of the multiplier is used to generate the next partial product 123 × 5 = 615. But since digit 5 has a positional weight of 10, we should actually do 123 × 50 = 6150. This is implicitly done by left-shifting the partial product 615 by one digit position. The process is repeated until all digits of the multiplier are processed. Binary multiplication follows exactly the same procedure except that the base-2 arithmetic is performed, as shown in the next example. □

Example A.16 *Binary multiplication of unsigned integers.*

```
14D ⇒                1110B   ← multiplicand
11D ⇒          ×     1011B   ← multiplier
1110 × 1 ⇒           1110
1110 × 1 ⇒          1110
1110 × 0 ⇒         0000
1110 × 1 ⇒        1110
Product ⇒       10011010B    = 154D
```

As you can see, the final product generated is the correct result. □

The following algorithm formalizes this procedure with a slight modification:

Algorithm: Multiplication of unsigned binary numbers
Input: Two n-bit numbers—a multiplicand and a multiplier
Output: A $2n$-bit result that represents the product
Procedure:

```
product = 0
for (i = 1 to n)
    if (least significant bit of the multiplier = 1)
    then
        product = product + multiplicand
    end if
    shift left multiplicand by one bit position
        {Equivalent to multiplying by 2}
    shift right the multiplier by one bit position
        {This will move the next higher bit into
         the least significant bit position for testing}
end for
```

Here is how the algorithm works on the data of Example A.16.

Iteration	Product	Multiplicand	Multiplier
Initial values	00000000	1110	1011
After iteration 1	00001110	11100	101
After iteration 2	00101010	111000	10
After iteration 3	00101010	1110000	1
After iteration 4	10011010	11100000	0

Division

The division operation is complicated as well. It generates two results: a *quotient* and a *remainder*. If we are dividing two n-bit numbers, division could produce an n-bit quotient and another n-bit remainder. As in the case of multiplication, let us first look at a decimal longhand division example:

Example A.17 *Decimal division.*
Use longhand division to divide 247861D by 123D.

```
                    2015   ← quotient
divisor →    123) 247861
123 × 2 ⇒         -246
                    18
123 × 0⇒           -00
                    186
123 × 1⇒           -123
                     631
123 × 5⇒           -615
                      16   ← remainder
```

This division produces a quotient of 2015 and a remainder of 16. □

Binary division is simpler than decimal division because binary numbers are restricted to 0s and 1s: either subtract the divisor or do nothing. Here is an example of binary division.

Example A.18 *Binary division of unsigned numbers.*
Divide two 4-bit binary numbers: the dividend is `1011B` (11D), and the divisor is `0010B` (2D). The dividend is extended to 8 bits by padding 0s at the left-hand side.

```
                     00101    ← quotient
divisor →    0010) 00001011
0010 × 0 ⇒         -0000
                     0001
0010 × 0 ⇒         -0000
                     0010
0010 × 1 ⇒         -0010
                     0001
0010 × 0 ⇒         -0000
                     0011
0010 × 1 ⇒         -0010
                      001   ← remainder
```

The quotient is `00101B` (5D) and the remainder is `001B` (1D). □

The following binary division algorithm is based on this longhand division method:

Algorithm: Division of two n-bit unsigned integers
Inputs: A $2n$-bit dividend and n-bit divisor
Outputs: An n-bit quotient and an n-bit remainder replace the $2n$-bit dividend. Higher-order n bits of the dividend (dividend_Hi) will have the n-bit remainder, and the lower-order n bits (dividend_Lo) will have the n-bit quotient.
Procedure:

```
for (i = 1 to n)
    shift the 2n-bit dividend left by one bit position
        {vacated right bit is replaced by a 0.}
    if (dividend_Hi ≥ divisor)
    then
        dividend_Hi = dividend_Hi − divisor
        dividend = dividend + 1 {set the rightmost bit to 1}
    end if
end for
```

The following table shows how the algorithm works on the data of Example A.18:

Iteration	Dividend	Divisor
Initial values	00001011	0010
After iteration 1	00010110	0010
After iteration 2	00001101	0010
After iteration 3	00011010	0010
After iteration 4	00010101	0010

The dividend after iteration 4 is interpreted as

$$\underbrace{0001}_{remainder}\ \underbrace{0101}_{quotient}.$$

The lower four bits of the dividend (0101B = 5D) represent the quotient, and the upper four bits (0001B = 1D) represent the remainder.

A.4 Signed Integer Representation

There are several ways in which signed numbers can be represented. These include

- Signed magnitude,
- Excess-M,
- 1's complement, and
- 2's complement.

Table A.5 Number representation using 4-bit binary (All numbers except Binary column in decimal)

Unsigned representation	Binary pattern	Signed magnitude	Excess-7	1's Complement	2's Complement
0	0000	0	−7	0	0
1	0001	1	−6	1	1
2	0010	2	−5	2	2
3	0011	3	−4	3	3
4	0100	4	−3	4	4
5	0101	5	−2	5	5
6	0110	6	−1	6	6
7	0111	7	0	7	7
8	1000	−0	1	−7	−8
9	1001	−1	2	−6	−7
10	1010	−2	3	−5	−6
11	1011	−3	4	−4	−5
12	1100	−4	5	−3	−4
13	1101	−5	6	−2	−3
14	1110	−6	7	−1	−2
15	1111	−7	8	−0	−1

The following subsections discuss each of these methods. However, most modern computer systems, including Pentium-based systems, use the 2's complement representation, which is closely related to the 1's complement representation. Therefore, our discussion of the other two representations is rather brief.

A.4.1 Signed Magnitude Representation

In signed magnitude representation, one bit is reserved to represent the sign of a number. The most significant bit is used as the sign bit. Conventionally, a sign bit value of 0 is used to represent a positive number and 1 for a negative number. Thus, if we have N bits to represent a number, $(N-1)$ bits are available to represent the magnitude of the number. For example, when N is 4, Table A.5 shows the range of numbers that can be represented. For comparison, the unsigned representation is also included in this table. The range of n-bit signed magnitude representation is $-2^{n-1}+1$ to $+2^{n-1}-1$. Note that in this method, 0 has two representations: $+0$ and -0.

A.4.2 Excess-M Representation

In this method, a number is mapped to a nonnegative integer so that its binary representation can be used. This transformation is done by adding a value called *bias* to the number to be

represented. For an n bit representation, the bias should be such that the mapped number is less than 2^n.

To find out the binary representation of a number in this method, simply add the bias M to the number and find the corresponding binary representation. That is, the representation for number X is the binary representation for the number $X + M$, where M is the bias. For example, in the excess-7 system, -3D is represented as

$$-3 + 7 \;=\; +4 \;=\; 0100\text{B}\,.$$

Numbers represented in excess-M are called *biased integers* for obvious reasons. Table A.5 gives examples of biased integers using 4-bit binary numbers. This representation, for example, is used to store the exponent values in the floating-point representation (discussed in Section A.5).

A.4.3 1's Complement Representation

As in the excess-M representation, negative values are biased in 1's complement and 2's complement representations. For positive numbers, the standard binary representation is used. As in the signed magnitude representation, the most significant bit indicates the sign (0 = positive and 1 = negative). In 1's complement representation, negative values are biased by $b^n - 1$, where b is the base or radix of the number system. For the binary case that we are interested in here, the bias is $2^n - 1$. For the negative value $-X$, the representation used is the binary representation for $(2^n - 1) - X$. For example, if n is 4, we can represent -5 as

$$\begin{array}{rcr} 2^4 - 1 & = & 1111\text{B} \\ -5 & = & \underline{-0101\text{B}} \\ & & 1010\text{B} \end{array}$$

As you can see from this example, the 1's complement of a number can be obtained by simply complementing individual bits (converting 0s to 1s and vice versa) of the number. Table A.5 shows 1's complement representation using 4 bits. In this method also, 0 has two representations. The most significant bit is used to indicate the sign. To find the magnitude of a negative number in this representation, apply the process used to obtain the 1's complement (i.e., complement individual bits) again.

Example A.19 *Finding magnitude of a negative number in 1's complement representation.*
Find the magnitude of 1010B that is in 1's complement representation. Since the most significant bit is 1, we know that it is a negative number.

$$1010\text{B} \longrightarrow \text{complement bits} \longrightarrow 0101 = 5\text{D}.$$

Therefore, 1010B = -5D. □

Addition

Standard binary addition (discussed in Section A.3.1) can be used to add two numbers in 1's complement form with one exception: any carry-out from the leftmost bit (i.e., sign bit) should be added to the result. Since the carry-out can be 0 or 1, this additional step is needed only when a carry is generated out of the sign bit position.

Example A.20 *Addition in 1's complement representation.*
The first example shows addition of two positive numbers. The second example illustrates how subtracting $5 - 2$ can be done by adding -2 to 5. Notice that the carry-out from the sign bit position is added to the result to get the final answer.

```
+5D =  0101B          +5D =  0101B
+2D =  0010B          -2D =  1101B
+7D =  0111B               10010B
                             ↳  1
                      +3D =  0011B
```

The next two examples cover the remaining two combinations of the input operands.

```
-5D =  1010B          -5D =  1010B
+2D =  0010B          -2D =  1101B
-3D =  1100B               10111B
                             ↳  1
                      -7D =  1000B
```

Recall that, from Example A.12, a carry-out from the most significant bit position indicates an overflow condition for unsigned numbers. This, however, is not true here. □

Overflow: With unsigned numbers, we have stated that the overflow condition can be detected when there is a carry-out from the leftmost bit position. Since this no longer holds here, how do we know if an overflow has occurred? Let us see what happens when we create an overflow condition.

Example A.21 *Overflow examples.*
Here are two overflow examples that use 1's complement representation for signed numbers:

```
+5D =  0101B                    -5D =  1010B
+3D =  0011B                    -4D =  1011B
+8D ≠  1000B   (= −7D)               10101B
                                       ↳  1
                                -9D ≠  0110B   (= +6D)
```

Clearly, +8 and −9 are outside the range. Remembering that the leftmost bit is the sign bit, 1000B represents −7 and 0110B represents +6. Both answers are incorrect. □

If you observe these two examples closely, you will notice that in both cases the sign bit of the result is reversed. In fact, this is the condition to detect overflow when signed numbers are added. Also note that overflow can only occur with addition if both operands have the same sign.

Subtraction

Subtraction can be treated as the addition of a negative number. We have already seen this in Example A.20.

Example A.22 *Subtraction in 1's complement representation.*
To subtract 7 from 4 (i.e., to perform $4 - 7$), get 1's complement representation of −7, and add this to 4.

$$\begin{array}{ll} \texttt{+4D} = \texttt{0100B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{0100B} \\ \underline{\texttt{-7D} = \texttt{0111B} \overset{1\text{'s complement}}{\longrightarrow\longrightarrow\longrightarrow\longrightarrow}} & \underline{\texttt{1000B}} \\ \texttt{-3D} = & \texttt{1100B} \end{array}$$

The result is 1100B = −3, which is the correct answer. □

Overflow: The overflow condition cannot arise with subtraction if the operands involved are of the same sign. The overflow condition can be detected here if the sign of the result is the same as that of the subtrahend (i.e., second operand).

Example A.23 *A subtraction example with overflow.*
Subtract −3 from 5 (i.e., perform $5 - (-3)$).

$$\begin{array}{lll} \texttt{+5D} & = \texttt{0101B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{0101B} \\ \underline{\texttt{-(-3D)}} & = \texttt{1100B} \overset{1\text{'s complement}}{\longrightarrow\longrightarrow\longrightarrow\longrightarrow} & \underline{\texttt{0011B}} \\ \texttt{+8D} & \neq & \texttt{1000B} \end{array}$$

Overflow has occurred here because the subtrahend is negative and the result is negative. □

Example A.24 *Another subtraction example with underflow.*
Subtract 3 from −5 (i.e., perform $-5 - (3)$).

$$\begin{array}{lll} \texttt{-5D} & = \texttt{1010B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{1010B} \\ \texttt{-(+3D)} & = \texttt{0011B} \overset{1\text{'s complement}}{\longrightarrow\longrightarrow\longrightarrow\longrightarrow} & \underline{\texttt{1100B}} \\ & & \texttt{10110B} \\ & & \underline{\texttt{↳ 1}} \\ \texttt{-8D} & \neq & \texttt{0111B} \end{array}$$

An underflow has occurred in this example, as the sign of the subtrahend is the same as that of the result (both are positive). □

Representation of signed numbers in 1's complement representation allows the use of simpler circuits for performing addition and subtraction than the other two representations we have seen so far (signed magnitude and excess-M). Some older computer systems used this representation for integers. An irritant with this representation is that 0 has two representations. Furthermore, the carry bit generated out of the sign bit will have to be added to the result. The 2's complement representation avoids these pitfalls. As a result, 2's complement representation is the choice of current computer systems.

A.4.4 2's Complement Representation

In 2's complement representation, positive numbers are represented the same way as in the signed magnitude and 1's complement representations. The negative numbers are biased by 2^n, where n is the number of bits used for number representation. Thus, the negative value $-A$ is represented by $(2^n - A)$ using n bits. Since the bias value is one more than that in the 1's complement representation, we have to add 1 after complementing to obtain the 2's complement representation of a negative number. We can, however, discard any carry generated out of the sign bit.

Example A.25 *2's complement representation.*
Find the 2's complement representation of -6, assuming that 4 bits are used to store numbers.

$$\begin{array}{lll} \texttt{6D} = \texttt{0110B} \longrightarrow & \text{complement} \longrightarrow & \texttt{1001B} \\ & \text{add 1} & \texttt{1B} \\ & & \hline \\ & & \texttt{1010B} \end{array}$$

Therefore, 1010B represents -6D in 2's complement representation. □

Table A.5 shows the 2's complement representation of numbers using 4 bits. Notice that there is only one representation for 0. The range of an n-bit 2's complement integer is -2^{n-1} to $+2^{n-1} - 1$. For example, using 8 bits, the range is -128 to $+127$.

To find the magnitude of a negative number in the 2's complement representation, as in the 1's complement representation, simply reverse the sign of the number. That is, use the same conversion process (i.e., complement and add 1 and discard any carry generated out of the leftmost bit).

Example A.26 *Finding the magnitude of a negative number in 2's complement representation.*
Find the magnitude of the 2's complement integer 1010B. Since the most significant bit is 1, we know that it is a negative number.

$$\begin{array}{lll} \texttt{1010B} \longrightarrow & \text{complement} \longrightarrow & \texttt{0101B} \\ & \text{add 1} & \texttt{1B} \\ & & \hline \\ & & \texttt{0110B} \quad (= \text{6D}) \end{array}$$

The magnitude is 6. That is, 1010B = $-$6D. □

Addition and Subtraction

Both of these operations work in the same manner as in the case of 1's complement representation except that any carry-out from the leftmost bit (i.e., sign bit) is discarded. Here are some examples:

Example A.27 *Examples of addition operation.*

```
+5D =  0101B          +5D =  0101B
+2D =  0010B          -2D =  1110B
+7D =  0111B          +3D   10011B
```

Discarding the carry leaves the result `0011B` = +3D.

```
-5D =  1011B          -5D =  1011B
+2D =  0010B          -2D =  1110B
-3D =  1101B          -7D   11001B
```

Discarding the carry leaves the result `1001B` = −7D.

As in the 1's complement case, subtraction can be done by adding the negative value of the second operand.

A.5 Floating-Point Representation

So far, we have discussed various ways of representing integers, both unsigned and signed. Now let us turn our attention to representation of numbers with fractions (called *real numbers*). We start our discussion by looking at how fractions can be represented in the binary system. Next we discuss how fractions can be converted from decimal to binary and vice versa. Finally, we discuss how real numbers are stored in computers.

A.5.1 Fractions

In the decimal system, which is a positional number system, fractions are represented like the integers except for different positional weights. For example, when we write in decimal

$$0.7913$$

the value it represents is

$$(7 \times 10^{-1}) + (9 \times 10^{-2}) + (1 \times 10^{-3}) + (3 \times 10^{-4}).$$

The decimal point is used to identify the fractional part of a number. The position immediately to the right of the decimal point has the weight 10^{-1}, the next position 10^{-2}, and so on. If we count the digit position from the decimal point (left to right) starting with 1, the weight of position n is 10^{-n}.

This can be generalized to any number system with base b. The weight should be b^{-n}, where n is defined as above. Let us apply this to the binary system that is of interest to us. If we write a fractional binary number

$$0.11001\text{B}$$

the decimal value it represents is

$$1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 0.78125\text{D}\,.$$

The period in the binary system is referred to as the *binary point*. Thus, the algorithm to convert a binary fraction to its equivalent in decimal is straightforward.

Algorithm: Binary fraction to decimal
Input: A fractional binary number $0.d_1\,d_2 \ldots d_{n-1}d_n$ with n bits (trailing 0s can be ignored)
Output: Equivalent decimal value
Procedure: Bits in the input fraction are processed from right to left starting with bit d_n.

```
decimal_value = 0.0
for (i = n downto 1)
    decimal_value = (decimal_value + d_i )/b
end for
```

Here is an example showing how the algorithm works on the binary fraction 0.11001B:

Iteration	Decimal_value
Initial value	0.0
Iteration 1	(0.0 + 1)/2 = 0.5
Iteration 2	(0.5 + 0)/2 = 0.25
Iteration 3	(0.25 + 0)/2 = 0.125
Iteration 4	(0.125 + 1)/2 = 0.5625
Iteration 5	(0.5625 + 1)/2 = 0.78125

Now that we know how to convert a binary fraction into its decimal equivalent, let us look at how we can do the reverse conversion: from decimal fraction to equivalent binary.

This conversion can be done by repeatedly multiplying the fraction by the base of the target system, as shown in the following example:

Example A.28 *Conversion of a fractional decimal number to binary.*
Convert the decimal fraction 0.78125D into its equivalent in binary.

$$\begin{array}{rcll} 0.78125 \times 2 & = & 1.5625 & \longrightarrow \quad 1 \\ 0.5625 \times 2 & = & 1.125 & \longrightarrow \quad 1 \\ 0.125 \times 2 & = & 0.25 & \longrightarrow \quad 0 \\ 0.25 \times 2 & = & 0.5 & \longrightarrow \quad 0 \\ 0.5 \times 2 & = & 1.0 & \longrightarrow \quad 1 \\ \text{Terminate.} & & & \end{array}$$

The binary fraction is 0.11001B, which is obtained by taking numbers from the top and writing them left to right with a binary point. □

What we have done is to multiply the number by the target base (to convert to binary use 2) and the integer part of the multiplication result is placed as the first digit immediately to the right of the radix or base point. Take the fractional part of the multiplication result and repeat the process to produce more digits. The process stops when the fractional part is 0, as in the above example, or when we have the desired number of digits in the fraction. This is similar to what we do in the decimal system when dividing 1 by 3. We write the result as 0.33333 if we want only 5 digits after the decimal point.

Example A.29 *Conversion of a fractional decimal number to octal.*
Convert 0.78125D into the octal equivalent.

$$\begin{array}{rcll} 0.78125 \times 8 & = & 6.25 & \longrightarrow \quad 6 \\ 0.25 \times 8 & = & 2.0 & \longrightarrow \quad 2 \\ \text{Terminate.} & & & \end{array}$$

Therefore, the octal equivalent of 0.78125D is 0.62Q. □

Without a calculator, multiplying a fraction by 8 or 16 is not easy. We can avoid this problem by using the binary as the intermediate form, as in the case of integers. First convert the decimal number to its binary equivalent and group 3 bits (for octal conversion) or 4 bits (for hexadecimal conversion) from left to right (pad 0s at the right if the number of bits in the fraction is not a multiple of 3 or 4).

Example A.30 *Conversion of a fractional decimal number to octal.*
Convert 0.78125D to octal using the binary intermediate form. From Example A.28, we know that 0.78125D = 0.11001B. Now convert 0.11001B to octal.

$$0.\underbrace{110}_{6}\,\underbrace{01\mathbf{0}}_{2} = 0.62\text{Q}\,.$$

Notice that we have added a 0 (shown in bold) on the right. □

Example A.31 *Conversion of a fractional decimal number to hexadecimal.*
Convert 0.78125D to hexadecimal using the binary intermediate form. From Example A.28, we know that 0.78125D = 0.11001B. Now convert 0.11001B to hexadecimal.

$$0.\underbrace{1100}_{12=\mathrm{C}}\underbrace{1\mathbf{000}}_{8} = 0.\mathrm{C8H}\,.$$

We have to add three 0s on the right to make the number of bits equal to 8 (a multiple of 4). □

The following algorithm gives this conversion process:

Algorithm: Conversion of fractions from decimal to base b system
Input: Decimal fractional number
Output: Its equivalent in base b with a maximum of F digits
Procedure: The function `integer` returns the integer part of the argument and the function `fraction` returns the fractional part.

```
value = fraction to be converted
digit_count = 0
repeat
    next digit of the result = integer (value × b)
    value = fraction (value × b)
    digit_count = digit_count + 1
until ((value = 0) OR (digit_count = F))
```

We leave tracing the steps of this algorithm as an exercise.

If a number consists of both integer and fractional parts, convert each part separately and put them together with a binary point to get the desired result. This is illustrated in Example A.33 on page 894.

A.5.2 Representing Floating-Point Numbers

A naive way to represent real numbers is to use direct representation: allocate I bits to store the integer part and F bits to store the fractional part, giving us the format with N ($= I + F$) bits as shown below:

$$\underbrace{??\cdots??}_{I\text{ bits}}.\underbrace{??\cdots??}_{F\text{ bits}}.$$

This is called *fixed-point representation*.

Representation of integers in computers should be done with a view of the range of numbers that can be represented. The desired range dictates the number of bits required to store a number. As discussed earlier,

$$\text{the number of bits required} = \lceil \log_b R \rceil\,,$$

where R is the number of different values to be represented. For example, to represent 256 different values, we need 8 bits. The range can be 0 to 255D (for unsigned numbers) or -128D to +127D (for signed numbers). To represent numbers outside this range requires more bits.

Representation of real numbers introduces one additional factor: once we have decided to use N bits to represent a real number, the next question is where do we place the binary point. That is, what is the value of F? This choice leads to a tradeoff between the *range* and *precision*. Precision refers to how accurately we can represent a given number. For example, if we use 3 bits to represent the fractional part ($F = 3$), we have to round the fractional part of a number to the nearest 0.125 ($= 2^{-3}$). Thus, we lose precision by introducing rounding errors. For example, 7.80D may be stored as 7.75D. In general, if we use F bits to store the fractional part, the rounding error is bound by $\frac{1}{2} \cdot \frac{1}{2^F}$ or $1/2^{F+1}$.

In summary, range is largely determined by the integer part, and precision is determined by the fractional part. Thus, given N bits to represent a real number where $N = I + F$, the tradeoff between range and precision is obvious. Increasing the number of bits F to represent the fractional part increases the precision but reduces the range, and vice versa.

Example A.32 *Range versus precision tradeoff.*
Suppose we have $N = 8$ bits to represent positive real numbers using fixed-point representation. Show the range versus precision tradeoff when F is changed from 3 to 4 bits.

When $F = 3$, the value of I is $I = N - F = 5$ bits. Using this allocation of bits for F and I, a real number X can be represented that satisfies $0 \leq X < 2^5$ (i.e., $0 \leq X < 32$). The precision (i.e., maximum rounding error) is $1/2^{3+1} = 0.0625$.

If we increase F by 1 bit to 4 bits, the range decreases approximately by half to $0 \leq X < 2^4$ (i.e., $0 \leq X < 16$). The precision, on the other hand, improves to $1/2^{4+1} = 0.03125$. □

Fixed-point representation is simple but suffers from the serious disadvantage of limited range. This may not be acceptable for most applications, in particular, fixed-point's inability to represent very small and very large numbers without requiring a large number of bits.

Using scientific notation, we can make better use of the given number of bits to represent a real number. The next section discusses *floating-point* representation, which is based on the scientific notation.

A.5.3 Floating-Point Representation

Using the decimal system for a moment, we can write very small and very large numbers in scientific notation as follows:

$$1.2345 \times 10^{45},$$

$$9.876543 \times 10^{-37}.$$

Expressing such numbers using the positional number notation is difficult to write and understand, errorprone, and requires more space. In a similar fashion, binary numbers can be written in scientific notation. For example,

$$\begin{aligned} +1101.101 \times 2^{+11001} &= 13.625 \times 2^{25} \\ &= 4.57179 \times 10^{8}. \end{aligned}$$

As indicated, numbers expressed in this notation have two parts: a *mantissa* (or *significand*), and an *exponent*. There can be a sign ($+$ or $-$) associated with each part.

Numbers expressed in this notation can be written in several equivalent ways, as shown below:

$$\begin{aligned} 1.2345 &\times 10^{45}, \\ 123.45 &\times 10^{43}, \\ 0.00012345 &\times 10^{49}. \end{aligned}$$

This causes implementation problems to perform arithmetic operations, comparisons, and the like. This problem can be avoided by introducing a standard form called *normal form*. Reverting to the binary case, a normalized binary form has the format

$$\pm 1.X_1X_2 \cdots X_{M-1}X_M \times 2^{\pm Y_{N-1}Y_{N-2}\cdots Y_1Y_0},$$

where X_i and Y_j represent a bit, $1 \leq i \leq M$, and $0 \leq j < N$. The normalized form of

$$+1101.101 \times 2^{+11010}$$

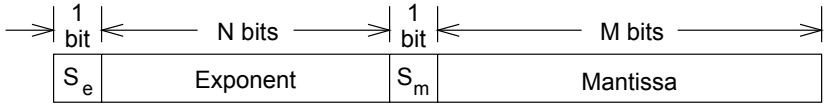
is

$$+1.101101 \times 2^{+11101}.$$

We normally write such numbers as

$$+1.101101\mathrm{E}11101.$$

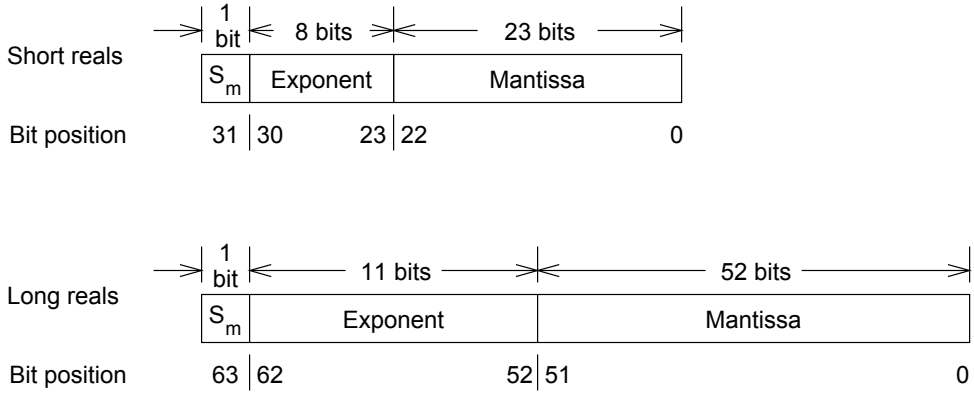
To represent such normalized numbers, we might use the format shown below:



where S_m and S_e represent the sign of mantissa and exponent, respectively.

Implementation of floating-point numbers on computer systems varies from this generic format, usually for efficiency reasons or to conform to a standard. From here on, we discuss the specific format used by the Pentium, which conforms to the IEEE 754 floating-point standard. Such standards are useful, for example, to exchange data among several different computer systems and to write efficient numerical software libraries.

The Pentium supports three formats for floating-point numbers: two of these are for external use and one for internal use. The internal format is used to store temporary results, and we do not discuss this format. The remaining two formats are shown below:



Certain points are worth noting about these formats:

1. The mantissa stores only the fractional part of a normalized number. The 1 to the left of the binary point is not explicitly stored but implied to save a bit. Since this bit is always 1, there is really no need to store it. However, representing 0.0 requires special attention, as we show later.
2. There is no sign bit associated with the exponent. Instead, the exponent is converted to an excess-M form and stored. For short reals, the bias used is 127D (= 7FH), and for long reals, 1023 (= 3FFH).

We now show how a real number can be converted to its floating-point equivalent:

Algorithm: Conversion to floating-point representation
Input: A real number in decimal
Output: Floating-point equivalent of the decimal number
Procedure: The procedure consists of four steps.
Step 1: *Convert the real number to binary.*
1a: Convert the integer part to binary using the procedure described in Section A.2.2 (page 870).
1b: Convert the fractional part to binary using the procedure described in Section A.5.1 (page 890).
1c: Put them together with a binary point.
Step 2: *Normalize the binary number.*
Move the binary point left or right until there is only a single 1 to the left of the binary point while adjusting the exponent appropriately. You should increase the exponent value by 1 if the binary point is moved to the left by one bit position; decrement by 1 if moving to the right.
Note that 0.0 is treated as a special case; see text for details.

Step 3: *Convert the exponent to excess or biased form.*
For short reals, use 127 as the bias;
For long reals, use 1023 as the bias.
Step 4: *Separate the three components.*
Separate mantissa, exponent, and sign
to store in the desired format.

Here is an example to illustrate the above procedure:

Example A.33 *Conversion to floating-point format.*
Convert 78.8125D to short floating-point format.

Step 1: Convert 78.8125D to the binary form.
1a: Convert 78 to the binary.
78D = 1001110B.
1b: Convert 0.8125D to the binary form.
0.8125D = 0.1101B.
1c: Put together the two parts.
78.8125D = 1001110.1101B.
Step 2: Normalize the binary number.
1001110.1101 = 1001110.1101E0
= 1.0011101101E110.
Step 3: Convert the exponent to the biased form.
110B + 1111111B = 10000101B (i.e., 6D + 127D = 133D).
Thus, 78.8125D = 1.0011101101E10000101
in the normalized short real form.
Step 4: Separate the three components.
Sign: 0 (positive number)
mantissa: 0011101101
(1 to the left of the binary point is implied)
exponent: 10000101.

Storing the short real in memory requires 4 bytes (32 bits), and the long real requires 8 bytes (or 64 bits). For example, the short real form of 78.8125D is stored as shown below:

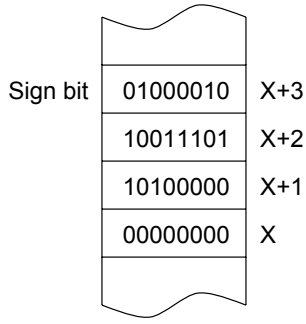
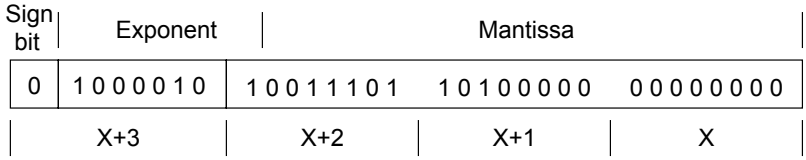


Table A.6 Representation of special values in the floating-point format

Special number	Sign	Exponent (biased)	Mantissa
$+0$	0	0	0
-0	1	0	0
$+\infty$	0	FFH	0
$-\infty$	1	FFH	0
NaN	$0/1$	FFH	$\neq 0$
Denormals	$0/1$	0	$\neq 0$

If we lay these four bytes linearly, they look like this:



To find the decimal values of a number that is in one of the floating-point formats, use the procedure in reverse.

Special Values

The representations of 0 and infinity (∞) require special attention. Table A.6 shows the values of the three components to represent these values. Zero is represented by a zero exponent and fraction. We can have a -0 or $+0$ depending on the sign bit. An exponent of all ones indicates a special floating-point value. An exponent of all ones with a zero mantissa indicates infinity. Again, the sign bit indicates the sign of the infinity. An exponent of all ones with a nonzero mantissa represents a not-a-number (NaN). The NaN values are used to represent operations like 0/0 and $\sqrt{-1}$.

The last entry in Table A.6 shows how *denormalized values* are represented. The denormals are used to represent values smaller than the smallest value that can be represented with normalized floating-point numbers. For denormals, the implicit 1 to the left of the binary point becomes a 0. The smallest normalized number has a 1 for the exponent (note zero is not allowed) and 0 for the fraction. Thus, the smallest number is 1×2^{-126}. The largest denormalized number has a zero exponent and all 1s for the fraction. This represents approximately $0.9999999 \times 2^{-127}$. The smallest denormalized number would have zero as the exponent and a 1 in the last bit position (i.e., position 23). Thus, it represents $2^{-23} \times 2^{-127}$, which is approximately 10^{-45}. A thorough discussion of floating-point numbers is in [13].

A.5.4 Floating-Point Addition

Adding two floating-point numbers involves the following four steps:

- *Match Exponents:* This can be done by shifting right the smaller exponent number. As an example, consider the following two floating-point numbers: 1.10101×2^3(13.25D) and 1.0011×2^2 (4.75D). Since the second number is smaller, it is shifted right by two positions to match the exponents. Thus, after shifting, the second number becomes 0.10011×2^3.
- *Add the Two Mantissas:* In our example, we add 1.10101 and 0.10011 to get 10.01.
- *Normalize the Result:* We move the binary point to the right of the leftmost 1 and adjust the exponent accordingly. In our example, our result 10.01×2^3 is not in the normal form. After normalization, the final result is 1.001×2^4 (18D), which is correct.
- *Test for Overflow/Underflow:* This final step is needed to make sure that the result is within the bounds. In our example, we don't have this problem.

Floating-point subtraction can be done in a similar fashion. The following example illustrates this:

Example A.34 *A floating-point subtraction example.*
Perform 13.25 − 4.75. In the floating-point notation, we can write this as $1.10101 \times 2^3 - 1.00111 \times 2^1$.

- *Step 1:* As in the last example, we shift the second operand to match the exponents.
- *Step 2:* Subtract the mantissas. For our example, we get $1.10101 - 0.10011 = 1.00010$.
- *Step 3:* The result 1.00010×2^3 is already in the normalized form.
- *Step 4:* No underflow as the result is within the range. Thus, the final result is 1.00010×2^3. In decimal, it is equivalent to 8.50, which is correct. □

This procedure can be applied to the IEEE 754 standard format in a straightforward manner.

A.5.5 Floating-Point Multiplication

Floating-point multiplication is straightforward as shown below:

- Add the two exponents using an integer adder;
- Multiply the two mantissas using an integer multiplier;
- Compute the result sign bit as the XOR of the two input sign bits;
- Normalize the final product;
- Check for underflow/overflow.

Example A.35 *A floating-point multiplication example.*
Multiply 1.101×2^3 and 1.01×2^2.

- *Step 1:* We add the two exponents to get 5 as the exponent of the result.
- *Step 2:* Multiplying two mantissas, we get $1.101 \times 1.01 = 10.00001$.
- *Step 3:* The sign of the result is positive.
- *Step 4:* Our result 10.00001×2^5 needs to be normalized.
  The final normalized result is 1.000001×2^6. □

When we apply this algorithm to the IEEE 754 format, we encounter one problem. Since the exponents are biased, when we add the two exponents, the bias from both numbers appears in the result. Thus, we have to subtract the bias value from the result. For short reals, we have to subtract 127 and for long reals, subtract 1023.

A.6 Summary

We discussed how numbers are represented using the positional number system. Positional number systems are characterized by a base and an alphabet. The familiar decimal system is a base-10 system with the alphabet 0 through 9. Computer systems use the binary system for internal storage. This is a base-2 number system with 0 and 1 as the alphabet. The remaining two number systems—octal (base-8) and hexadecimal (base-16)—are mainly used for convenience to write a binary number. For example, debuggers use the hexadecimal numbers to display address and data information.

When we are using several number systems, there is often a need to convert numbers from one system to another. Conversion among binary, octal, and hexadecimal systems is simple and straightforward. We also discussed how numbers are converted from decimal to binary and vice versa.

The remainder of the chapter was devoted to internal representation of numbers. The focus was on the representation of numbers: both integers and real numbers were considered. Representation of unsigned integers is straightforward and uses binary representation. There are, however, several ways of representing signed integers. We discussed four methods to represent signed integers. Of these four methods, current computer systems use the 2's complement representation. In this representation, subtraction can be treated as addition by reversing the sign of the subtrahend.

Floating-point representation on most computers follows the IEEE 754 standard. There are three components of a floating-point number: mantissa, exponent, and the sign of the mantissa. There is no sign associated with the exponent. Instead, the exponent is stored as a biased number. We illustrated how real numbers can be converted from decimal to floating-point format.

The next version of the IEEE 754 standard, known as the IEEE 784, includes decimal-base floating-point numbers. Details on this standard are available from the IEEE standards body.

A.7 Exercises

A–1 How many different values can be represented using four digits in the hexadecimal system? What is the range of numbers that can be represented?

A–2 Repeat the above exercise for the binary system and the octal system.

A–3 Find the decimal equivalent of the following:

(a) 737Q, (c) AB15H, (e) 1234Q,
(b) 11010011B, (d) 1234H, (f) 100100B.

A–4 To represent numbers 0 through 300 (both inclusive), how many digits are required in the following number systems?

1. Binary.
2. Octal.
3. Hexadecimal.

A–5 What are the advantages of the octal and hexadecimal number systems over the binary system?

A–6 Perform the following number conversions:

1. 1011010011B = _____ Q.
2. 1011010011B = _____ H.
3. 1204Q = _____ B.
4. ABCDH = _____ B.

A–7 Perform the following number conversions:

1. 56D = _____ B.
2. 217D = _____ Q.
3. 150D = _____ H.

Verify your answer by converting your answer back to decimal.

A–8 Assume that 16 bits are available to store a number. Specify the range of numbers that can be represented by the following number systems:

1. Unsigned integer.
2. Signed magnitude.
3. Excess-1023.
4. 1's complement.
5. 2's complement.

A–9 What is the difference between a half-adder and a full-adder?

A–10 Perform the following operations assuming that the numbers are unsigned integers. Make sure to identify the presence or absence of the overflow or underflow condition.

1. 01011010B + 10011111B.
2. 10110011B + 01101100B.
3. 11110001B + 00011001B.
4. 10011101B + 11000011B.
5. 01011010B – 10011111B.
6. 10110011B – 01101100B.
7. 11110001B – 00011001B.
8. 10011101B – 11000011B.

A–11 Repeat the above exercise assuming that the numbers are signed integers that use the 2's complement representation.

A–12 Find the decimal equivalent of the following binary numbers assuming that the numbers are expressed in

1. Unsigned integer.
2. Signed magnitude.
3. Excess-1023.
4. 1's complement.
5. 2's complement.

(a) 01101110, (b) 11011011, (c) 00111101,
(d) 11010011, (e) 10001111, (f) 01001101.

A–13 Convert the following decimal numbers into signed magnitude, excess-127, 1's complement, and 2's complement number systems. Assume that 8 bits are used to store the numbers:

(a) 60, (b) 0, (c) –120,
(d) –1, (e) 100, (f) –99.

A–14 Find the decimal equivalent of the following binary numbers:

(a) 10101.0101011, (b) 10011.1101, (c) 10011.1010,
(d) 1011.1011, (e) 1101.001101, (f) 110.111001.

A–15 Convert the following decimal numbers into the short floating-point format:

1. 19.3125.

2. −250.53125.

A–16 Convert the following decimal numbers into the long floating-point format:

1. 19.3125.
2. −250.53125.

A–17 Find the decimal equivalent of the following numbers, which are in the short floating-point format:

1. 7B59H.
2. A971H.
3. BBC1H.

A–18 Give a summary of the special values used in the IEEE 754 standard.

A–19 Explain why denormals are introduced in the IEEE 754 standard.

A–20 We gave the smallest and largest values represented by the denormals for single-precision floating-point numbers. Give the corresponding values for the double precision numbers.

A–21 Perform the following floating-point arithmetic operations (as in Example A.34):

1. 22.625 + 7.5.
2. 22.625 − 7.5.
3. 35.75 + 22.625.
4. 35.75 − 22.625.

A.8 Programming Exercises

A–P1 Implement the algorithm on page 869 to perform binary-to-decimal conversion in your favorite high-level language. Use your program to verify the answers of the exercises that require this conversion.

A–P2 Implement the algorithm on page 870 to perform decimal-to-binary conversion in your favorite high-level language. Use your program to verify the answers of the exercises that require this conversion.

A–P3 Implement the algorithm on page 893 to convert real numbers from decimal to short floating-point format in your favorite high-level language. Use your program to verify the answers of the exercise that requires this conversion.

A–P4 Implement the algorithm to convert real numbers from the short floating-point format to decimal in your favorite high-level language. Assume that the input to the program is given as four hexadecimal digits. Use your program to verify the answers of the exercise that requires this conversion.

Appendix B

Character Representation

Objectives

- To discuss character representation;
- To give ASCII character encoding;
- To describe UCS and Unicode universal character sets.

We give a brief description of character representation in this appendix. We identify the desirable characteristics in a character-encoding scheme. We illustrate these features using the ASCII encoding scheme. We also present ASCII encoding of characters. The ASCII character set is good for representing English letters. It is not useful in encoding characters of the world's languages. We describe two character sets—UCS and Unicode—that provide a uniform standard to encode all (or most) of these characters. We conclude the chapter with a summary.

B.1 Character Sets

As computers have the capability to store and understand the alphabet 0 and 1, characters should be assigned a sequence over this alphabet (i.e., characters should be encoded using this alphabet). If you build and use your computer system in isolation and never communicate or exchange data or programs with others, you can assign arbitrary bit patterns to represent characters. Even then, you may be forced to follow certain guidelines for efficiency reasons. Some of these guidelines are as follows:

1. Assigning a contiguous sequence of numbers (if treated as unsigned binary numbers) to letters in alphabetical order is desired. Upper and lowercase letters (`A` through `Z` and `a` through `z`) can be treated separately, but a contiguous sequence should be assigned to each case.

2. In a similar fashion, digits should be assigned a contiguous sequence in numerical order.
3. A space character should precede all letters and digits.

These guidelines allow for efficient character processing including sorting by names or character strings. For example, to test if a given character code corresponds to a lowercase letter, all we have to do is to see if the code of the character is between that of `a` and `z`. These guidelines also aid in applications requiring sorting, for instance, sorting a class list by last name.

Since computers are rarely used in isolation, exchange of information is an important concern. This leads to the necessity of having some standard way of representing characters. Two such standard character codes have been developed: EBCDIC (Extended Binary Coded Decimal Interchange Code) and ASCII (American Standard Code for Information Interchange). EBCDIC is used on IBM mainframe computers. Most modern computer systems, including the IBM PC, use ASCII for character representation.

The standard ASCII uses 7 bits to encode a character. Thus, $2^7 = 128$ different characters can be represented. This number is sufficiently large to represent uppercase and lowercase characters, digits, special characters such as `!`,`^`, and control characters such as CR (carriage return), LF (line feed), and so on.

Since we store the bits in units of a power of 2, we end up storing 8 bits for each character, even though ASCII requires only 7 bits. The eighth bit is put to use for two purposes:

1. *To Parity Encode for Error Detection:* The eighth bit can be used to represent the parity bit. This bit is made 0 or 1 such that the total number of 1s in a byte is even (for even parity) or odd (for odd parity). This can be used to detect simple errors in data transmission.
2. *To Represent an Additional 128 Characters:* By using all 8 bits we can represent a total of $2^8 = 256$ different characters. This is referred to as extended ASCII. On an IBM PC, special graphics symbols, Greek letters, and so on make up the additional 128 characters.

Notice from the table on page 906 that ASCII encoding satisfies the three guidelines mentioned earlier. For instance, successive bit patterns are assigned to uppercase letters, lowercase letters, and digits. This assignment leads to some good properties. For example, the difference between the uppercase and lowercase characters is constant. That is, the difference between the character codes of `a` and `A` is the same as that between `n` and `N`, which is 32D (20H). This characteristic can be exploited for efficient case conversion.

Another interesting feature of ASCII is that the character codes are assigned to the 10 digits such that the lower-order 4 bits represent the binary equivalent of the corresponding digit. For example, digit `5` is encoded as 0110101. If you take the rightmost 4 bits (0101), they represent 5 in binary. This feature, again, helps in writing an efficient code for character-to-numeric conversion. Such conversion, for example, is required when you type a number as a sequence of digit characters.

B.2 Universal Character Set

The 7-bit ASCII character encoding is fine for English, but is not good for other languages. Some languages such as French and German require accents (ó) and diacritical (ö) marks. It cannot represent characters from other languages (Indian, Chinese, Japanese). As software vendors sell their products to various non-English speaking countries around the world, ASCII encoding is no longer sufficient. As a first attempt, the ASCII encoding was extended by using the eighth bit. Thus, an additional 128 encodings have been added, mostly to take care of the Latin letters, accents, and diacritical marks.

In this section, we look at the universal character set (UCS) to represent characters in various world languages. The next section describes a restricted version called the Unicode.

UCS is a new character-encoding standard from the International Organization for Standardization (ISO/IEC 0646). The objective is to develop a standard to encode all characters used in all the written languages of the world including mathematical and other symbols. To allow this encoding, the code uses two encoding forms:

- UCS-2 uses 16 bits;
- UCS-4 uses 31 bits.

The UCS-2 uses two octets, and the UCS-4 consists of four octets. In fact, the official name for the UCS is "Universal Multiple-Octet Coded Character Set." UCS is not only meant for internal character representation but also for data transmission. As an aside, communication people use octet instead of byte to refer to 8 bits.

UCS-2 allows up to 65536 encodings, which are divided into 256 rows with each row consisting of 256 cells. The first 128 characters are the same as the ASCII encodings. UCS-4 can represent more than 2 billion (i.e., 2^{31}) different characters. This space is divided into 128 groups with each group containing 256 planes. The first octet gives the group number and the second octet gives the plane number. The third and fourth octets give the row and cell numbers as in UCS-2. The characters that can be represented by UCS-2 are called the basic multilingual plane (BMP). An encoding in UCS-2 can be transformed into the UCS-4 by appending two zero octets.

UCS encoding can also be used for data communications. However, most communication protocols treat the values in the range 0 to 1FH as control characters (see the table on page 905). To facilitate adaptation for the communication area, several UCS transformation formats (UTF) are defined. For example, UTF-8 replaces the first half of the first row of the BMP by the ASCII encodings. The other transformations include the UTF-7, which is useful for the SMTP protocol.

B.3 Unicode

The Unicode Consortium consisting of major American computer manufacturers and organizations developed the Unicode standard. It uses 16 bits to encode the characters. Each encoding in Unicode is called a *code point*. The number of code points available (65,536) is much smaller

than the number of characters in the world languages. Thus, care should be exercised in allocating code points. As does the UCS, the Unicode Standard further includes punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, and so on. The Unicode Standard, Version 3.0 allocated code points for 49,194 (out of 65,536) for characters from the world's alphabets, ideograph sets, and symbol collections. These all fit into the first 64 K characters of the BMP. The Unicode Standard also reserves code points for private use. Vendors or end-users can use these code points for their own characters and symbols, or use them with specialized fonts.

The Unicode is compatible with the UCS-2. Unicode 3.0 contains all the same characters and encoding points as ISO/IEC 10646-1:2000. The Unicode Standard provides additional information about the characters and their use. Any implementation that is conformant to Unicode is also conformant to ISO/IEC 10646 [38].

Some text elements may be encoded as composed character sequences, which should be rendered together for presentation. For example, â is a *composite character* created by rendering "a" and "^" together. A composed character sequence is typically made up of a base letter, which occupies a single space, and one or more nonspacing marks.

Like the UCS, several UTFs are defined for the Unicode. UTF-8 is popular for HTML and similar protocols. The main advantage of UTF-8 is that it maintains compatibility with ASCII. Unicode characters transformed into UTF-8 can be used with the existing software without extensive software modifications. UTF-16 strikes a balance between efficient access to characters and economical use of storage. It is reasonably compact, all the heavily used characters fit into a single 16-bit code unit, and all other characters are accessible via pairs of 16-bit code units. When storage efficiency is not a concern, we can use UTF-32 to provide fixed-width for all characters as does the UCS. Character composition is no longer needed in this encoding form.

B.4 Summary

This appendix discussed character representation. We identified some desirable properties that a character-encoding scheme should satisfy in order to facilitate efficient character processing. Although there are two simple character codes—EBCDIC and ASCII—most computers use the ASCII character set. We noted that ASCII satisfies the requirements of an efficient character code. We also presented details on two universal character sets, UCS and Unicode.

The next pages give the standard ASCII character set. We divide the character set into control and printable characters. The control character codes are given on the next page and the printable ASCII characters are on page 906.

ASCII Control Codes

Hex	Decimal	Character	Meaning
00	0	NUL	NULL
01	1	SOH	Start of heading
02	2	STX	Start of text
03	3	ETX	End of text
04	4	EOT	End of transmission
05	5	ENQ	Enquiry
06	6	ACK	Acknowledgment
07	7	BEL	Bell
08	8	BS	Backspace
09	9	HT	Horizontal tab
0A	10	LF	Line feed
0B	11	VT	Vertical tab
0C	12	FF	Form feed
0D	13	CR	Carriage return
0E	14	SO	Shift out
0F	15	SI	Shift in
10	16	DLE	Data link escape
11	17	DC1	Device control 1
12	18	DC2	Device control 2
13	19	DC3	Device control 3
14	20	DC4	Device control 4
15	21	NAK	Negative acknowledgment
16	22	SYN	Synchronous idle
17	23	ETB	End of transmission block
18	24	CAN	Cancel
19	25	EM	End of medium
1A	26	SUB	Substitute
1B	27	ESC	Escape
1C	28	FS	File separator
1D	29	GS	Group separator
1E	30	RS	Record separator
1F	31	US	Unit separator
7F	127	DEL	Delete

ASCII Printable Character Codes[†]

Hex	Decimal	Character	Hex	Decimal	Character	Hex	Decimal	Character
20	32	Space	40	64	@	60	96	‘
21	33	!	41	65	A	61	97	a
22	34	”	42	66	B	62	98	b
23	35	#	43	67	C	63	99	c
24	36	$	44	68	D	64	100	d
25	37	%	45	69	E	65	101	e
26	38	&	46	70	F	66	102	f
27	39	’	47	71	G	67	103	g
28	40	(	48	72	H	68	104	h
29	41	)	49	73	I	69	105	i
2A	42	*	4A	74	J	6A	106	j
2B	43	+	4B	75	K	6B	107	k
2C	44	,	4C	76	L	6C	108	l
2D	45	–	4D	77	M	6D	109	m
2E	46	.	4E	78	N	6E	110	n
2F	47	/	4F	79	O	6F	111	o
30	48	0	50	80	P	70	112	p
31	49	1	51	81	Q	71	113	q
32	50	2	52	82	R	72	114	r
33	51	3	53	83	S	73	115	s
34	52	4	54	84	T	74	116	t
35	53	5	55	85	U	75	117	u
36	54	6	56	86	V	76	118	v
37	55	7	57	87	W	77	119	w
38	56	8	58	88	X	78	120	x
39	57	9	59	89	Y	79	121	y
3A	58	:	5A	90	Z	7A	122	z
3B	59	;	5B	91	[	7B	123	{
3C	60	<	5C	92	\	7C	124	|
3D	61	=	5D	93	]	7D	125	}
3E	62	>	5E	94	^	7E	126	~
3F	63	?	5F	95	_			

[†]Note that 7FH (127 in decimal) is a control character listed on the previous page.

Appendix C

Assembling and Linking Pentium Assembly Language Programs

Objectives

- To present the structure of the standalone assembly language programs used in this book;
- To describe the input and output routines provided with this book;
- To explain the assembly process.

In this appendix, we discuss the necessary mechanisms to write and execute Pentium assembly language programs. We begin by taking a look at the structure of assembly language programs that we use in this book. To make the task of writing assembly language programs easier, we make use of the simplified segment directives provided by the assembler. Section C.1 describes the structure of the standalone assembly language programs used in this book.

Unlike high-level languages, assembly language does not provide a convenient mechanism to do input/output. To overcome this deficiency, we have provided a set of I/O routines to facilitate character, string, and numeric input/output. These routines are described in Section C.2.

Once we have written an assembly language program, we have to transform it into its executable form. Typically, this takes two steps: we use an assembler to translate the source program into what is called an object program and then use a linker to transform the object program into an executable code. Section C.3 gives details of these steps. The appendix concludes with a summary.

```
TITLE     brief title of program      file-name
COMMENT   |
           Objectives:
               Inputs:
              Outputs:
          |
.MODEL SMALL

.STACK    100H            ; defines a 256-byte stack

.DATA
(data go here)

.CODE
.486                      ; not necessary if only 8086
                          ; instructions are used
INCLUDE  io.mac           ; include I/O routines
main  PROC
      .STARTUP            ; setup segments
            .
            .
    (code goes here)
            .
            .
      .EXIT               ; returns control
main  ENDP
      END    main
```

Figure C.1 Structure of the standalone assembly language programs used in this book.

C.1 Structure of Assembly Language Programs

Writing an assembly language program is a complicated task, particularly for a beginner. We make this daunting task simple by hiding unnecessary details. A typical assembly language program consists of three parts. The code part of the program defines the program's functionality by a sequence of assembly language instructions. The code part of the program, after translating it to the machine language code, is placed in the code segment. The data part reserves memory space for the program's data. The data part of the program is mapped to the data segment. Finally, we also need the stack data structure, which is mapped to the stack segment. The stack serves two main purposes: it provides temporary storage and acts as the medium to pass parameters in procedure calls. We use the template shown in Figure C.1 for writing standalone assembly language programs. These are the programs that are written completely in assembly language.

Now let us dissect the statements in this template. This template consists of two types of statements: executable instructions and assembler directives. Executable instructions generate machine code for Pentium to execute when the program is run. Assembler directives, on the other hand, are meant only for the assembler. They provide information to the assembler on the various aspects of the assembly process. In this book, all assembler directives are shown in uppercase letters, and instructions are shown in lowercase.

The TITLE line is optional and when included, usually contains a brief heading of the program and the disk file name. The TITLE information can be up to 128 characters. To understand the purpose of the TITLE directive, you should know that the assembler produces, if you want, a nicely formatted listing file (with extension `.lst`) after the source file has been assembled. In the listing file, each page heading contains the information provided in the TITLE directive.

The COMMENT assembler directive is useful for including several lines of text in assembly language programs. The format of this directive is

```
COMMENT  delimiter [text]
[text]
[text] delimiter [text]
```

where the brackets [ ] indicate optional text. The delimiter is used to delineate the comment block. The delimiter is any nonblank character after the COMMENT directive. The assembler ignores the text following the delimiter until the second occurrence of the delimiter. It also ignores any text following the second delimiter on the same line. We use the COMMENT directive to include objectives of the program and its inputs and outputs. For an example, see `sample.asm` given on page 914.

The .MODEL directive specifies a standard memory configuration for the assembly language program. For our purposes, a small model is sufficient. A restriction of this model is that our program's code should be $\leq$64K, and the total storage for the data should also be $\leq$64K. This directive should precede the .STACK, .DATA, and .CODE directives.

The .STACK directive defines the stack segment to be used with the program. The size of the stack can be specified. By default, we always use a 100H byte (256 bytes or 128 words) stack.

The .DATA directive defines the data segment for the assembly language program. The program's variables are defined here. Chapter 9 discusses various directives to define and initialize variables used in assembly language programs.

The .CODE directive terminates the data segment and starts the code segment. You need to use `.486` only if the code contains instructions of 32-bit processors such as the 80486 and the Pentium. This line is not necessary if the assembly language code uses only the 8086 instructions. The INCLUDE directive causes the assembler to include source code from another file (`io.mac` here). The code

```
main  PROC
      . . .
      . . .
main  ENDP
```

defines a procedure called `main` using the directives PROC (procedure) and ENDP (end procedure).

The last statement uses the END directive for two distinct purposes:

1. By using the label `main`, it identifies the entry point into the program (first instruction of `main` procedure here),
2. It signals the assembler that the end of the source file has been reached.

The choice of `main` in the template is arbitrary. You can use any other name with the restriction that the same name should appear in all three places.

The .STARTUP assembler directive sets up the data and stack segments appropriately. In its place you can write code to set up the data segment yourself. To do this, use the following code:

```
mov     AX,@DATA
mov     DS,AX
```

These two lines initialize the DS register so that it points to the program's data segment. Note that `@DATA` points to the data segment.

To return control from the assembly program, use the `.EXIT` assembler directive. This directive places the code to call the `int 21H` function 4CH to return control. In this directive's place, you can write your own code to call `int 21H`, as shown below:

```
mov     AX,4C00H
int     21H
```

Control is returned to the operating system by interrupt 21H service 4CH. The service required under interrupt 21H is indicated by moving 4CH into the AH register. This service also returns an error code in the AL register. It is good practice to set AL to 0 to indicate normal termination of the program.

C.2 Input/Output Routines

We rarely write programs that do not input and/or output data. High-level languages provide facilities to input and output data. For example, C provides `scanf` and `printf` functions to input and output data, respectively. Typically, high-level languages can read numeric data (integers, floating-point numbers), characters, and strings.

Assembly language, however, does not provide a convenient mechanism to input/output data. The operating system provides some basic services to read and write data, but these are fairly limited. For example, there is no function to read an integer from the keyboard.

Table C.1 Summary of I/O routines defined in `io.mac`

Name	Operand(s)	Operand location	Size	What it does
`PutCh`	Source	Value Register Memory	8 bits	Displays the character located at source
`GetCh`	Destination	Register Memory	8 bits	Reads a character into destination
`nwln`	None	—	—	Displays a carriage return and line feed
`PutStr`	Source	Memory	Variable	Displays the NULL-terminated string at source
`GetStr`	Destination [,`buffer_size`]	Memory	Variable	Reads a carriage-return-terminated string into destination and stores it as a NULL-terminated string. Maximum string length is `buffer_size`−1
`PutInt`	Source	Register Memory	16 bits	Displays the signed 16-bit number located at source
`GetInt`	Destination	Register Memory	16 bits	Reads a signed 16-bit number into destination
`PutLint`	Source	Register Memory	32 bits	Displays the signed 32-bit number located at source
`GetLint`	Destination	Register Memory	32 bits	Reads a signed 32-bit number into destination

In order to facilitate I/O in assembly language programs, it is necessary to write the required procedures. We have written a set of I/O routines to read and display signed integers, characters, and strings. The remainder of this section describes these routines. Each I/O routine call looks like an assembly language instruction. This is achieved by using macros. Each macro call typically expands to several assembly language statements and includes a call to an appropriate procedure. These macros are all defined in the `io.mac` file and actual assembled procedures that perform I/O are in the `io.obj` file. Table C.1 provides a summary of the I/O routines defined in `io.mac`.

C.2.1 Character I/O

Two macros are defined to input and output characters: `PutCh` and `GetCh`. The format of `PutCh` is

```
PutCh    source
```

where source can be any general-purpose 8-bit register, a byte in memory, or a character value. Some examples follow:

```
PutCh    'A'         ; displays character A
PutCh    AL          ; displays the character in AL
PutCh    response    ; displays the byte located in
                     ; memory (labeled response)
```

The format of `GetCh` is

```
GetCh    destination
```

where destination can be either an 8-bit general-purpose register or a byte in memory. Some examples are as follows:

```
GetCh    DH
GetCh    response
```

In addition, a `nwln` macro is defined to display a newline, which sends a carriage return (CR) and a line feed (LF). It takes no operands.

C.2.2 String I/O

`PutStr` and `GetStr` are defined to display and read strings, respectively. The strings are assumed to be in NULL-terminated format. That is, the last character of the string is the NULL ASCII character, which signals the end of the string. Strings are discussed in Chapter 12.

The format of `PutStr` is

```
PutStr    source
```

where source is the name of the buffer containing the string to be displayed. For example,

```
PutStr    message
```

displays the string stored in the buffer `message`. Strings are limited to 80 characters. If the buffer does not contain a NULL-terminated string, a maximum of 80 characters is displayed.

The format of `GetStr` is

```
GetStr    destination [, buffer_size]
```

where destination is the buffer name into which the string from the keyboard is read. The input string can be terminated by a CR. You can also specify the optional `buffer_size` value. If not specified, a buffer size of 81 is assumed. Thus, in the default case, a maximum of 80 characters is read into the string. If a value is specified, `buffer_size`−1 characters are read. The string is stored as a NULL-terminated string. You can backspace to correct input. Here are some examples:

```
GetStr    in_string    ; reads at most 80 characters
GetStr    TR_title,41  ; reads at most 40 characters
```

C.2.3 Numeric I/O

There are four macro definitions for performing integer I/O: two are defined for 16-bit integers and two for 32-bit integers. First we look at the 16-bit integer I/O routines `PutInt` and `GetInt`. The formats of these routines are

```
PutInt    source
GetInt    destination
```

where source and destination can be a 16-bit general-purpose register or the label of a memory word.

`PutInt` displays the signed number at the source. It suppresses all leading 0s. `GetInt` reads a 16-bit signed number into destination. You can backspace while entering a number. The valid range of input numbers is −32,768 to +32,767. If an invalid input (such as typing a nondigit character) or out-of-range number is given, an error message is displayed and the user is asked to type a valid number. Some examples are as follows:

```
PutInt    AX
PutInt    sum
GetInt    CX
GetInt    count
```

Long integer I/O is similar except that the source and destination must be a 32-bit register or a label of a memory doubleword (i.e., 32 bits). For example, if `total` is a 32-bit number in memory, we can display it by

```
PutLint   total
```

and read a long integer from the keyboard into `total` by

```
GetLint   total
```

Some examples that use registers are the following:

```
PutLint   EAX
GetLint   EDX
```

An Example

Program C.1 gives a simple example to demonstrate how some of these I/O routines can be used to facilitate I/O. The program uses the DB (define byte) assembly language directive to declare several strings (lines 11 to 15). All these strings are terminated by 0, which is the ASCII value for the NULL character. Similarly, 16 bytes are allocated for a buffer to store the user name and another byte is reserved for the response. In both cases, ? indicates that the data are not initialized.

Program C.1 An example assembly program

```
 1:  TITLE    An example assembly language program     SAMPLE.ASM
 2:  COMMENT  |
 3:           Objective: To demonstrate the use of some I/O
 4:                      routines and to show the structure
 5:                      of assembly language programs.
 6:              Inputs: As prompted.
 7:  |          Outputs: As per input.
 8:  .MODEL SMALL
 9:  .STACK  100H
10:  .DATA
11:  name_msg      DB  'Please enter your name: ',0
12:  query_msg     DB  'How many times to repeat welcome message? ',0
13:  confirm_msg1  DB  'Repeat welcome message ',0
14:  confirm_msg2  DB  ' times? (y/n) ',0
15:  welcome_msg   DB  'Welcome to Assembly Language Programming ',0
16:
17:  user_name    DB  16 DUP (?)   ; buffer for user name
18:  response     DB  ?
19:
20:  .CODE
21:  INCLUDE io.mac
22:
23:  main  PROC
24:        .STARTUP
25:        PutStr  name_msg       ; prompt user for his/her name
26:        nwln
27:        GetStr  user_name,16   ; read name (max. 15 characters)
28:        nwln
29:  ask_count:
30:        PutStr  query_msg      ; prompt for repeat count
31:        GetInt  CX             ; read repeat count
32:        nwln
33:        PutStr  confirm_msg1   ; confirm repeat count
34:        PutInt  CX             ; by displaying its value
35:        PutStr  confirm_msg2
```

```
36:         GetCh   response        ; read user response
37:         nwln
38:         cmp     response,'y'    ; if 'y', display welcome message
39:         jne     ask_count       ; otherwise, request repeat count
40:   display_msg:
41:         PutStr  welcome_msg     ; display welcome message
42:         PutStr  user_name       ; display the user name
43:         nwln
44:         loop    display_msg     ; repeat count times
45:         .EXIT
46:   main  ENDP
47:         END     main
```

The program requests the name of the user and a repeat count. After confirming the repeat count, it displays a welcome message repeat count times. We use `PutStr` on line 25 to prompt for the user name. The name is read as a string using `GetStr` into the `user_name` buffer. Since we have allocated only 16 bytes for the buffer, the name cannot be more than 15 characters. We enforce this by specifying the optional buffer size parameter in `GetStr` (line 27). The `PutStr` on line 30 requests a repeat count, which is read by `GetInt` on line 31. The confirmation message is displayed by lines 33 to 35. The response of the user `y/n` is read by `GetCh` on line 36. If the response is `y`, the loop (lines 40 to 44) displays the welcome message repeat count times. A sample interaction with the program is shown below:

```
Please enter your name:
Veda
How many times to repeat welcome message? 4
Repeat welcome message 4 times? (y/n) y
Welcome to Assembly Language Programming Veda
Welcome to Assembly Language Programming Veda
Welcome to Assembly Language Programming Veda
Welcome to Assembly Language Programming Veda
```

C.3 Assembling and Linking

Figure C.2 shows the steps involved in converting an assembly language program into an executable program. The source assembly language file (e.g., `sample.asm`) is given as input to the assembler. The assembler translates the assembly language program into an object program (e.g., `sample.obj`). The linker takes one or more object programs (e.g., `sample.obj` and `io.obj`) and combines them into an executable program (e.g., `sample.exe`). The following subsections describe each of these steps in detail.

C.3.1 The Assembly Process

To assemble a program, you need to have an assembler (e.g., TASM.EXE or MASM.EXE). In the remainder of this section, we describe the Turbo assembler TASM. MASM also works in a

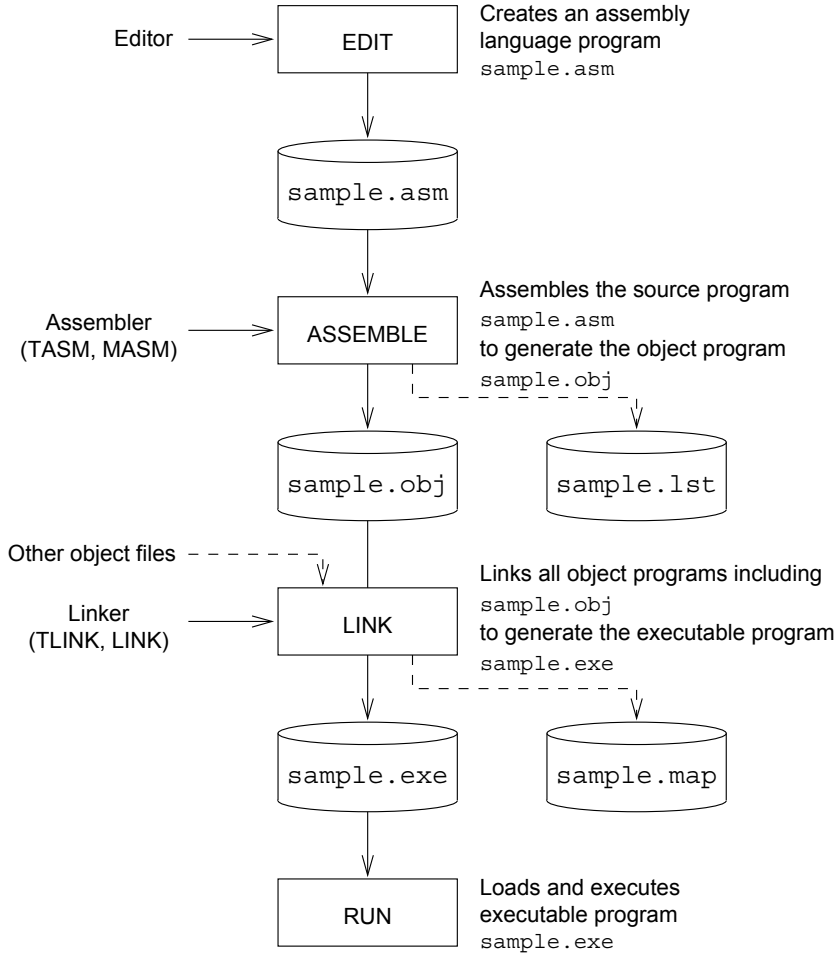


Figure C.2 Assembling, linking, and executing assembly language programs (optional inputs and outputs are shown by dashed lines).

similar way (see your assembler documentation). The general format to assemble an assembly language program is

```
TASM  [options] source-file [,obj-file] [,list-file] [,xref-file]
```

where the specification of fields in [ ] is optional. If we simply specify only the source file, TASM just produces only the object file. Thus, to assemble our example source file `sample.asm`, type

```
TASM  sample
```

You don't have to type the extension. By default, TASM assumes the `.asm` extension. During the assembly process, TASM displays error messages (if any). After successfully assembling the source program, TASM generates an object file with the same file name as the source file but with the `.obj` extension. Thus, in our example, it generates the `sample.obj` file.

If you want the assembler to generate the listing file, you can use

```
TASM  sample,,
```

This produces two files: `sample.obj` and `sample.lst`. The list file contains detailed information about the assembly process, as we show shortly. If you want to use a different file name for the listing file, you have to specify the file name (the extension `.lst` is assumed), as in the following example:

```
TASM  sample,,myprog
```

which generates two files: `sample.obj` and `myprog.lst`.

If the fourth field `xref-file` is specified, TASM generates a listing file containing cross-reference information (discussed shortly).

Options

You can also use command line option `L` to produce the listing file. For example,

```
TASM  /L sample
```

produces `sample.obj` and `sample.lst` files. During the assembly process, TASM displays error messages but does not display the corresponding source lines. You can use option `Z` to force TASM to display the error source lines. Other interesting options are `N` to suppress symbol table information in the listing file, and `ZI` to include complete debugging information for debuggers (such as Turbo Debugger TD). A complete list of options is displayed by typing TASM.

The List File

Program C.2 gives a simple program that reads two signed integers from the user and displays their sum if there is no overflow; otherwise, it displays an error message. The input numbers should be in the range −2,147,483,648 to +2,147,483,647, which is the range of a 32-bit signed number. The program uses `PurStr` and `GetLInt` to prompt and read input numbers (see lines 24, 25 and 29, 30). The sum of the input numbers is computed on lines 34 to 36. If the resulting sum is outside the range of a signed 32-bit integer, the overflow flag is set by the `add` instruction. In this case, the program displays the overflow message (line 40). If there is no overflow, the sum is displayed (lines 46 and 47).

Program C.2 An assembly language program to add two integers `sumprog.asm`

```
 1:  TITLE   Assembly language program to find sum   SUMPROG.ASM
 2:  COMMENT  |
 3:           Objective: To add two integers.
 4:              Inputs: Two integers.
 5:  |           Output: Sum of input numbers.
 6:  .MODEL SMALL
 7:  .STACK  100H
 8:  .DATA
 9:  prompt1_msg  DB  'Enter first number: ',0
10:  prompt2_msg  DB  'Enter second number: ',0
11:  sum_msg      DB  'Sum is: ',0
12:  error_msg    DB  'Overflow has occurred!',0
13:
14:  number1      DD  ?  ; stores first number
15:  number2      DD  ?  ; stores second number
16:  sum          DD  ?  ; stores sum
17:
18:  .CODE
19:  INCLUDE io.mac
20:  .486
21:  main  PROC
22:        .STARTUP
23:        ; prompt user for first number
24:        PutStr  prompt1_msg
25:        GetLint number1
26:        nwln
27:
28:        ; prompt user for second number
29:        PutStr  prompt2_msg
30:        GetLint number2
31:        nwln
32:
33:        ; find sum of two 32-bit numbers
34:        mov     EAX,number1
35:        add     EAX,number2
36:        mov     sum,EAX
37:
38:        ; check for overflow
39:        jno     no_overflow
40:        PutStr  error_msg
41:        nwln
42:        jmp     done
43:
```

```
44:         ; display sum
45:   no_overflow:
46:         PutStr  sum_msg
47:         PutLint sum
48:         nwln
49:   done:
50:         .EXIT
51:   main  ENDP
52:         END     main
```

The list file for the source program `sumprog.asm` is shown in Program C.3. It contains, in addition to the original source code lines, a lot of useful information about the results of the assembly. This additional information includes the actual machine code generated for the executable statements, offsets of each statement, and tables of information about symbols and segments.

The top line of each page consists of a header that identifies the assembler, its version, date, time, and page number. If TITLE is used, the title line is printed on each page of the listing. There are two parts to the listing file: the first part consists of annotated source code, and the second part gives tables of information about the symbols and segments used by the program.

Source Code Lines

The format of the source code lines is as follows:

```
nesting-level  line#  offset  machine-code  source-line
```

`nesting-level`: the level of nesting of "include files" and macros. We discussed macros in Section 9.8 on page 366.

`line#`: the number of the listing file line numbers. These numbers are different from the line numbers in the source file. This can be due to include files, macros, and so on, as shown in Program C.3.

`offset`: a 4-digit hexadecimal offset value of the machine code for the source statement. For example, the offset of the first instruction (line 31) is 0000, and that of the add instruction on line 45 is 0044H. Source lines such as comments do not generate any offset.

`machine-code`: the hexadecimal representation of the machine code for the assembly language instruction. For example, the machine language encoding of

```
    mov     EAX,number1
```

is 66 | A1004B (line 44) and requires 4 bytes (66 is the operand size override prefix). Similarly, the machine language encoding of

```
        jmp     done
```

is EB1990 (line 52), requiring 3 bytes of memory. Again, source code lines such as comments do not generate any machine code for obvious reasons.

`source-line`: a copy of the original source code line. As you can see from Program C.3, the number of bytes required for the machine code depends on the source instruction. When operands are in memory like `number1`, their relative address value is appended with `r` (see line 44) to indicate that the actual value is fixed up by the linker when the segment is combined with other segments (e.g., `io.obj` in our example). You will see an `e` instead of `r` if the symbol is defined externally to the source file (thus available only at link time). For segment values, an `s` is appended to the relative addresses.

Program C.3 The list file for the example assembly program `sumprog.asm`

```
Turbo Assembler  Version 4.0        08/09/97 16:58:59       Page 1
sumprog.ASM
Assembly language program to find sum   SUMPROG.ASM


      1                                 COMMENT  |
      2                                          Objective: To add two integers.
      3                                             Inputs: Two integers.
      4                                 |           Output: Sum of input numbers.
      5     0000                        .MODEL SMALL
      6     0000                        .STACK  100H
      7     0000                        .DATA
      8     0000  45 6E 74 65 72 20 66+ prompt1_msg  DB  'Enter first number: ',0
      9           69 72 73 74 20 6E 75+
     10           6D 62 65 72 3A 20 00
     11     0015  45 6E 74 65 72 20 73+ prompt2_msg  DB  'Enter second number: ',0
     12           65 63 6F 6E 64 20 6E+
     13           75 6D 62 65 72 3A 20+
     14           00
     15     002B  53 75 6D 20 69 73 3A+ sum_msg      DB  'Sum is: ',0
     16           20 00
     17     0034  4F 76 65 72 66 6C 6F+ error_msg    DB  'Overflow has occurred!',0
     18           77 20 68 61 73 20 6F+
     19           63 63 75 72 72 65 64+
     20           21 00
     21
     22     004B  ????????              number1      DD  ?  ; stores first number
     23     004F  ????????              number2      DD  ?  ; stores second number
     24     0053  ????????              sum          DD  ?  ; stores sum
     25
     26     0057                        .CODE
     27                                 INCLUDE io.mac
1    28
1    29
```

```
     30                                 .486
     31     0000                        main  PROC
     32                                       .STARTUP
     33                                       ; prompt user for first number
     34                                       PutStr  prompt1_msg
     35                                       GetLint number1
     36                                       nwln
     37
     38                                       ; prompt user for second number
     39                                       PutStr  prompt2_msg
     40                                       GetLint number2
     41                                       nwln
     42
     43                                       ; find sum of two 32-bit numbers
     44     0040  66| A1 004Br                mov     EAX,number1
     45     0044  66| 03 06 004Fr             add     EAX,number2
     46     0049  66| A3 0053r                mov     sum,EAX
     47
     48                                       ; check for overflow
     49     004D  71 12 90 90                 jno     no_overflow
     50                                       PutStr  error_msg
     51                                       nwln
     52     005E  EB 19 90                    jmp     done
     53
     54                                       ; display sum
     55     0061                        no_overflow:
     56                                       PutStr  sum_msg
     57                                       PutLint sum
Turbo Assembler  Version 4.0        08/09/97 16:58:59       Page 2
sumprog.ASM
Assembly language program to find sum   SUMPROG.ASM


     58                                       nwln
     59     0079                         done:
     60                                       .EXIT
     61     007D                        main  ENDP
     62                                       END     main
Turbo Assembler  Version 4.0        08/09/97 16:58:59       Page 3
Symbol Table
Assembly language program to find sum   SUMPROG.ASM


Symbol Name                       Type   Value

??DATE                            Text   "08/09/97"
??FILENAME                        Text   "sumprog "
??TIME                            Text   "16:58:59"
??VERSION                         Number 0400
@32BIT                            Text   0
@CODE                             Text   _TEXT
```

```
@CODESIZE                       Text   0
@CPU                            Text   1F1FH
@CURSEG                         Text   _TEXT
@DATA                           Text   DGROUP
@DATASIZE                       Text   0
@FILENAME                       Text   SUMPROG
@INTERFACE                      Text   00H
@MODEL                          Text   2
@STACK                          Text   DGROUP
@STARTUP                        Near   _TEXT:0000
@WORDSIZE                       Text   4
DONE                            Near   _TEXT:0079
ERROR_MSG                       Byte   DGROUP:0034
MAIN                            Near   _TEXT:0000
NO_OVERFLOW                     Near   _TEXT:0061
NUMBER1                         Dword  DGROUP:004B
NUMBER2                         Dword  DGROUP:004F
PROC_GETCH                      Near   _TEXT:---- Extern
PROC_GETINT                     Near   _TEXT:---- Extern
PROC_GETLINT                    Near   _TEXT:---- Extern
PROC_GETSTR                     Near   _TEXT:---- Extern
PROC_NWLN                       Near   _TEXT:---- Extern
PROC_PUTCH                      Near   _TEXT:---- Extern
PROC_PUTINT                     Near   _TEXT:---- Extern
PROC_PUTLINT                    Near   _TEXT:---- Extern
PROC_PUTSTR                     Near   _TEXT:---- Extern
PROMPT1_MSG                     Byte   DGROUP:0000
PROMPT2_MSG                     Byte   DGROUP:0015
SUM                             Dword  DGROUP:0053
SUM_MSG                         Byte   DGROUP:002B
TEMP                            Byte   _TEXT:---- Extern

Macro Name

GETCH
GETINT
GETLINT
GETSTR
NWLN
PUTCH
PUTINT
PUTLINT
PUTSTR
Turbo Assembler  Version 4.0        08/09/97 16:58:59       Page 4
Symbol Table
Assembly language program to find sum   SUMPROG.ASM



Groups & Segments               Bit Size Align  Combine Class

DGROUP                          Group
```

```
  STACK                           16  0100 Para   Stack   STACK
  _DATA                           16  0057 Word   Public  DATA
_TEXT                             16  007D Word   Public  CODE
```

Symbol Table

The second part of the listing file consists of two tables of information. The first one lists all the symbols used in the program in alphabetical order. These include the variables and labels used in the program. For each symbol, the symbol table gives its type and value. For example, `number1` and `number2` are words with offsets 4BH and 4FH, respectively, in the DGROUP segment group. This segment group has `_DATA` and `STACK` segments.

The I/O procedures (`PROC_GETCH`, etc.) are near procedures that are defined as external in `io.mac`. Procedures are discussed in Chapter 10. The object code for these procedures is available at the time of linking (`io.obj` file). The macros listed are defined in `io.mac`.

If the fourth field `xref-file` on the TASM command line is specified, the listing file would contain cross-reference information for each symbol. The cross-reference information gives where (i.e., line number) the symbol was defined and the line numbers of all the lines in the program on which that symbol was referenced.

Group and Segment Table

The other table gives information on groups and segments. Segment groups do not have any attributes and are listed with the segments making up the group. For example, the DGROUP consists of `_DATA` and `STACK` segments. Segments, however, have attributes. For each segment, five attributes are listed.

Bit: Gives the data size, which is 16 in our case.

Size: Indicates the segment size in hex. For example, the STACK segment is 100H (i.e., 256) bytes long.

Align: Indicates the type of alignment. This refers to the memory boundaries that a segment can begin. Some alignment types are as follows:

BYTE	Segment can begin at any address;
WORD	Segment can begin only at even addresses;
PARA	Segment can begin only at an address that is a multiple of 16 (para = 16 bytes).

For example, `STACK` is para-aligned, whereas `_DATA` and `_TEXT` are word-aligned.

Combine: Specifies how segments of the same name are combined. With the PUBLIC combine type, identically named segments are concatenated into a larger segment. The combine type STACK is special and can only be used for the stack.

Class: Refers to the segment class, for example, CODE, DATA, or STACK. The linker uses this information to order segments.

C.3.2 Linking Object Files

Linker is a program that takes one or more object programs as its input and produces an executable program. In our example, since I/O routines are defined separately, we need two object files—`sample.obj` and `io.obj`—to generate the executable file `sample.exe`. To do this, we use the command

```
TLINK sample io
```

The syntax of TLINK is given by

```
TLINK [options] obj-files,exe-file,map-file,lib-file
```

where `obj-files` is a list of object files to be linked, and `exe-file` is the name of the executable file. If no executable file name is given, the name of the first object file specified is used with the `.exe` extension. TLINK, by default, also generates a map file. If no map file name is given on the command line, the first object file name is used with the `.map` extension. `lib-file` specifies library files, and we do not discuss them here.

The map file provides information on segments. The map file generated for the `sample` program is shown below:

```
 Start  Stop   Length Name               Class

 00000H 0037FH 00380H _TEXT             CODE
 00380H 0053FH 001C0H _DATA             DATA
 00540H 0063FH 00100H STACK             STACK

Program entry point at 0000:0000
```

For each segment, it gives the starting and ending addresses along with the length of the segment in bytes, its name, and its class. For example, the `CODE` segment is named `_TEXT` and starts at address 0 and ends at 37FH. The length, therefore, is 380H.

If you intend to debug your program using Turbo Debugger, you should use `V` in order to link the necessary symbolic information. For example, the `sample.obj` object program, along with `io.obj`, can be linked by

```
TLINK /V sample io
```

You have to make sure that the `ZI` option has been used during the assembly.

C.4 Summary

Assembly language programs consist of three parts: stack, data, and code segments. These three segments can be defined using simplified segment directives provided by both TASM and

MASM assemblers. By means of simple examples, we have seen the structure of a typical standalone assembly language program.

Since assembly language does not provide a convenient mechanism to do input/output, we defined a set of I/O routines to help us in performing simple character, string, and numeric input and output. The numeric I/O routines provided can input/output both 16-bit and 32-bit signed integers.

To execute an assembly language program, we have to first translate it into an object program by using an assembler. Then we have to pass this object program, along with any other object programs needed by the program, to a linker to produce an executable program. Both the assembler and linker generate additional files that provide information on the assembly and link processes.

C.5 Exercises

C–1 What is the purpose of the TITLE directive?

C–2 How is the stack defined in the assembly language programs used in this book?

C–3 In the assembly language program structure used in this book, how are the data and code parts specified?

C–4 What is meant by a "standalone" assembly language program?

C–5 What is an assembler? What is the purpose of it?

C–6 What files are generated by your assembler? What is the purpose of each of these files?

C–7 What is the function of the linker? What is the input to the linker?

C–8 Why is it necessary to define our own I/O routines in assembly language?

C–9 What is a NULL-terminated string?

C–10 Why is buffer size specification necessary in `GetStr` but not in `PutStr`?

C–11 What happens if the buffer size parameter is not specified in `GetStr`?

C–12 What happens if the buffer specified in `PutStr` does not contain a NULL-terminated string?

C–13 What is the range of numbers that `GetInt` can read from the keyboard? Give an explanation for the range.

C–14 Repeat the last exercise for `GetLint`.

C.6 Programming Exercises

C–P1 Write an assembly language program to explore the behavior of the various character and string I/O routines. In particular, comment on the behavior of the `GetStr` and `PutStr` routines.

C–P2 Write an assembly language program to explore the behavior of the various numeric I/O routines. In particular, comment on the behavior of the `GetInt` and `GetLint` routines.

C–P3 Modify the `sample.asm` by deliberately introducing errors into the program. Assemble the program and see the type of errors reported by your assembler. Also, generate the listing file and briefly explain its contents.

C–P4 Assemble the `sample.asm` program to generate cross-reference information. Comment on how this information is presented by your assembler.

Appendix D

Debugging Assembly Language Programs

Objectives

- To present some basic strategies to debug assembly language programs;
- To describe the DOS debugger DEBUG;
- To explain the basic features of the Turbo Debugger (TD);
- To provide a brief discussion of the Microsoft debugger (CodeView).

Debugging assembly language programs is more difficult and time-consuming than debugging high-level language programs. However, the fundamental strategies that work for high-level languages also work for assembly language programs. Section D.1 gives a discussion of these strategies. Since you are familiar with debugging in a high-level language, this discussion is rather brief.

The following three sections discuss three popular debuggers. Although the DOS DEBUG is a line-oriented debugger, the other two—Turbo Debugger and CodeView—are window-oriented and are much better. All three share some basic commands required to support debugging assembly language programs.

Our goal in this appendix is to introduce the three debuggers briefly, as the best way to get familiar with these debuggers is to try them. We use a simple example to explain some of the commands of DEBUG (in Section D.2) and Turbo Debugger (in Section D.3). Since CodeView is similar in spirit to the Turbo Debugger, we give only a brief overview of it in Section D.4. The appendix concludes with a summary.

D.1 Strategies to Debug Assembly Language Programs

Programming is a complicated task. Very few real-life programs are ever written that work perfectly the very first time. Loosely speaking, a program can be thought of as mapping a set of input values to a set of output values. The functionality of the mapping performed by a program is given as the specification for the programming task. It goes without saying that when the program is written, it should be verified to meet the specifications. In programming parlance, this activity is referred to as testing and validating the program.

Testing a program itself is a complicated task. Typically, test cases, selected to validate the program, should test each possible path in the program, boundary cases, and so on. During this process, errors ("bugs") are discovered. Once a bug is found, it is necessary to find the source code causing the error and fix it. This process is known by its colorful name, *debugging*.

Debugging is not an exact science. We have to rely on our intuition and experience. However, there are tools that can help us in this process. We look at three such tools in this chapter: DEBUG, Turbo Debugger TD, and Microsoft CodeView.

Finding bugs in a program is very much dependent on the individual program. Once an error is detected, there are some general ways of locating the source code lines causing the error. The basic principle that helps us in writing the source program in the first place—the divide and conquer technique—is also useful in the debugging process. Structured programming methodology facilitates debugging greatly.

A program typically consists of several modules, where each module may have several procedures. When developing a program, it is best to do incremental development. In this methodology, a single or a few procedures are added to the program to add some specific functionality and test it before adding other functions to the program. In general, it is a bad idea to write the whole program and start the testing process, unless the program is "small." The best strategy is to write code that has as few bugs as possible. This can be achieved by using pseudocode and verifying the logic of the pseudocode even before we attempt to translate it into the assembly language program. This is a good way of catching many of the logical errors and saves a lot of debugging time. Never write an assembly language code with the pseudocode in your head! Furthermore, don't be in a hurry to write some assembly code that appears to work. This is short-sighted, as you will end up spending more time in the debugging phase.

To isolate a bug, program execution should be observed in slow motion. Most debuggers provide a command to execute programs in single-step mode. In this mode, the program executes one statement at a time and pauses. Then we can examine contents of registers, data in memory, stack contents, and the like. In this mode, a procedure call is treated as a single statement, and the entire procedure is executed before pausing the program. This is useful if you know that the called procedure works correctly. Debuggers also provide another command to trace even the statements of procedure calls, which is useful for testing procedures.

Often we know that some parts of the program work correctly. In this case, it is a sheer waste of time to single-step or trace the code. What we would like is to execute this part of the program and then stop for more careful debugging (perhaps by single-stepping). Debuggers provide commands to set up breakpoints and to execute up to a breakpoint. Another helpful

feature that most debuggers provide is the watch facility. By using watches, it is possible to monitor the state (i.e., values) of the variables in the program as the execution progresses.

In the following three sections, we discuss three debuggers and how they are useful in debugging the program `addigits.asm` discussed in Chapter 9. We limit our discussion to 16-bit segments and operands. The program used in our debugging sessions is shown in Program D.1. This program does not use the `.STARTUP` and `.EXIT` assembler directives. As explained in Appendix C, we use

```
mov     AX,@DATA
mov     DS,AX
```

in place of the `.STARTUP` directive and

```
mov     AX,4C00H
int     21H
```

in place of the `.EXIT` directive.

Program D.1 An example program used to explain debugging

```
 1:  TITLE   Add individual digits of a number   ADDIGITS.ASM
 2:  COMMENT |
 3:          Objective: To find the sum of individual digits of
 4:                     a given number. Shows character to binary
 5:                     conversion of digits.
 6:              Input: Requests a number from keyboard.
 7:  |          Output: Prints the sum of the individual digits.
 8:  DOSSEG
 9:  .MODEL SMALL
10:  .STACK 100H
11:  .DATA
12:  number_prompt  DB  'Please type a number (<10 digits): ',0
13:  out_msg        DB  'The sum of individual digits is: ',0
14:  number         DB  11 DUP (?)
15:  .CODE
16:  INCLUDE io.mac
17:  main    PROC
18:          mov     AX,@DATA     ; initialize DS
19:          mov     DS,AX
20:          PutStr  number_prompt  ; request an input number
21:          GetStr  number,11    ; read input number as a string
22:          nwln
23:          mov     BX,OFFSET number  ; BX := address of number
24:          sub     DX,DX        ; DX := 0 -- DL keeps the sum
25:  repeat_add:
```

```
26:          mov     AL,[BX]       ; move the digit to AL
27:          cmp     AL,0          ; if it is the NULL character
28:          je      done          ;  sum is done
29:          and     AL,0FH        ; mask off the upper 4 bits
30:          add     DL,AL         ; add the digit to sum
31:          inc     BX            ; increment BX to point to next digit
32:          jmp     repeat_add    ;  and jump back
33:  done:
34:          PutStr  out_msg
35:          PutInt  DX            ; write sum
36:          nwln
37:          mov     AX,4C00H      ; return to DOS
38:          int     21H
39:  main    ENDP
40:          END   main
```

D.2 DEBUG

DEBUG is invoked by

```
DEBUG  file_name
```

For example, to debug the `addigits` program, we can use

```
DEBUG  addigits.exe
```

DOS loads DEBUG into memory, which in turn loads `addigits.exe`. It is necessary to enter the extension `.exe`, as DEBUG does not assume any extension. DEBUG displays a hyphen (-) as a prompt. At this prompt, it can accept one of several commands. Table D.1 shows some of the commands useful in debugging programs.

D.2.1 Display Group

U (Unassemble)

This command unassembles the next 32 bytes. The general format is

```
U [address]        or        U [range]
```

If no address is specified in the command, the next 32 bytes since the last U command are unassembled. If there is no U command, the default address CS:IP is used. The address should be specified in hex. The range can be specified either by giving a start and end address, or by giving a start address and length in bytes. When specifying length, the prefix `L` should be used, as shown in the following example:

Table D.1 Summary of DEBUG commands

Command	Function
Display Commands:	
`U`	Unassembles next 32 bytes
`U address`	Unassembles next 32 bytes at `address`
`U range`	Unassembles the bytes in the specified `range`
`D`	Displays the next 128 bytes of memory in hex and ASCII
`D address`	Displays the next 128 bytes of memory at `address`
`D range`	Displays the contents of memory in the specified `range`
`R`	Displays the contents of all registers and shows the next instruction
`R register`	Displays the contents of `register` and accepts hex data to update `register`
`E address`	Displays the contents of the memory location specified by `address`
`E address value-list`	Copies the hex data from `value-list` into memory from `CS:address`
Execution Commands:	
`T`	Traces (i.e., single-step mode) execution; executes one instruction and displays the register contents and the next instruction
`T count`	Executes next `count` instructions
`T =address`	Executes the instruction at `CS:address`
`T =address count`	Executes `count` instructions at `CS:address`
`P`	Like trace but proceeds through `call`, `loop`, `int`
`P count`	Proceeds through the next `count` statements
`P =address`	Executes the statement at `CS:address`
`P =address count`	Executes `count` statements at `CS:address`
`G`	Executes program to completion or until a breakpoint is encountered
`G bkpt-address`	Executes program until the breakpoint specified by `bkpt-address`
`G =address bkpt-address`	Executes program until the breakpoint specified by `bkpt-address` starting from `address`
Miscellaneous Commands:	
`L`	Reloads program after termination
`Q`	Quits DEBUG

```
U           ; unassembles the next 32 bytes
U 3B        ; unassembles 32 bytes from CS:3BH
U 3B 4B     ; unassembles from CS:3BH to CS:4BH
U 3B L10    ; unassembles 16 (= 10H) bytes from CS:3BH
```

Note that in the last example, length is specified as `L10`, where 10H = 16D is the length.

D (Display or Dump)

This command displays the contents of the specified memory locations both in hex and ASCII. The general format is similar to that of the U command and is given by

```
D [address]        or        D [range]
```

The default segment is the segment pointed by DS and the default range is 128 (i.e., 80H) bytes.

```
D           ; displays the next 128 bytes from last display
D CS:0      ; displays 128 bytes from CS:0
D 10 17     ; displays from DS:10H to DS:17H
D 3B LB     ; displays 11 (= BH) bytes from DS:3BH
```

E (Enter)

This command can be used to enter data. The general format is

```
E address        or        E address values
```

If the first format is used (i.e., with no values), it displays the contents of the addressed location. The default segment is the data segment pointed by DS. For example,

```
E 12
```

displays the contents of DS:12H. In the second format, the list of specified values replaces the contents of the addressed memory locations. For example,

```
E 46 31 32 33
```

changes the contents of memory locations 46H through 48H to 31H, 32H, and 33H, respectively. We can also do the same with the following command:

```
E 46 '123'
```

The same command can be used to replace machine code. For example,

```
E CS:5 8B D8
```

replaces the machine code by `8BD8`, which represents

```
mov     BX,AX
```

R (Register)

This command displays the contents of registers and the next instruction. The general format is

```
R       or       R register
```

If no register is specified, it displays the contents of all registers, including the flags, instruction pointer, and segment registers. The flags register contents are displayed as follows:

Flag	Set	Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary carry	AC	NA
Parity	PE	PO
Carry	CY	NC

When a register name is specified in the command, it displays the contents of the register and prompts (displays ':') for a replacement value. For example,

```
-R AX
AX 0000    ; displays the contents of AX (here 0000)
:7FFF      ; prompts for a replacement value
           ; here we want to write 7FFFH into AX
```

modifies AX to 7FFFH. Simply type return to keep the register contents. We can also use this command to modify the IP register.

D.2.2 Execution Group

T (Trace)

This command executes the program in single-step mode; after the execution, it displays the contents of the registers and the next instruction. The general format is

```
T    or    T count    or
T =address    or    T =address count
```

If a count value is specified, it traces count instructions. It displays contents of registers and the next instruction after the execution of each instruction. If an address is specified, tracing starts at the specified address. Here are some examples:

```
T =5D      ; trace the instruction at CS:5DH
T 3        ; trace the next 3 instructions
T =5D 3    ; trace 3 instructions from CS:5DH
```

P (Proceed)

This is similar to trace except it considers an interrupt call (`int`), procedure call (`call`), loop, and so on, as single instructions. Normally this command is used unless we want to debug a procedure, interrupt routine, and the like.

G (Go)

This command executes a program to a specified breakpoint. The format is

```
G   or   G bkpt-address   or
G =address bkpt-address
```

This command is useful in setting breakpoints. We can specify up to 10 breakpoint addresses. If the optional start address (`=address`) is given, execution begins from this address. This, for example, is useful in debugging a procedure or a part of the program, without executing it from the beginning. Some examples are given below:

```
G             ; execute program to completion
G 31          ; execute up to CS:31H
G =31 45      ; execute from CS:31H to CS:45H
```

D.2.3 Miscellaneous Group

The other two commands in Table D.1 are useful for reloading the program (`L`) and exiting the DEBUG (`Q`).

D.2.4 An Example

A sample DEBUG run on `addigits.exe` is shown in Program D.2. The U command on line 2 displays the code by unassembling the first 32 bytes. A drawback with this is that there is no symbolic information. For example,

```
mov     AX,@DATA
```

is displayed as

```
mov     AX,3F09
```

where 3F09 (in hex) is the data segment value. Similarly, procedure calls include the offset values but not the procedure names. This deficiency is remedied by the other two debuggers.

Notice that the code shown here does not exactly correspond to the code of Program D.1. The reason is that each macro call (such as `PutStr`, `GetStr`, and `nwln`) is expanded by using the macro definitions in `io.mac`. For example, the `PutStr` macro call is expanded by the four lines of code (lines 5 to 8). Using symbolic information, we can write these four lines of code as

```
push   AX
mov    AX,OFFSET number_prompt
call   proc_PutStr
pop    AX
```

As discussed in Appendix C, these macros are defined in `io.mac`. The `GetStr` macro is expanded to lines 9 to 15 and `nwln` to lines 16 to 18.

Now let us examine the data segment contents. In order to use the default DS register, we have to set up this register to point to our data segment. This is done by the first two lines of the code. One way to execute these two lines of code is to use the T command (line 20). It makes no difference whether we use the P or T command, as there are no procedure calls or loop instructions. Note that the trace command executes in single-step mode. Thus, after executing each instruction, it displays the contents of the registers, status of the flags, and the next instruction to be executed. From line 27, we can see that DS is initialized to the data segment.

Now we can use the D command (line 29) to display the first 128 bytes starting at offset 0. The data segment contains the two message strings

```
Please type a number (<10 digits):
The sum of individual digits is:
```

and the storage space for `number` starts after these two message strings at 3F09:0046. Since we have not initialized it, the contents do not matter at this point.

Now let us execute the program until after reading an input number. That is, we set up a breakpoint at the instruction

```
mov    BX,0046
```

at offset 001FH. We can do this by using the G command on line 38. The prompt and the input number are shown on line 39. At the breakpoint, it displays the contents of the registers, flags, and the next instructions, as in the trace command we have seen before. Although the G command allows us to set up breakpoints in the program, the other two symbolic debuggers provide a much better screen-oriented user interface, as we show later in this appendix.

Now let us verify that the input has been read properly. We use the D command

```
D 46 LB
```

on line 44 to examine the contents of `number`. In this D command, we are not only specifying the address (46H), but also indicating that 11 (=BH) bytes are to be displayed. Thus, we just see the contents (11 bytes) of `number` (lines 45 and 46).

Let us suppose that we want to check the logic of the loop (lines 25 to 32 in Program D.1). We can do this by executing the loop in single-step mode using the T command on line 47. This gives us an opportunity to check the logic one instruction at a time. An interesting point is that, on line 55, when we are using the indirect addressing mode, it displays the address and its contents:

```
DS:0046=31
```

At the end of the loop, DX = 1, which is what it should be for the given input.

Having checked the logic of the loop, let us run the whole loop without any interruption. This is done by setting a breakpoint using the G command on line 80. (In this example, it is useful to have the list file handy to know the offset values of the code at various points.) This breakpoint is set at line 34 of Program D.1. We note that the sum in the DX register is the correct value (2DH = 45D) for the input given in this sample run.

The rest of the DEBUG output is straightforward to follow. Notice that after the program has terminated, we have used the L command to reload the application for another execution, this time without any breakpoints. Finally, on line 100, we have used the Q command to exit DEBUG.

Program D.2 A sample DEBUG session

```
 1:  A:\>debug addigits.exe
 2:  -U
 3:  3ED1:0000 B8093F        MOV AX,3F09
 4:  3ED1:0003 8ED8          MOV DS,AX
 5:  3ED1:0005 50            PUSH AX
 6:  3ED1:0006 B80000        MOV AX,0000
 7:  3ED1:0009 E85600        CALL 0062
 8:  3ED1:000C 58            POP AX
 9:  3ED1:000D 51            PUSH CX
10:  3ED1:000E B90B00        MOV CX,000B
11:  3ED1:0011 50            PUSH AX
12:  3ED1:0012 B84600        MOV AX,0046
13:  3ED1:0015 E88101        CALL 0199
14:  3ED1:0018 58            POP AX
15:  3ED1:0019 59            POP CX
16:  3ED1:001A 50            PUSH AX
17:  3ED1:001B E83500        CALL 0053
18:  3ED1:001E 58            POP AX
19:  3ED1:001F BB4600        MOV BX,0046
20:  -T 2
21:
22:  AX=3F09  BX=0000  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
23:  DS=3EC1  ES=3EC1  SS=3F20  CS=3ED1  IP=0003   NV UP EI PL NZ NA PO NC
24:  3ED1:0003 8ED8          MOV DS,AX
25:
26:  AX=3F09  BX=0000  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
27:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0005   NV UP EI PL NZ NA PO NC
28:  3ED1:0005 50            PUSH AX
29:  -D 0
30:  3F09:0000  50 6C 65 61 73 65 20 74-79 70 65 20 61 20 6E 75   Please type a nu
31:  3F09:0010  6D 62 65 72 20 28 3C 31-30 20 64 69 67 69 74 73   mber (<10 digits
32:  3F09:0020  29 3A 20 00 54 68 65 20-73 75 6D 20 6F 66 20 69   ): .The sum of i
33:  3F09:0030  6E 64 69 76 69 64 75 61-6C 20 64 69 67 69 74 73   ndividual digits
34:  3F09:0040  20 69 73 3A 20 00 00 00-00 00 00 00 00 00 00 00    is: ............
35:  3F09:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
36:  3F09:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
37:  3F09:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
38:  -G 1F
39:  Please type a number (<10 digits): 1234567890
40:
41:  AX=3F09  BX=0000  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
```

```
 42:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=001F   NV UP EI PL NZ NA PO NC
 43:  3ED1:001F BB4600        MOV BX,0046
 44:  -D 46 LB
 45:  3F09:0040                   31 32-33 34 35 36 37 38 39 30         1234567890
 46:  3F09:0050  00                                                    .
 47:  -T 8
 48:
 49:  AX=3F09  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
 50:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0022   NV UP EI PL NZ NA PO NC
 51:  3ED1:0022 2BD2          SUB DX,DX
 52:
 53:  AX=3F09  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
 54:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0024   NV UP EI PL ZR NA PE NC
 55:  3ED1:0024 8A07          MOV AL,[BX]                           DS:0046=31
 56:
 57:  AX=3F31  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
 58:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0026   NV UP EI PL ZR NA PE NC
 59:  3ED1:0026 3C00          CMP AL,00
 60:
 61:  AX=3F31  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
 62:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0028   NV UP EI PL NZ NA PO NC
 63:  3ED1:0028 7407          JZ 0031
 64:
 65:  AX=3F31  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
 66:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002A   NV UP EI PL NZ NA PO NC
 67:  3ED1:002A 240F          AND AL,0F
 68:
 69:  AX=3F01  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
 70:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002C   NV UP EI PL NZ NA PO NC
 71:  3ED1:002C 02D0          ADD DL,AL
 72:
 73:  AX=3F01  BX=0046  CX=04EC  DX=0001  SP=0100  BP=0000  SI=0000  DI=0000
 74:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002E   NV UP EI PL NZ NA PO NC
 75:  3ED1:002E 43            INC BX
 76:
 77:  AX=3F01  BX=0047  CX=04EC  DX=0001  SP=0100  BP=0000  SI=0000  DI=0000
 78:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002F   NV UP EI PL NZ NA PE NC
 79:  3ED1:002F EBF3          JMP 0024
 80:  -G 31
 81:
 82:  AX=3F00  BX=0050  CX=04EC  DX=002D  SP=0100  BP=0000  SI=0000  DI=0000
 83:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0031   NV UP EI PL ZR NA PE NC
 84:  3ED1:0031 50            PUSH AX
 85:  -G45
 86:  The sum of individual digits is: 45
 87:
 88:  AX=3F00  BX=0050  CX=04EC  DX=002D  SP=0100  BP=0000  SI=0000  DI=0000
 89:  DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0045   NV UP EI PL NZ NA PO NC
 90:  3ED1:0045 B8004C        MOV AX,4C00
 91:  -G
 92:
 93:  Program terminated normally
 94:  -L
 95:  -G
 96:  Please type a number (<10 digits): 456
 97:  The sum of individual digits is: 15
 98:
 99:  Program terminated normally
100:  -Q
101:
102:  A:\>
```

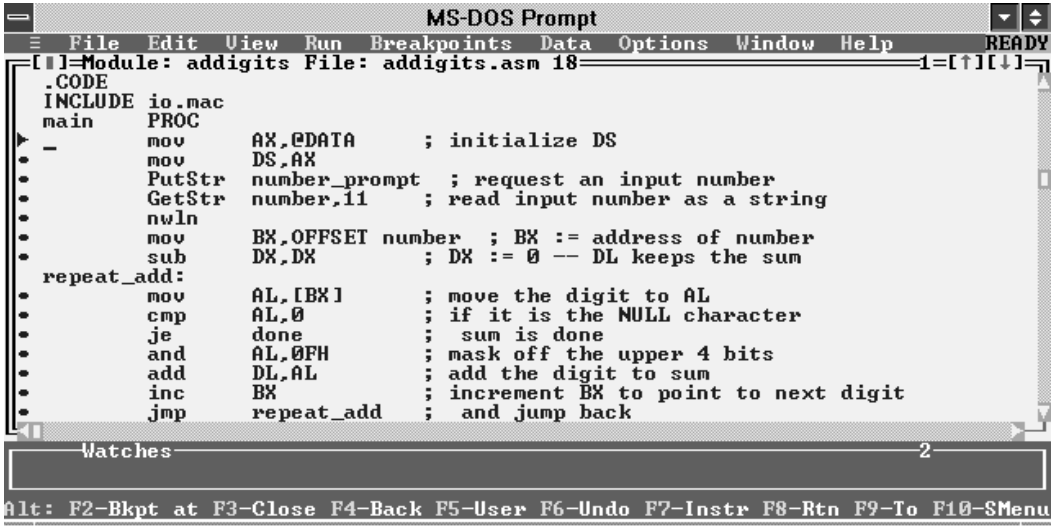


Figure D.1 TD window at the start of `addigits.asm` program.

D.3 Turbo Debugger TD

Turbo Debugger is a window-oriented debugger that facilitates symbolic debugging at the source-code level. TD can be used to debug programs written in high-level languages such as C as well as in assembly language. In this section, we briefly discuss some of the features of TD relevant to debugging assembly language programs.

In order for TD to use symbolic information during debugging, we have to assemble our program with the ZI option and link it with the V option. For example, to debug `addigits.asm`, we use the following commands to prepare the program:

```
TASM  /zi  addigits
TLINK  /v  addigits io
```

The Turbo Debugger can then be invoked by

```
TD  addigits
```

Figure D.1 shows the screen that we would see after invoking TD as indicated. The screen consists of a menu bar (called main menu) at the top, and a quick reference help line at the bottom. In addition, it displays two windows: a module window and a watches window. Each window has a number associated with it. The window number appears in the upper right-hand corner of the window. For example, the module window is window 1 and the watches window is window 2. The active window, the module window in Figure D.1, has a double-line border around it and inactive windows have a single-line border (e.g., see the watches window).

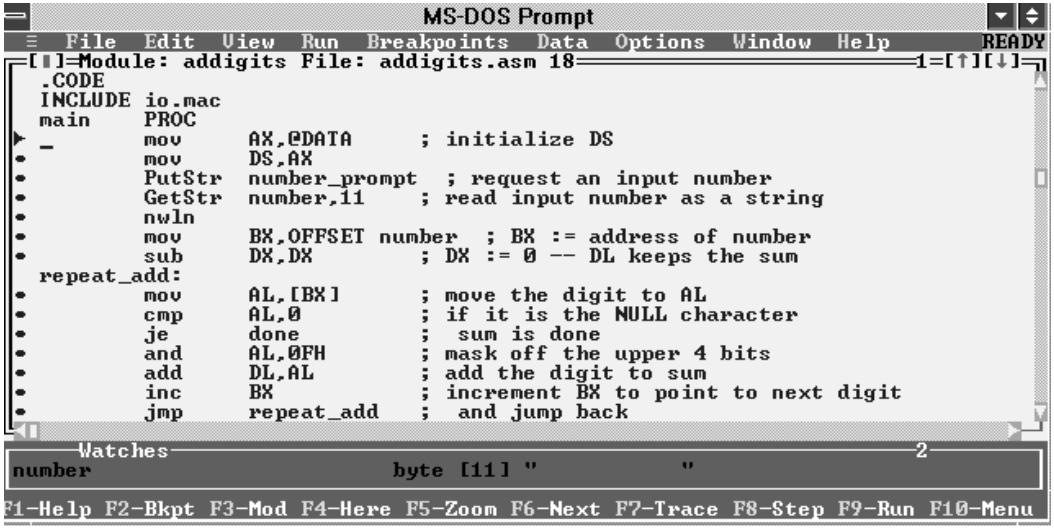


Figure D.2 TD window after adding `number` to watch list.

The module window shows the source program. The arrow at the left points to the next instruction to be executed. Since we haven't yet run the program, the arrow points to the first line of the `main` procedure in Figure D.1.

We can make an inactive window active by pressing Alt-x, where x is the window number. For example, Alt-2 makes the watches window active.

The main menu can be activated by F10. Press carriage return to open the selected pull-down menu. We can then use the arrow keys to navigate the menu items. Here we take a brief look at the `View` and `Run` menu options.

The `View` pull-down menu provides several options to view the status of the program. Some of the options available are listed in Table D.2.

As we indicated in Section D.1, watches are useful to monitor the state of a set of variables as the program execution progresses. In Turbo Debugger, a variable or an expression can be added to the watch list by using `add watch` in the Data menu (Ctrl-F7). Figure D.2 shows the watches window when the variable `number` is added to the watch list. Notice that the TD shows the name of the variable, its type, and contents. Now, for example, we can test the initial part of our program by placing a breakpoint after reading the input number by `GetStr`. This can be done by using an option under the `Run` menu, which we discuss next.

Program execution is controlled by the `Run` menu. Some of the options in this menu are shown in Table D.3. Now let us execute the program until

```
mov     BX,OFFSET number
```

One way is to move the cursor to this line and press F4. This execution prompts us for a number (we have given 1,234,567,890 as input in this example execution) that is stored in variable

Table D.2 Selected `View` menu options

Breakpoints	Displays a list of breakpoints set in the program
Stack	Displays the active procedures
Watches	Displays the values of the variables and expressions in the watch list
Variables	Shows the names and values of all variables accessible from current location of the program
CPU	Shows the status of the program (discussed in text)
Dump	Shows the contents of a part of memory (similar to DEBUG's Dump command)
Register	Shows the contents of all registers including the flags

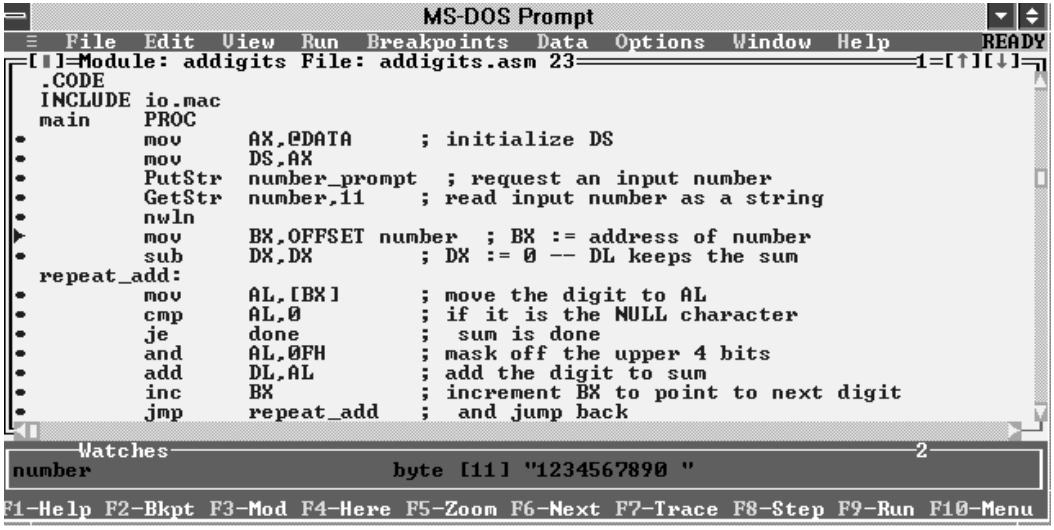


Figure D.3 TD window after reading the value 1234567890 into `number`.

`number`. As shown in Figure D.3, the watches window shows that `number` has properly received the input value. Breakpoints in a program can also be set by the `Breakpoints` menu.

The module window is useful in debugging programs at the source-code level. This is particularly helpful in debugging programs written in high-level languages such as C. Although

Table D.3 Selected `Run` menu options

Run (F9)	Execute program until completion or until a breakpoint is encountered
Goto cursor (F4)	Execute program up to the line that the cursor is on
Trace into (F7)	Execute one instruction at a time in single-step mode (similar to DEBUG Trace command)
Step over (F8)	Execute one statement at a time (a procedure call, interrupt, loop are treated as a single statement as in the Proceed command of DEBUG)

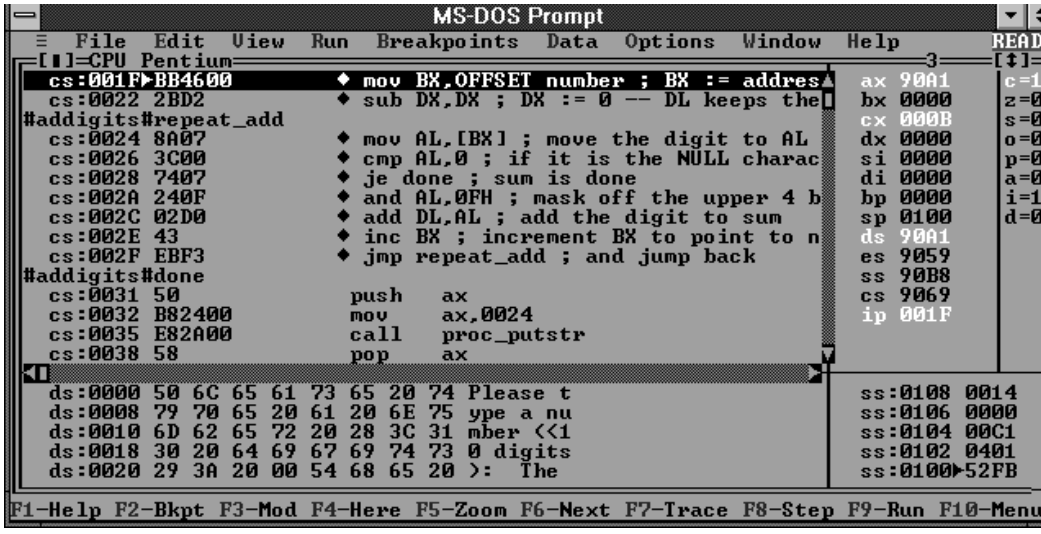


Figure D.4 CPU window before the `repeat_add` loop.

the source-code level of debugging is also useful in debugging assembly language programs (e.g., we can set convenient watches to monitor the progress), the CPU window is much more useful for low-level debugging. The remainder of the section focuses on the CPU window.

The CPU window provides a snapshot view of the program state. The CPU window after executing the program until

```
mov     BX,OFFSET number
```

is shown in Figure D.4. The window is divided into five panes. The code pane (top left pane) shows the CS:IP, along with the machine code and source-code lines. The current instruction is indicated by the arrow and also by highlighting the line if the code pane is active.

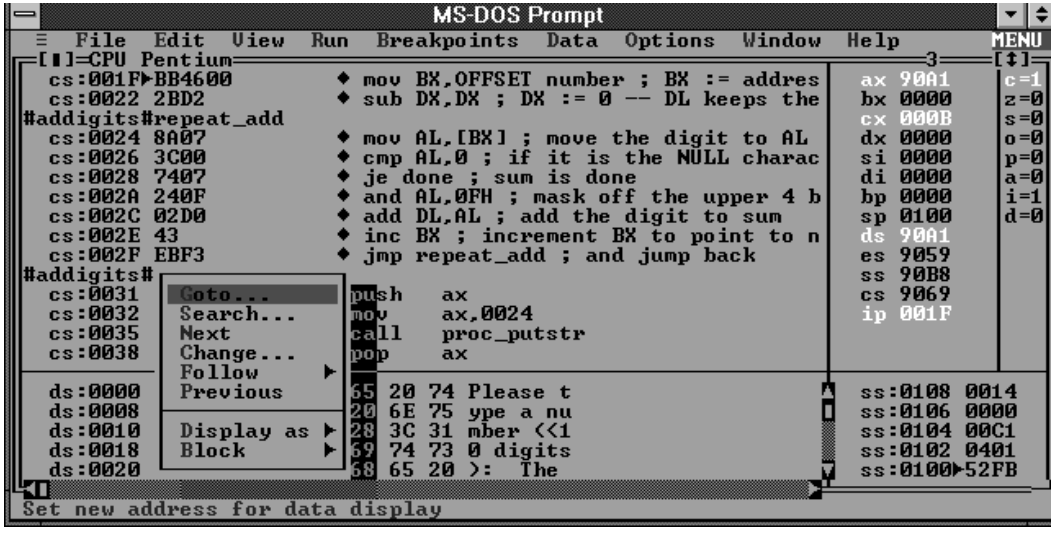


Figure D.5 CPU window with `Data Pane` local menu.

The next pane is the register pane and shows the contents of all 16-bit registers except flags. (We can display 32-bit registers by using an option in the local menu.) The status of flags is shown in the top right pane (flags pane). Unlike the DEBUG, the flag values are shown as 1 or 0 to indicate whether the flag is set or cleared, respectively. Also, changes in register values and flags are highlighted. For example, see registers AX, CX, and the carry flag.

The bottom left pane (data pane) shows the contents of the data segment. As shown in Figure D.4, the data pane shows the contents both in hex and ASCII. The fifth pane (stack pane) shows SS:SP and the contents of the stack. The top of the stack is indicated by an arrow. Remember that the stack grows toward low memory addresses. Therefore, SP values are displayed in decreasing order from top to bottom. We can use the tab key to move the cursor from one pane to the next.

An interesting feature of TD is its context-sensitive local menus. Depending on where the cursor is, a local pop-up menu can be activated by Alt-F10 or Ctrl-F10. Figure D.5 shows the pop-up local menu of the data pane. For example, we can use the option

```
Goto...
```

to specify an address to change the area of a data segment memory to be displayed. If we want to see the contents of `number` (whose offset is 46H), we can use this option of the local menu. The resulting data pane is shown in Figure D.6. It shows the input number that we have given to the program. We also see another similar sequence starting at DS:0064. This is actually the buffer into which `GetStr` reads the input number first before copying it into `number`.

Now if we want to check the logic of the `repeat_add` loop, we can use Trace Into (F7) or Step Over (F8) to single-step while monitoring the contents of the registers and flags. Since

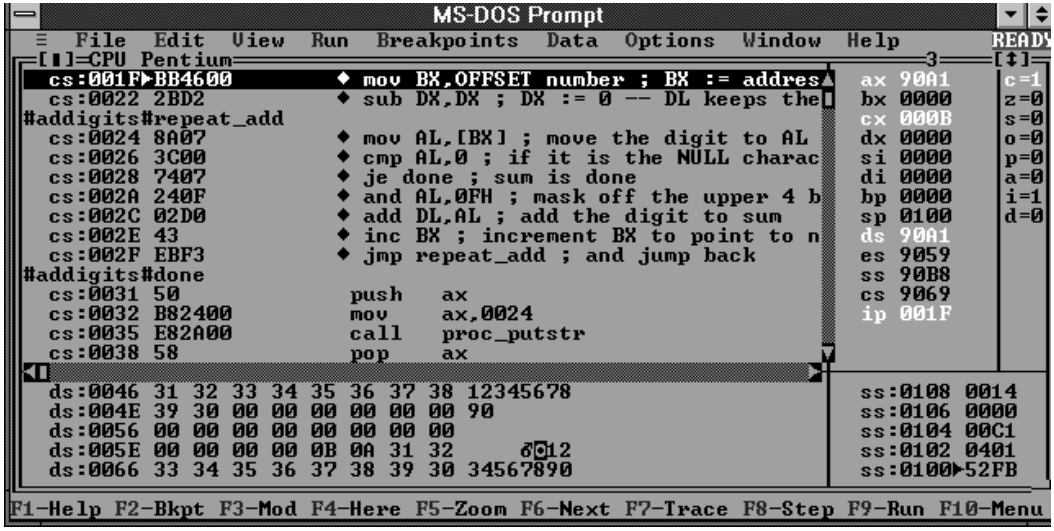


Figure D.6 CPU window after executing `Goto...` command.

there are no procedure calls or loop instructions, both F7 and F8 behave the same way for our example program. To see the complete execution of the `repeat_add` loop, move the cursor to the

```
push    AX
```

instruction at CS:0031 and press F4. The resulting state is shown in Figure D.7. Now notice that the sum in the DX register is 2DH, which is the hexadecimal equivalent of 45D.

In this brief discussion, we have glossed over numerous features available in TD. Now it is up to you to fully utilize the help offered by TD in debugging your assembly language programs.

D.4 CodeView

Microsoft's CodeView is similar in spirit to the Turbo Debugger. As in TD, the program should be assembled using the ZI option and linked with the CO option. This causes the symbolic information to be placed in the execution file.

Depending on the version of CodeView, some of the details might vary. Here we briefly discuss some generic features.

As in TD, we can add a variable or an expression to the watch list. The values of variables in the watch list are displayed in the watch window. The watch menu can be used to either add or delete an expression or a variable to or from the watch list. Also, breakpoints can be set or edited (i.e., added, deleted, etc.). Go (F5) can be used to execute from the next instruction to the completion of the program or until a breakpoint is encountered.

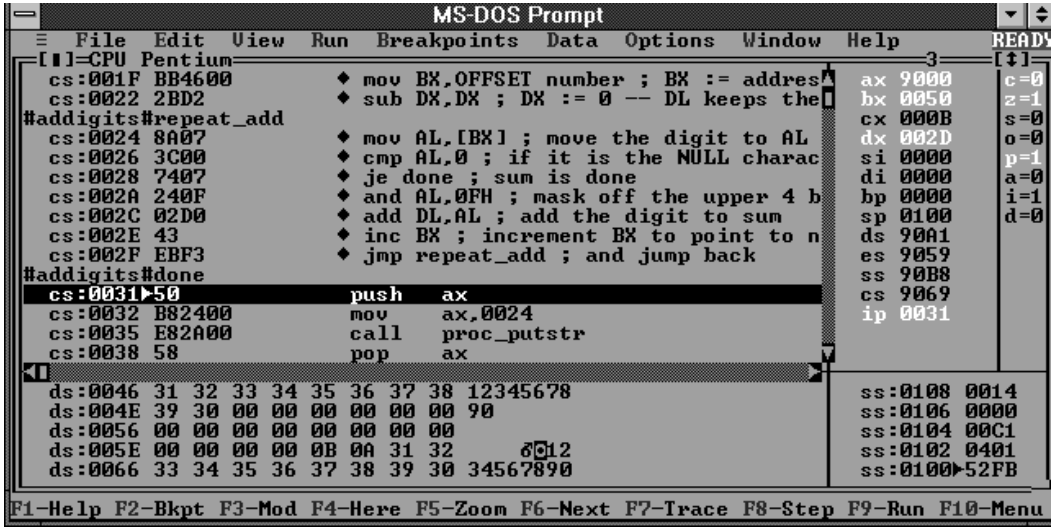


Figure D.7 CPU window after completing the `repeat_add` loop.

Trace (F8) and step (F10) commands are also available to control program execution. These are similar to trace into and step over commands available in Turbo Debugger.

The register window displays the contents of all registers including the flags register. The flag values are reported using the two-letter encoding used in DEBUG (see page 933).

The `view` menu provides several options to open other windows. For example, the `memory` option under this menu can be used to open the memory window and the `output` option switches to the program output window.

D.5 Summary

We started this appendix with a brief discussion of the basic debugging techniques. Since assembly language is a low-level programming language, debugging tends to be even more tedious than debugging a program written in a high-level language. It is, therefore, imperative to follow good programming practices in order to help debug and maintain assembly language programs.

There are several tools available for debugging programs. We discussed three debuggers—DEBUG, Turbo Debugger, and CodeView—in this appendix. DEBUG is a line-oriented debugger, and the other two are window-oriented and offer a much better user interface. The best way to learn to use these debuggers is by hands-on experience.

D.6 Exercises

D–1 Discuss some general techniques useful in debugging programs.

D–2 How are window-oriented debuggers such as Turbo Debugger better than line-oriented debuggers such as DEBUG?

D–3 What is the difference between the T and P commands of DEBUG?

D–4 Discuss how breakpoints are useful in debugging programs.

D–5 We have stated that the CPU window of the Turbo Debugger is more useful in debugging assembly language programs. Explain the reasons for this.

D.7 Programming Exercises

D–P1 Take a program from Chapter 9, and ask a friend to deliberately introduce some logical errors into the program. Then use your debugger to locate and fix errors. Discuss the features of your debugger that you found most useful.

D–P2 Using your debugger's capability to modify flags, verify the conditions mentioned for conditional jumps in Section 12.3.2 on page 500.

Appendix E

Running Pentium Assembly Language Programs on a Linux System

Objectives

- To present the structure of the assembly language programs that can run on a Linux system;
- To describe the NASM assembler;
- To give example NASM assembly language programs.

The main text presented Pentium assembly language programs that run under DOS. In this appendix, we give details on running these programs under Linux on an Intel PC. We use the NASM assembler to assemble the programs. We start this appendix with details on the NASM assembler. We have developed a set of input and output macros that are very similar to the macros used in the main text. For the most part, the programs for the DOS work with only minor modifications dictated by the NASM syntax. As a result, we only discuss the differences in this appendix.

E.1 Introduction

NASM, which stands for netwide assembler, is a portable, free public domain, IA-32 assembler that can generate a variety of object file formats. In this appendix, we restrict our discussion to a Linux system running on an Intel PC.

NASM can be downloaded from several sources (see the book's Web page for details). The NASM manual [27] has clear instructions on how to install NASM under Linux. Here is a summary extracted from the NASM manual:

1. Download the Linux source archive `nasm-X.XX.tar.gz`, where `X.XX` is the NASM version number in the archive.
2. Unpack the archive into a directory, which creates a subdirectory `nasm-X.XX`.
3. `cd` to `nasm-X.XX` and type `./configure`. This shell script will find the best C compiler to use and set up Makefiles accordingly.
4. Type `make` to build the `nasm` and `ndisasm` binaries.
5. Type `make install` to install `nasm` and `ndisasm` in `/usr/local/bin` and to install man pages.

This should install NASM on your system.

NASM can support several object file formats including the ELF (execute and link format) format used by Linux. The assembling and linking process is similar to that we discussed in Appendix C. For example, to assemble `addigits.asm`, we use

```
nasm -f elf addigits.asm
```

This generates the object file `addigits.o`. To generate the executable file `addigits`, we have to link this file with our I/O routines. This is done by

```
ld -s -o addigits addigits.o io.o
```

Note that `nasm` requires the `io.mac` file and `ld` needs the `io.o` file. Make sure that you have the two I/O files in your current directory. We give details about the I/O macros and routines in the next section.

E.2 NASM Assembly Language Program Template

We have written a set of macros and I/O routines to simplify input and output. These routines behave as do the ones we described on page 910 for the TASM and MASM assemblers. There are two I/O files:

- `io.mac` file contains the macro definitions for the I/O functions. This file is included in the assembly program (see Figure E.1);
- `io.o` contains the I/O routines that actually perform the operation. As described before, the linker needs this file.

```
;TITLE     brief title of program      file-name
;COMMENT
;            Objectives:
;                Inputs:
;               Outputs:
;
%include  "io.mac"

section  .data
 (initialized data go here)

section  .bss
 (uninitialized data go here)

section .text
        .STARTUP            ; setup
          . . .
          . . .
    (code goes here)
          . . .
          . . .
        .EXIT               ; returns control
```

Figure E.1 Template for the NASM assembly language programs.

We use the template shown in Figure E.1 for writing NASM assembly language programs. This is very similar to the template used for the DOS programs (see page 908). We include the `io.mac` file by using the `%include` directive. It is important to note that NASM is case-sensitive.

The data part is split into two: the `section .data` directive is used for initialized data and the `section .bss` directive for uninitialized data. The code part is identified by the `section .text` directive. The `.STARTUP` macro handles the code for setup, and the `.EXIT` macro returns control.

Notice that the data part is different from the TASM/MASM version. We can only use the define directives (DB, DW, . . .) for initialized data. For uninitialized data, we use RESB (reserve a byte) to reserve a byte. We can use the following directives:

`RESB`	Reserves a byte
`RESW`	Reserves a word
`RESD`	Reserves a doubleword
`RESQ`	Reserves a quadword
`REST`	Reserves 10 bytes

in the `.bss` section.

NASM uses different syntax to specify addresses. It does not support the `OFFSET` directive. The variable name is treated as representing the address. Thus, the TASM/MASM statement

```
mov     EBX,OFFSET number
```

is written in NASM as

```
mov     EBX,number
```

The TASM/MASM statement

```
mov     EBX,number
```

is written in NASM as

```
mov     EBX,[number]
```

Unlike the TASM/MASM versions we discussed in the main text, NASM supports 32-bit addressing. Other differences between TASM/MASM and NASM assemblers are given in [27]. Next we give some example programs that follow the NASM syntax.

E.3 Illustrative Examples

To show that migrating TASM/MASM programs require only minor changes, we present three examples from Part V: `addigits.asm`, `varapara.asm`, and `procfib2.asm`.

Example E.1 *Sum of the individual digits of a number.*
Program E.1 shows the NASM version of the `addigits.asm` program given on page 378. The structure of the program follows the template given in Figure E.1. The data part is split into initialized and uninitialized sections. The uninitialized section (`.bss`) uses the `RESB` directive to reserve 11 bytes for `number`. The `PutStr` and `GetStr` macros retain their semantics so there is no change on lines 20 and 21. Since NASM supports 32-bit addresses, we have to change `BX` to `EBX` on line 23. Also, we don't need the `OFFSET` directive to copy the address of `number`. To be consistent, we have also changed `BX` to `EBX` on line 31.

Program E.1 Sum of the individual digits of a number

```
1:  ;   Add individual digits of a number   ADDIGITS.ASM
2:  ;
3:  ;         Objective: To find the sum of individual digits of
4:  ;                    a given number. Shows character to binary
5:  ;                    conversion of digits.
6:  ;             Input: Requests a number from keyboard.
7:  ;            Output: Prints the sum of the individual digits.
8:
9:  %include "io.mac"
```

```
10:
11:  section  .data
12:  number_prompt  DB  'Please type a number (<11 digits): ',0
13:  out_msg        DB  'The sum of individual digits is: ',0
14:
15:  section  .bss
16:  number         RESB  11
17:
18:  section .text
19:          .STARTUP
20:          PutStr  number_prompt  ; request an input number
21:          GetStr  number,11      ; read input number as a string
22:          nwln
23:          mov     EBX,number   ; EBX := address of number
24:          sub     DX,DX        ; DX := 0 -- DL keeps the sum
25:  repeat_add:
26:          mov     AL,[EBX]     ; move the digit to AL
27:          cmp     AL,0         ; if it is the NULL character
28:          je      done         ;  sum is done
29:          and     AL,0FH       ; mask off the upper 4 bits
30:          add     DL,AL        ; add the digit to sum
31:          inc     EBX          ; increment BX to point to next digit
32:          jmp     repeat_add   ;  and jump back
33:  done:
34:          PutStr  out_msg
35:          PutInt  DX           ; write sum
36:          nwln
37:          .EXIT
```

Example E.2 *Passing a variable number of parameters via the stack.*

Program E.2 shows the NASM version of `varpara.asm` on page 419. Since the NASM `EQU` semantics are different, we have to use `%define` for `CRLF` on line 9. We have modified the program slightly to use EBX as an index into the stack to read the arguments. A significant change in this program is the offset value used to read the first number. In Program 10.6, we added 6 to BX to point to the first number (see line 61). Since NASM uses 32-bit addresses, the call on line 33 pushes EIP and the `enter` on line 50 pushes EBP. Thus, the NASM version pushes four more bytes onto the stack. Therefore, we have to add 10 (see line 56). The other minor changes follow our decision to use EBX as the index into the stack.

Program E.2 Passing a variable number of parameters via the stack

```
 1:  ; Variable number of parameters passed via stack   VARPARA.ASM
 2:  ;
 3:  ;        Objective: To show how variable number of parameters
 4:  ;                   can be passed via the stack.
 5:  ;            Input: Requests variable number of nonzero integers.
 6:  ;                   A zero terminates the input.
 7:  ;           Output: Outputs the sum of input numbers.
 8:
 9:  %define  CRLF   13,10    ; carriage return and line feed
10:
11:  %include  "io.mac"
12:
13:  section .data
14:  prompt_msg  DB  'Please input a set of nonzero integers.',CRLF
15:              DB  'You must enter at least one integer.',CRLF
16:              DB  'Enter zero to terminate the input.',0
17:  sum_msg     DB  'The sum of the input numbers is: ',0
18:
19:  section  .text
20:        .STARTUP
21:        PutStr  prompt_msg     ; request input numbers
22:        nwln
23:        sub     ECX,ECX        ; CX keeps number count
24:  read_number:
25:        GetInt  AX             ; read input number
26:        cmp     AX,0           ; if the number is zero
27:        je      stop_reading   ; no more numbers to read
28:        push    AX             ; place the number on stack
29:        inc     CX             ; increment number count
30:        jmp     read_number
31:  stop_reading:
32:        push    CX             ; place number count on stack
33:        call    variable_sum   ; returns sum in AX
34:        ; clear parameter space on the stack
35:        inc     CX             ; increment CX to include count
36:        add     CX,CX          ; CX := CX * 2 (space in bytes)
37:        add     SP,CX          ; update SP to clear parameter
38:                               ; space on the stack
39:        PutStr  sum_msg        ; display the sum
40:        PutInt  AX
41:        nwln
42:  done:
43:        .EXIT
```

```
44:  ;------------------------------------------------------------
45:  ;This procedure receives a variable number of integers via the
46:  ; stack. The last parameter pushed on the stack should be
47:  ; the number of integers to be added. Sum is returned in AX.
48:  ;------------------------------------------------------------
49:  variable_sum:
50:        enter 0,0
51:        push    EBX             ; save EBX and ECX
52:        push    ECX
53:
54:        xor     ECX,ECX
55:        mov     CX,[EBP+8]      ; CX := # of integers to be added
56:        mov     EBX,10          ; EBX := pointer to first number
57:        xor     AX,AX           ; sum := 0
58:  add_loop:
59:        add     AX,[EBP+EBX]    ; sum := sum + next number
60:        add     EBX,2           ; EBX points to the next integer
61:        loop    add_loop        ; repeat count in CX
62:
63:        pop     ECX             ; restore registers
64:        pop     EBX
65:        leave
66:        ret                     ; parameter space cleared by main
```

Example E.3 *Fibonacci number computation using the stack for local variables.*
In our last example, we gave the NASM version of `procfib2.asm` on page 425 (see Program E.3). There are very few differences between these two programs. The main one is the use of `%define` instead of `EQU` on lines 35 and 36.

Program E.3 Fibonacci number computation using the stack for local variables

```
 1:  ; Fibonacci numbers (stack version)    PROCFIB2.ASM
 2:  ;
 3:  ;       Objective: To compute Fibonacci number using the stack
 4:  ;                  for local variables.
 5:  ;           Input: Requests a positive integer from the user.
 6:  ;          Output: Outputs the largest Fibonacci number that
 7:  ;                  is less than or equal to the input number.
 8:
 9:  %include "io.mac"
10:
11:  section .data
12:  prompt_msg   DB  'Please input a positive number (>1): ',0
13:  output_msg1  DB  'The largest Fibonacci number less than '
```

```
14:                DB  'or equal to ',0
15:  output_msg2   DB  ' is ',0
16:
17:  section .text
18:        .STARTUP
19:        PutStr  prompt_msg      ; request input number
20:        GetInt  DX              ; DX := input number
21:        call    fibonacci
22:        PutStr  output_msg1     ; print Fibonacci number
23:        PutInt  DX
24:        PutStr  output_msg2
25:        PutInt  AX
26:        nwln
27:  done:
28:        .EXIT
29:
30:  ;-----------------------------------------------------------
31:  ;Procedure fibonacci receives an integer in DX and computes
32:  ; the largest Fibonacci number that is less than the input
33:  ; number. The Fibonacci number is returned in AX.
34:  ;-----------------------------------------------------------
35:  %define FIB_LO word [EBP-2]
36:  %define FIB_HI word [EBP-4]
37:  fibonacci:
38:        enter 4,0
39:        push    BX
40:        ; FIB_LO maintains the smaller of the last two Fibonacci
41:        ;  numbers computed; FIB_HI maintains the larger one.
42:        mov     FIB_LO,1        ; initialize FIB_LO and FIB_HI to
43:        mov     FIB_HI,1        ;  first two Fibonacci numbers
44:  fib_loop:
45:        mov     AX,FIB_HI       ; compute next Fibonacci number
46:        mov     BX,FIB_LO
47:        add     BX,AX
48:        mov     FIB_LO,AX
49:        mov     FIB_HI,BX
50:        cmp     BX,DX           ; compare with input number in DX
51:        jle     fib_loop        ; if not greater, find next number
52:        ; AX contains the required Fibonacci number
53:        pop     BX
54:        leave                   ; clear local variable space
55:        ret
```

E.4 Summary

We presented details about the NASM assembler, which is used to assemble programs to run under Linux on an Intel PC. There are several minor syntactical differences between the TASM/MASM and NASM assemblers. We have made the migration simple by redefining the I/O macros for the Linux system. As you can see from the examples presented in the last section, we need to make only minor changes to run the programs given in Part V.

E.5 Exercises

E–1 What features of NASM do you prefer over TASM/MASM?

E–2 We have not discussed how macros are defined in NASM. Using the information in the NASM manual [27], discuss the differences between how macros are defined in TASM/MASM and NASM assemblers. To get an idea, you can look at `io.mac` files for the two systems.

E–3 How is storage space reserved in NASM for uninitialized data?

E.6 Programming Exercises

E–P1 We have stated that NASM does not support the `OFFSET` directive. By defining

```
%idefine  OFFSET
```

we can instruct the preprocessor to treat `OFFSET` as a no-op. This is useful in migrating TASM/MASM assembly code. Verify this on a program from Part V.

E–P2 In the NASM version presented in Example E.2, we changed the logic by using EBX as an index into the stack. In this exercise, modify the TASM/MASM version presented on page 419 without changing the logic.

Appendix F

Digital Logic Simulators

Objectives

- To present some basic strategies to troubleshoot digital logic circuits;
- To provide an overview of four digital logic simulators.

Troubleshooting complex digital logic circuits is not a simple task. In Section F.1, we briefly present some of the techniques and tools available to implement and troubleshoot digital circuits. The next section gives details on some digital logic simulators. Our goal is to provide an overview of the simulators so that the reader can experiment with them. We end the appendix with some pointers on how you can obtain the simulators.

F.1 Testing Digital Logic Circuits

Testing a digital logic circuit implies that we have to "somehow" build the circuit. A digital circuit can be implemented in several ways. If the circuit is finalized, we can get a custom-made IC for the circuit provided the volume justifies the overhead. Alternatively, we can design a PCB (printed circuit board) and use off-the-shelf components by soldering these components onto the PCB. For example, the motherboard in your PC is a PCB that has several chips soldered onto it. However, when we are in the testing phase, we want something that is "not permanent" so that we can make modifications to the circuit, if necessary. If you are working in a university laboratory, it is also important to reuse some of the expensive components. For these reasons, prototyping of digital circuits is done using solderless breadboarding.

A breadboard consists of rows of small sockets into which components and wires can be inserted. Before the advent of logic simulators, breadboarding used to be the only way to test your design in a lab. Even now, you may want to get your hands dirty to get a real feeling for the components and to test the electrical characteristics. Since our goal here is to test functionality of a digital circuit, we prefer to use logical simulators to verify our design.

Just as with debugging a program, simple mistakes can cause serious problems with digital logic design implementations. For example, a short circuit or an open circuit can cause serious functionality problems. In general, testing combinational circuits is much easier than testing sequential circuits. In combinational circuits, there is no feedback to complicate things (e.g., as in the flip-flops we have seen in Chapter 4). In addition, there is no clock to involve timing. Combinational circuit testing can be done by using a simple device called a *logic probe*. This is a penlike tool (like your electric tester) that indicates whether the value is 0 or 1 when you touch its metal tip to the circuit (e.g., output of a gate). Some of the logic simulators we discuss in the next section facilitate such testing.

Sequential circuit testing is done by using logic analyzers. Logic analyzers relate the state of various circuit points by displaying their logical level synchronized by time. Most simulators provide logic analyzers. We show some example logic analyzer windows in the next section. However, the best way to understand them is to use one of the simulators and explore its utility.

F.2 Digital Logic Simulators

We discuss four logic simulators: DIGSim, Digital Simulator, Multimedia Logic Simulator, and Logikad. The first three simulators are either freely available, at least to students and educational institutions, or cost between $10 and $20. At the end of this appendix, we give information on where you can get them.

F.2.1 DIGSim Simulator

This is a Java-based simulator and is available in public domain. You don't need to download and install it; the Applet runs in your Web browser. If you have problems running it under Explorer, switch to Netscape. The only caveat is that you cannot store your design if you use a Web browser due to a security problem. To save your designs, you need to run this in the Applet viewer from Sun. This may not be a simple task if you are not familiar with the Java environment.

A sample screen shot is shown in Figure F.1. This simulator is very easy to use and takes less time to specify your design (compared to the others we discuss next). The full-adder design given in Figure 3.18*b* (page 96) can be implemented and tested within a few minutes. Its use of point-and-click eliminates the need for reading a manual before using it.

DIGSim supports several useful devices including the following:

- Basic 2- and 3-input NAND, NOR, AND, OR gates and 2-input XOR and XNOR (equivalence) gates;
- Several types of latches including the three latches we have seen in Chapter 4: SR latch, clocked SR latch, and D latch;
- Simple flip-flops such as the D flip-flop, JK flip-flop, and T flip-flop;
- Special circuits such as shift registers, decoder, BCD-to-seven-segment decoder, and 4-bit binary counter;
- Display devices including different types of LEDs and a seven-segment display.

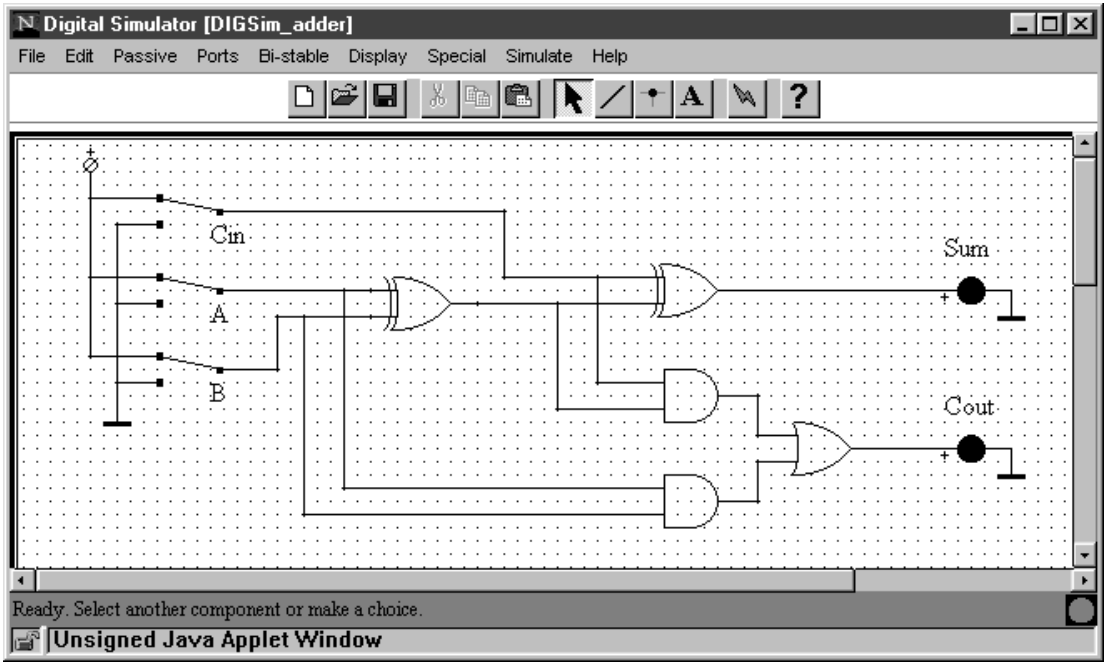


Figure F.1 DIGSim window showing the full-adder implementation.

Two features of DIGSim facilitate troubleshooting of digital circuits. A feature that is very useful is its color-coded wires to indicate the status of the signal:

- Red color is used to indicate the logic 0 value;
- Green color is used to indicate the logic 1 value.

Thus, we don't need a special logic probe to test the logical value of the signals. This is enough to troubleshoot combinational circuits.

DIGSim also provides a logic analyzer to troubleshoot sequential circuits. As an example, consider the 3-bit binary counter shown in Figure 4.14 on page 123. The timing diagram given in this figure is what we expect our logic analyzer to show when we add probe points to the clock input and three counter outputs. As you can see from Figure F.2, the counter behaves the way it should, and the logic analyzer captures the timing information.

F.2.2 Digital Simulator

Digital Simulator is a shareware program written by Ara Knaian. It provides the basic building blocks to implement and test digital designs. It is less flexible than the DIGSim simulator discussed in the last section. The basic devices supported by Digital Simulator are the ones you

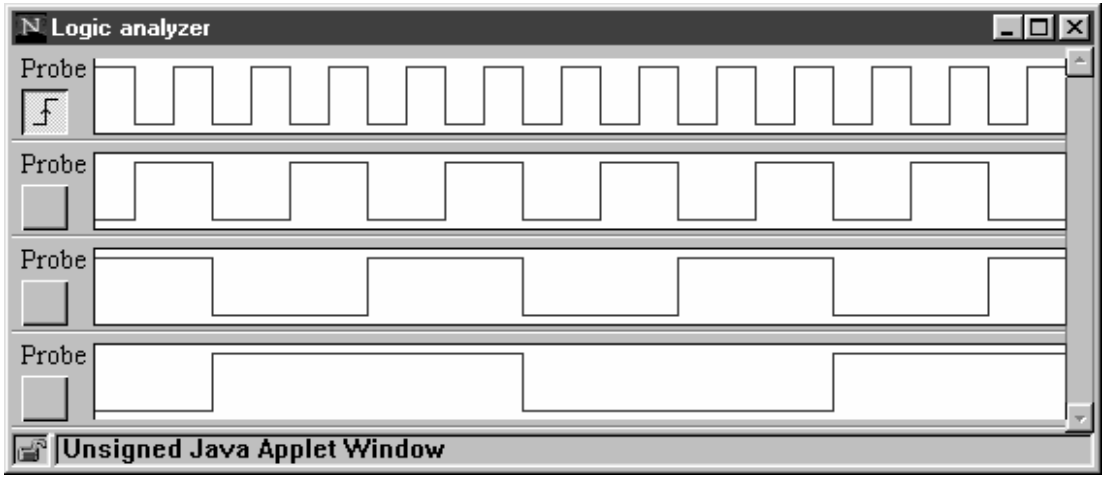


Figure F.2 DIGSim logic analyzer output for a 3-bit binary counter. Top waveform represents the clock and the bottom three are from the three outputs of the counter.

see in Figure F.3. It provides the basic AND, OR, XOR, and NOT gates. Unlike DIGSim, it does not support 3-input gates.

In addition, as shown in Figure F.3, Digital Simulator supports the following devices:

- An RS latch;
- D and JK flip-flops;
- A BCD-to-seven segment decoder;
- Display devices such as LEDs and seven-segment display.

Digital Simulator also provides a logic analyzer that is very similar to the DIGSim's logic analyzer. A snapshot of the 3-bit binary counter outputs is shown in Figure F.4.

Digital Simulator includes two memory devices:

- A 4×4 ROM;
- A 4×4 RAM.

The ROM has an output enable input. When you select this device, it will open a window for you to enter the contents of the ROM. Unfortunately, the address and data lines are labeled as A1 $\cdots$ A4 and D1 $\cdots$ D4, respectively. It would have been nice to see these labels start with A0 and D0.

The RAM has two control signals: a chip select ($\overline{\text{CS}}$) input and a read/write (R/$\overline{\text{W}}$) input. This memory block is very similar to the one discussed in Chapter 16 on page 676. The only difference is that we used two separate lines for the read and write control in Figure 16.9. It is

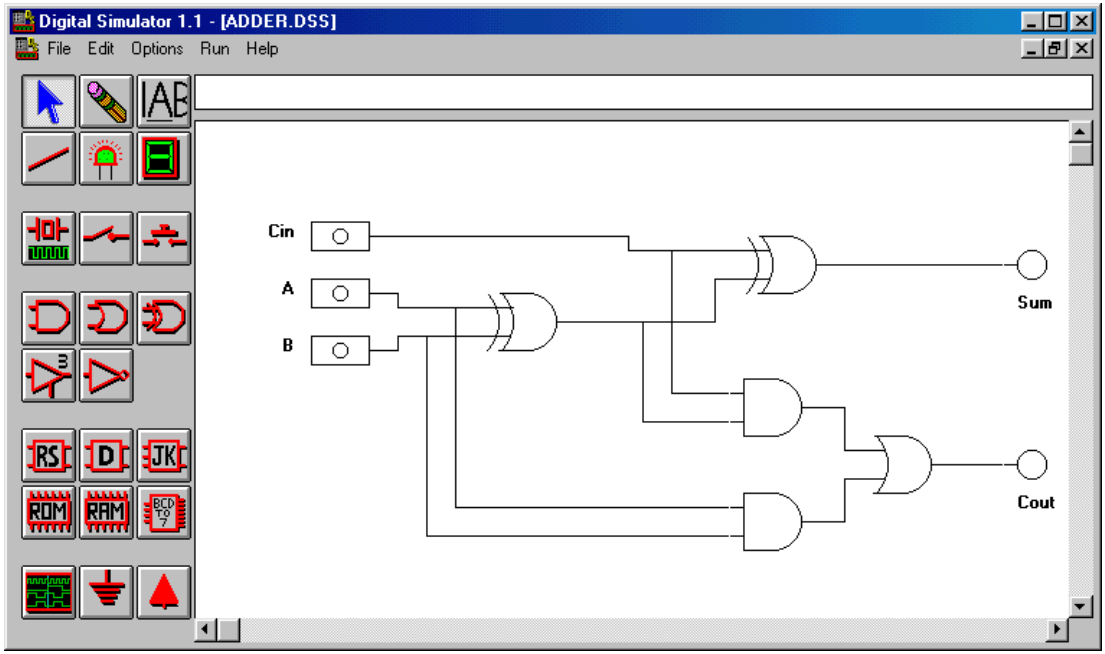


Figure F.3 Digital simulator window with the full-adder circuit.

straightforward to use these devices to verify our memory designs presented in Chapter 16. The tristate buffer comes in handy to test designs like the ones shown in Figure 16.8 on page 675.

One feature that makes it difficult to use is its inflexibility in device movement. Once a device is placed, you cannot move it. You will find it very annoying, particularly after using other simulators like DIGSim.

F.2.3 Multimedia Logic Simulator

The Multimedia Logic Simulator from Softronics is more sophisticated than the last two simulators. In terms of the basic gates, as with the Digital Simulator, only 2-input logic gates (AND, OR, XOR) are supported. Figure F.5 shows the simulator window along with the device palette.

The simulator window in this figure shows the full-adder circuit we have used in the other two simulators before. It supports the following devices:

- A basic RS latch;
- An 8-to-1 multiplexer;
- An 8-bit binary counter;
- A tristate inverter.

Some interesting features of this simulator are summarized below:

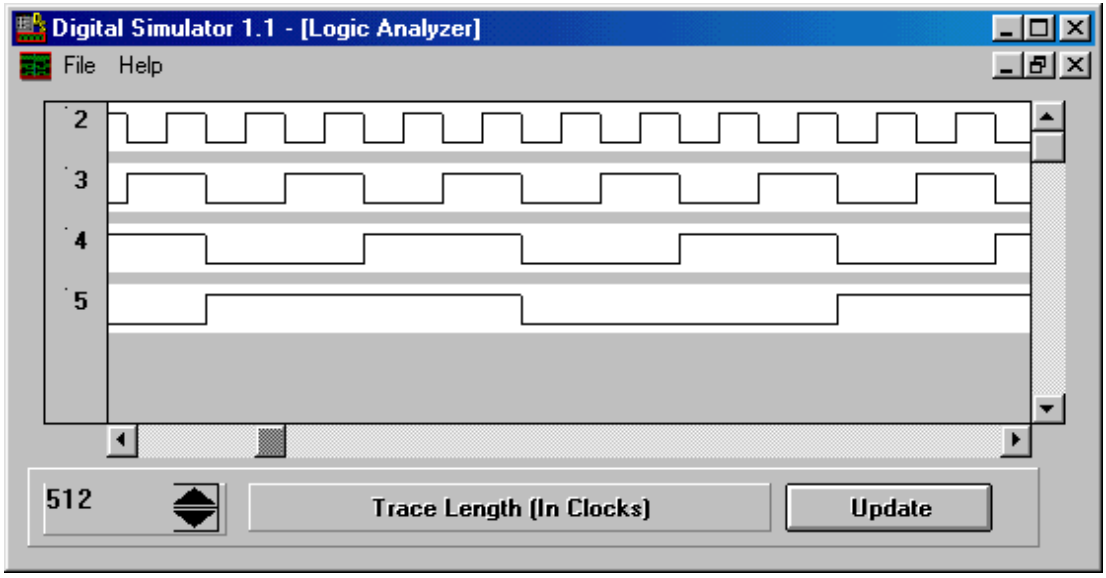


Figure F.4 Digital Simulator logic analyzer window for the 3-bit counter. This logic analyzer is very similar to the one provided by DIGSim.

- *Provides an 8-Function 8-Bit ALU:* The functions supported include addition, subtraction, multiplication, division, shifting, and comparison. The ALU has two output flags to indicate the status of an operation: a Z (zero) flag and an overflow (V) flag. The zero flag is similar to the Pentium zero flag. It is set to 1 if the result is zero; the Z flag is cleared otherwise. The overflow flag is similar to the Pentium's carry flag. It provides carry-in and carry-out to cascade several 8-bit ALUs to build larger ALUs.
- *Supports Input and Output Devices:* A simple 16-key Hex keyboard is useful to give number input to your circuits. The ASCII output can be displayed using the display device. It is a 8×16 display (8 lines with 16 characters in each line). The display supports carriage return and backspace functions.
- *Supports a* 256×8 *Memory Device:* Unfortunately, it does not provide a $\overline{\text{CS}}$ input. Instead, it has 8 data in lines, and 8 data out lines as in our memory design shown in Figure 16.1 on page 668. We will have to use tristate buffers to implement a memory block with $\overline{\text{CS}}$ input (as we did in Figure 16.8 on page 675).

This simulator provides a logic probe and a logic analyzer to facilitate troubleshooting.

F.2.4 Logikad Simulator

Logikad is available from Prentice-Hall. This simulator is different from the last three we have discussed. It uses chips rather than logic symbols to implement the design. In this sense, it

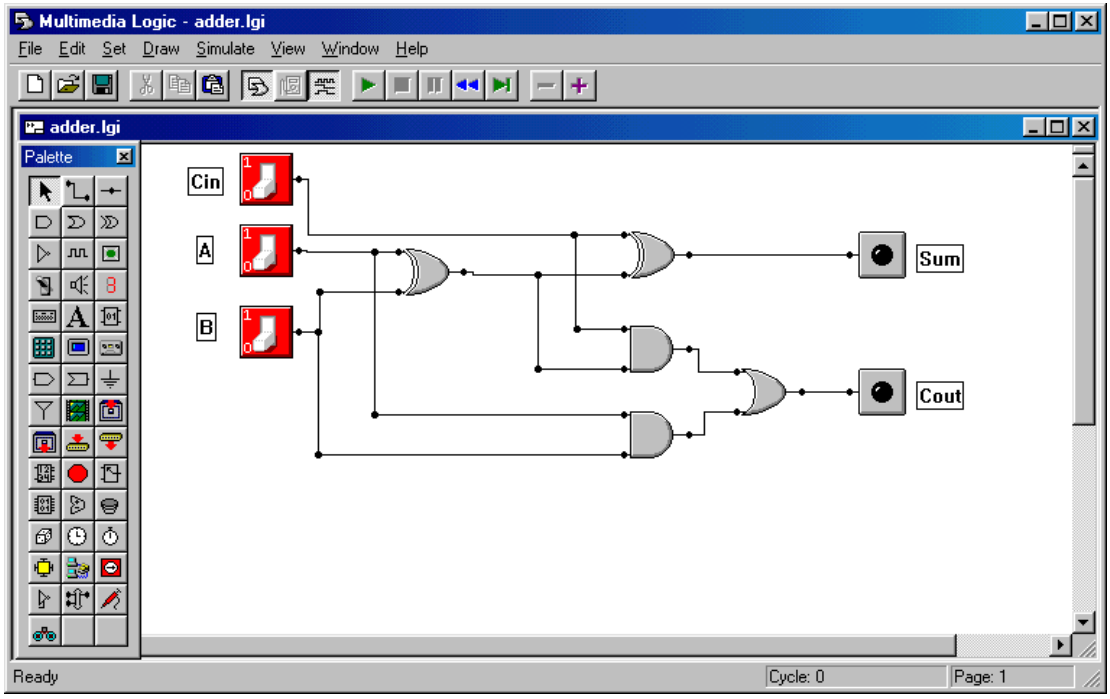


Figure F.5 Multimedia digital simulator window with the full-adder circuit.

emulates the breadboard experience. Figure F.6 shows a sample Logikad window.

The circuit in this figure implements the same full-adder we have used in the previous simulators. As you can see from this figure, it mimics a typical digital laboratory's breadboard. This circuit uses three chips:

- 7486: contains four 2-input XOR gates;
- 7408: contains four 2-input AND gates;
- 7432: contains four 2-input OR gates.

The small circle on each chip identifies pin 1. Figure 2.8 on page 50 shows details of these chips. To implement this circuit, we are using three chips: two of these are 50% used, and the third one is only 25% used. We leave it as an exercise to implement the same circuit using only two chips (see Exercise F–1).

Logikad provides several TTL chips. An example device selection window for NAND gates is shown in Figure F.7. In general, the 7400 series TTL chips are used for commercial systems; 5400 series chips are used for military applications as these chips can tolerate higher temperature ranges.

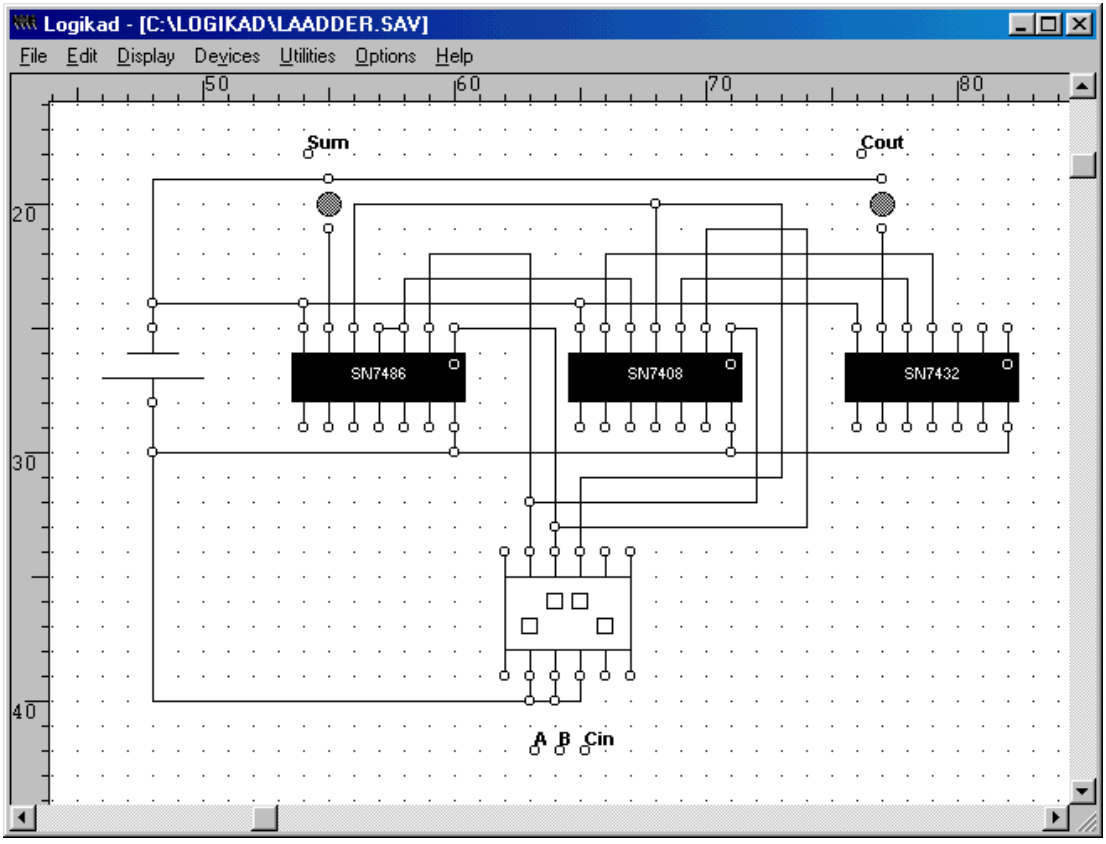


Figure F.6 Logikad digital simulator screen shot.

Logikad has a feature that we have not seen in the other three simulators. It can be used to simplify logical expressions with up to four variables. The Karnaugh map window for four variables is shown in Figure F.8. We enter the truth table by clicking the function output (**F** column in the figure). It accepts three inputs: 0, 1, X (don't care). Once the truth table is defined, clicking the "Calculate" button will produce the result. The result type can be selected to be either sum-of-products or product-of-sums. The truth table in this figure represents the seven-segment truth table shown on page 66. The final sum-of-products expression matches the one derived on page 65.

Logikad also derives NAND-only implementations. The NAND button in the Karnaugh map window generates a NAND-only implementation, as shown in Figure F.9. This NAND implementation corresponds to the seven-segment display example.

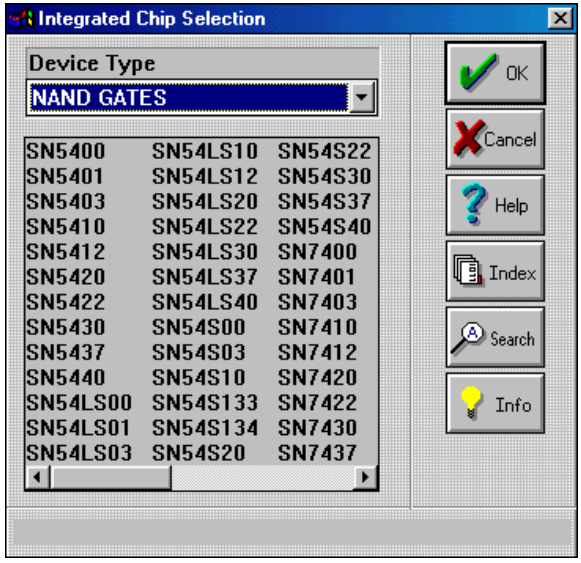


Figure F.7 Logikad device selection window for NAND gate devices.

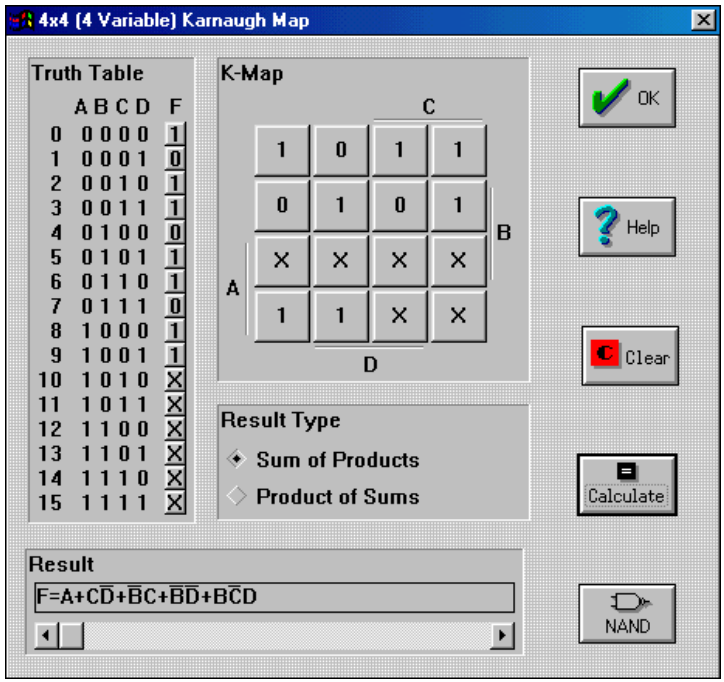


Figure F.8 Logikad supports Karnaugh map simplification for up to four logical variables.

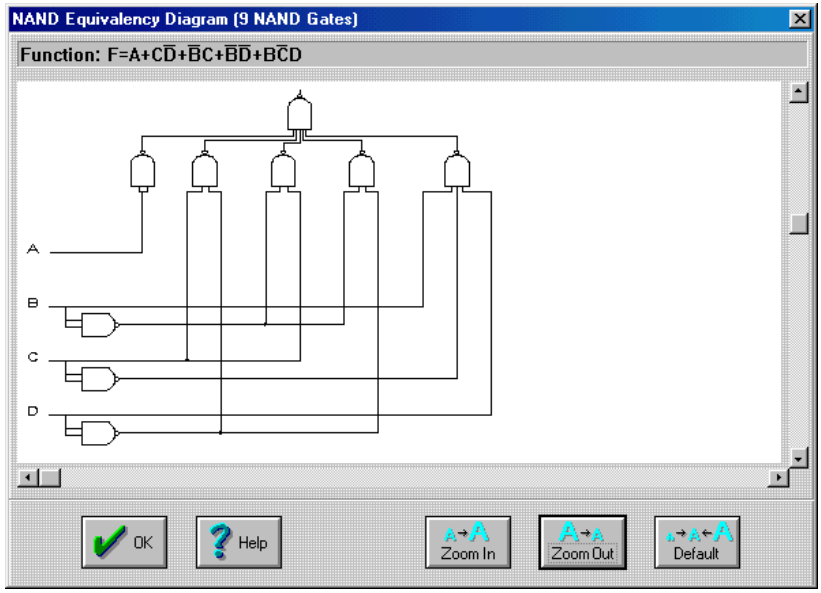


Figure F.9 Logikad's NAND-only implementation for the seven-segment display example.

F.3 Summary

We have reviewed four digital logic simulators to test the digital circuits discussed in this book. All four are adequate for testing combinational and sequential circuits. DIGSim is the simplest and easiest to use. There is no download and installation involved. You can test your designs online. It also uses a clever color-coding to give a visual indication of the logical level of each signal in the circuit. If you are new to this area, try this simulator first.

Digital Simulator's capabilities are somewhat similar to that of DIGSim. However, Digital Simulator provides ROM and RAM memory support. The Multimedia Logic simulator provides many more features than the other two simulators. Notable among these are a 16-key Hex keyboard, an 8×16 ASCII display, and a 256×8 memory.

The last simulator is marketed by Prentice-Hall. This simulator is included because it simulates breadboarding. It uses chips rather than gates. This simulator also provides substantial device support. The best way to learn to use these debuggers is by hands-on experience.

F.4 Web Resources

This section lists the URLs for the digital simulation software. It is possible that some of these might have changed. If you have difficulty, use a good search engine to find their current location.

Iwan van Rienen's DIGSim is available from several sources. Here is one URL that is likely to be stable: `sunsite.utk.edu/winners_circle/education/ED8N1T2I/applet.html`. If you want to save your circuit design, run DIGSim in the Applet viewer provided by Sun. You can get this viewer from `http://java.sun.com`.

You can obtain the Digital Simulator from `web.mit.edu/ara/www/ds.html`.

The Multimedia Logic Simulator is available from `www.softronics.com`. In April 2001, the beta version was available for free. Check their Web page for more details.

Logikad is available from Prentice-Hall. A student version, Logikad Lite, is also available. Check the Prentice-Hall Web page at

`www.prenticehall.ca/allbooks/ect_0132721880.html` (Full version);
`www.prenticehall.ca/allbooks/ect_0132628090.html` (Logikad Lite).

EasySim also provides the basic gates, counters, and so on. It is available for $14.95 from `www.starnet.com.au/research/easysim/easysim.htm`.

LogicWorks is available from Capilano Computing Systems Ltd. See their Web site for details: `www.capilano.com/logicworks/`. The software is available from Prentice-Hall (but it is expensive).

SuperSIM is available from `www.designnotes.com/SuperSIM.htm`. It comes in two versions. The cheaper version costs about $139.

For UNIX systems, the Chipmunk system provides software tools for electronic circuit simulation and schematic capture. Note that this package provides many other tools. For details, check the Chipmunk homepage at `www.pcmp.caltech.edu/chipmunk/`.

F.5 Exercises

F–1 Our implementation of the full-adder shown in Figure F.6 used three chips. Show an implementation of this circuit using only two chips: a 7400 NAND gate chip and a 7486 XOR chip.

F–2 Using your favorite digital simulator, verify that the circuit shown in Figure 3.27 on page 104 implements the four-function ALU.

F–3 Implement the master-slave JK flip-flop shown in Figure 4.10 on page 118. Using the logic analyzer, verify the timing diagram shown in Figure 4.10*b*.

F–4 Implement the 3-bit synchronous counter shown in Figure 4.19 on page 130. Using the logic analyzer, verify the functionality of this circuit.

F–5 Implement the 3-bit counter shown in Figure 4.22 on page 133. Using the logic analyzer, verify the functionality of this circuit.

Appendix G

SPIM Simulator and Debugger

Objectives

- To give details about downloading and using the SPIM simulator;
- To explain the basic SPIM interface;
- To describe the SPIM debugger commands.

SPIM is a simulator to run MIPS programs. SPIM supports various platforms and can be downloaded from the Web. SPIM also contains a simple debugger. In this appendix, we present details on how to download and use the SPIM simulator. We start with an introduction to the SPIM simulator. The following section gives details about SPIM settings. These settings determine how the simulator loads and runs your programs. We specify the setting you should use in order to run the example MIPS programs given in Chapter 15. Details about loading and running a MIPS program are discussed in the next section. This section also presents debugging facilities provided by SPIM. We conclude the appendix with a summary.

G.1 Introduction

This appendix describes the SPIM simulator, which was developed by Professor James Larus when he was at the Computer Science Department of the University of Wisconsin, Madison. This simulator executes the programs written for the MIPS R2000/R3000 processors. This is a two-in-one product: it contains a simulator to run the MIPS programs as well as a debugger.

SPIM runs on a variety of platforms including UNIX/Linux, Windows (95, 98, NT, 2000), and DOS. In this appendix, we provide details on the Windows 98 version of SPIM called PC-

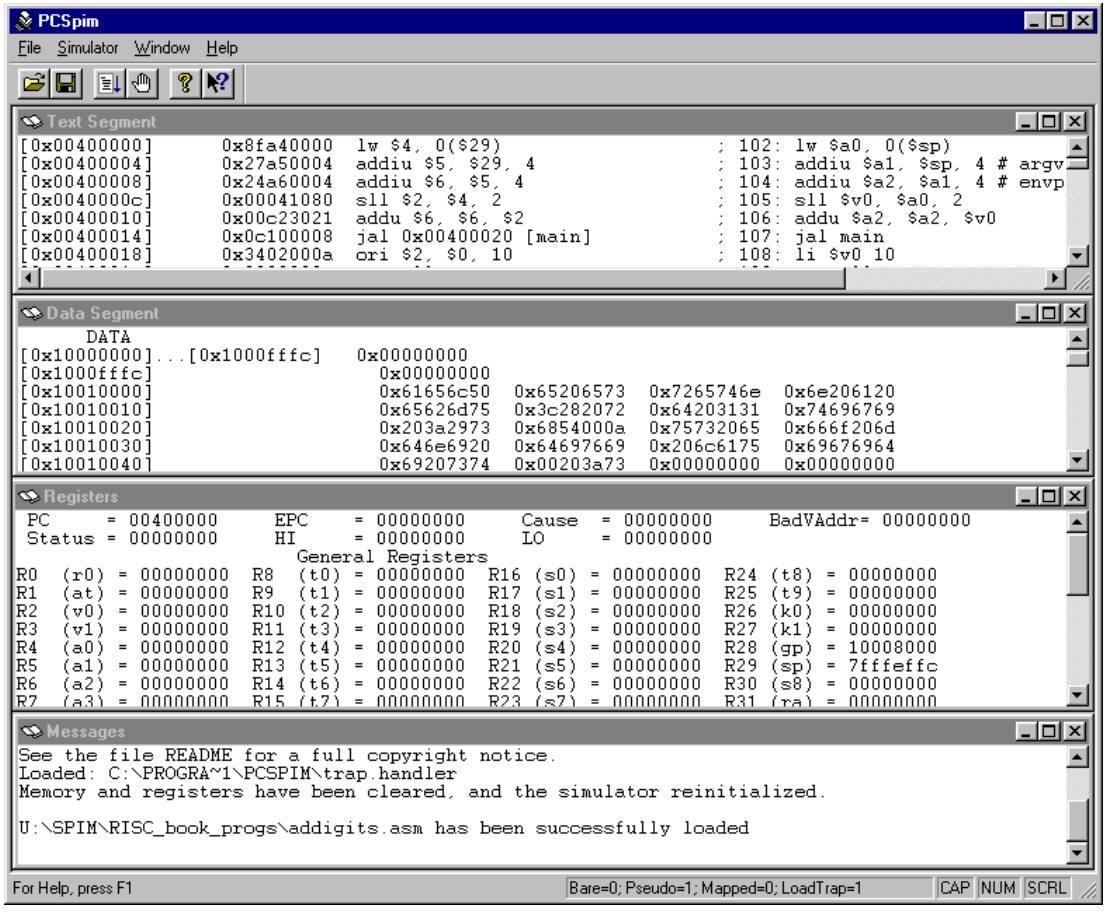


Figure G.1 SPIM windows.

Spim. The SPIM simulator can be downloaded from `http://www.cs.wisc.edu/~larus/spim.html`. This page also gives information on SPIM documentation. Although SPIM is available from this site at the time of this writing, use a good search engine to locate the URL, if it is not available from this URL. Also, you can check this book's homepage, which has a link to the SPIM simulator that is updated periodically.

Figure G.1 shows the PCSpim interface. As shown in this figure, PCSpim provides a menu bar and a toolbar at the top and a status bar at the bottom of the screen. The middle area displays four windows, as discussed next.

- **Menu Bar:** The menu bar provides the following commands for the simulator operation:
  - *File:* The `File` menu allows you select file operations. You can open an assembly language source file using `open...` or save a log file of the current simulator state.

In addition, you can quit PCSpim by selecting the `Exit` command. Of course, you can also quit PCSpim by closing the window.

- *Simulator:* This menu provides several commands to run and debug a program. We discuss these commands in Section G.3.2. This menu also allows you to select the simulator settings. When the `Settings...` command is selected, it opens a setting window to set the simulator settings, which are discussed in the next section.
- *Windows:* This menu allows you to control the presentation and navigation of windows. For example, in Figure G.1, we have tiled windows to show the four windows: Text Segment, Data Segment, Register, and Messages. In addition, you can also elect to hide or display the toolbar and status bar. The console window pops up when your program needs to read/write data to the terminal. It disappears after the program has terminated. When you want to see your program's input and output, you can activate this window by selecting the Console window command.
- *Help:* This menu allows you to obtain online help on PCSpim.

- **Toolbar:** The toolbar provides mouse buttons to open and close a MIPS assembly language source file, to run and insert breakpoints, and to get help.
- **Window Display Section:** This section displays four windows: Data Segment, Text Segment, Messages, and Register.
  - *Data Segment Window:* This window shows the data and stack contents of your program. Each line consists of an address (in square brackets) and the corresponding contents in hexadecimal notation. If a block of memory contains the same constant, an address range is specified as shown on the first line of the Data Segment in Figure G.1.
  - *Text Segment Window:* This window shows the instructions from your program as well as the system code loaded by PCSpim. The leftmost hex number in square brackets is the address of the instruction. The second hex number is the machine instruction encoding of the instruction. Next to it is the instruction mnemonic, which is a processor instruction. What you see after the semicolon is the source code line including any comments you have placed. This display is useful for seeing how the pseudoinstructions of the assembler are translated into the processor instructions. For example, the last line in the Text Segment of Figure G.1 shows that the pseudoinstruction

    ```
    li    $vi,10
    ```

    is translated as

    ```
    ori    $2,$0,10
    ```

  - *Registers:* This window shows the contents of the general and floating-point registers. The contents are displayed in either decimal or hex notation, depending on the settings used (discussed in the next section).
  - *Messages:* This window is used by PCSpim to display error messages.

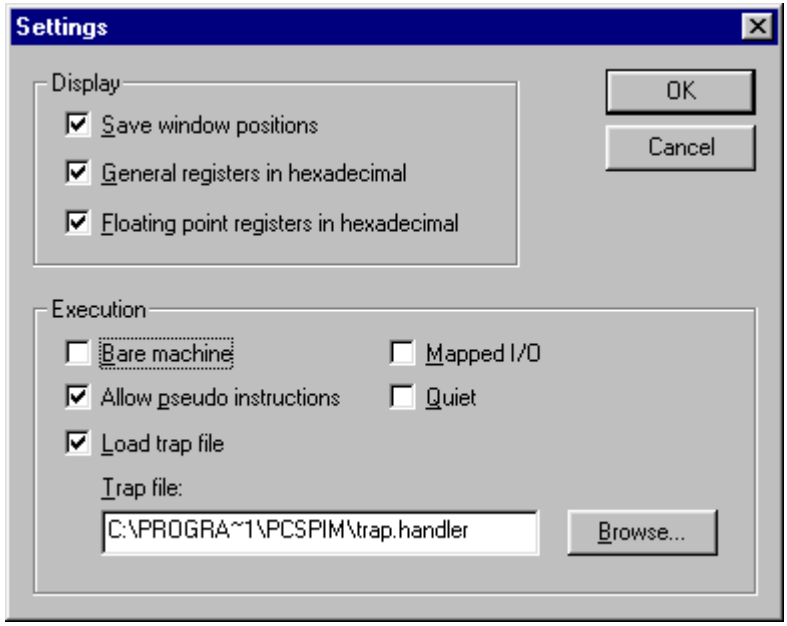


Figure G.2 SPIM settings window.

- **Status Bar:** The status bar at the bottom of the PCSpim window presents three pieces of information:
  - The left area is used to give information about the menu items and toolbar buttons. For example, when the mouse arrow is on the open file icon (first button) on the toolbar, this area displays the "Open an assembly file" message.
  - The middle area shows the current simulator settings. Simulator settings are described in the next section.
  - The right area is used to display if the Caps Lock key (CAP), Num Lock key (NUM), and Scroll Lock key (SCRL) are latched down.

G.2 Simulator Settings

PCSpim settings can be viewed by selecting the `Settings` command under the `Simulator` menu. This opens a setting window as shown in Figure G.2. PCSpim uses these settings to determine how to load and run your MIPS program. An incorrect setting may cause errors. The settings are divided into two groups: Display and Execution. The Display settings determine whether the window positions are saved and how the contents of the registers are displayed. When *Save window positions* is selected, PCSpim will remember the position of its windows when you exit and restore them when you run PCSpim later. If you select the register dis-

play option, contents of the general and floating-point registers are displayed in hexadecimal notation. Otherwise, register contents are displayed as decimal numbers.

The Execution part of the settings shown in Figure G.2 determines how your program is executed.

- **Bare Machine:** If selected, SPIM simulates a bare MIPS machine. This means that both pseudoinstructions and additional addressing modes, which are provided by the assembler, are not allowed. See Chapter 15 for details on the assembler-supported pseudoinstructions and addressing modes. Since the example MIPS programs presented in Chapter 15 use these additional features of the assembler, this option should not be selected to run our example programs.
- **Allow Pseudoinstructions:** This setting determines whether the pseudoinstructions are allowed in the source code. You should select this option as our example programs use pseudoinstructions.
- **Mapped I/O:** If this setting is selected, SPIM enables the memory-mapped I/O facility. Memory-mapped I/O is discussed in detail in Chapter 19. When this setting is selected, you cannot use SPIM system calls, described in Section 15.3 on page 632, to read from the terminal. Thus, this setting should not be selected to run our example programs from Chapter 15.
- **Quiet:** If this setting is selected, PCSpim will print a message when an exception occurs.
- **Load Trap File:** Selecting this setting causes PCSpim to load the standard exception handler and startup code. The trap handler can be selected by using the *Browse* button. When loaded, the startup code in the trap file invokes the `main` routine. In this case, we can label the first executable statement in our program as `main`. If the trap file is not selected, PCSpim starts execution from the statement labeled `__start`. Our example programs are written with the assumption that the trap file is loaded (we use the `main` label). If you decide not to use the trap file, you have to change the label to `__start` to run the programs. If the trap file is loaded, PCSpim transfers control to location 0x80000080 when an exception occurs. This location must contain an exception handler.

G.3 Running and Debugging a Program

G.3.1 Loading and Running

Before executing a program, you need to load the program you want to run. This can be done either by selecting the *Open File* button from the Toolbar or from the `File` menu. This command lets you browse for your assembly file by opening a dialog box. After opening the file, you can issue the `Run` command either from the Toolbar or from the `Simulator` menu to execute the program.

The Run command pops the Run window shown in Figure G.3. It automatically fills the start address. For our example programs, you don't have to change this value. If desired, the command line options can be entered in this window. Command line options that you can specify include the settings we have discussed in the last section. For example, you enter `-bare`

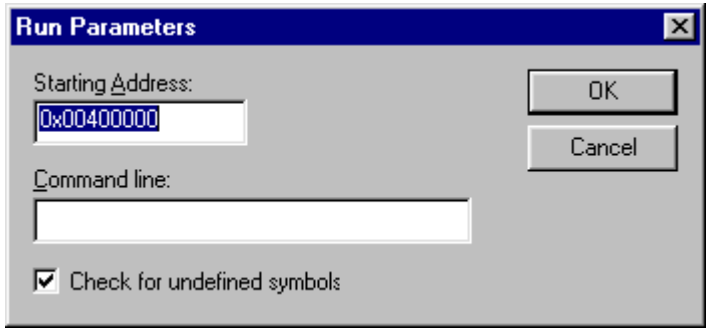


Figure G.3 Run window.

to simulate a bare MIPS machine, `-asm` to simulate the virtual MIPS machine provided by the assembler, and so on. The SPIM documentation contains a full list of acceptable command line options. If you have set up the settings as discussed in the last section, you don't have to enter any command line option to run the example programs from Chapter 15.

G.3.2 Debugging

SPIM provides the standard facilities to debug programs. As discussed in Appendix D, single-stepping and breakpoints are the two most popular techniques used to debug assembly language programs. Once you find a problem or as part of debugging, you often need to change the values in a register set or memory locations. As do the other debuggers discussed in Appendix D, SPIM also provides commands to alter the value of a register or memory location. All debug commands are available under the `Simulator` menu as shown in Figure G.4. These commands are briefly explained next.

- **Clear Registers:** This command clears all registers (i.e., the values of all registers are set to zero).
- **Reinitialize:** It clears all the registers and memory and restarts the simulator.
- **Reload:** This command reinitializes the simulator and reloads the current assembler file for execution.
- **Go:** You can issue this command to run the current program. Program execution continues until a breakpoint is encountered. We have discussed the Run command before. You can also use the F5 key to execute your program.
- **Break/Continue:** This can be used to toggle between break and continue. If the program is running, execution is paused. On the other hand, if the execution is paused, it continues execution.
- **Single Step:** This is the single-step command. The simulator executes one instruction and pauses execution. You can also use the F10 key for single-stepping.

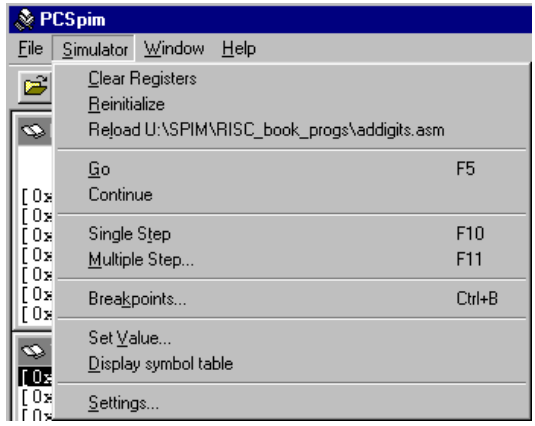


Figure G.4 Debug commands available under the `Simulator` menu.

- **Multiple Step:** This is a debug command we have not discussed in Appendix D. This is a generalization of single-stepping. In this command, you can specify the number of instructions each step should execute. When you select this command, SPIM opens a dialog window to get the number of instructions information.
- **Breakpoints...:** This command is useful to set up breakpoints. It opens the Breakpoint dialog box shown in Figure G.5. You can add/delete breakpoints through this dialog box. As shown in this figure, it also lists the active breakpoints. When the execution reaches a breakpoint, execution pauses and pops a query dialog box (Figure G.6) to continue execution. Normally, you enter the address of the instruction to specify a breakpoint. However, if the instruction has a global label, you can enter this label instead of its address.
- **Set Value...:** This command can be used to set the value of a register or a memory location. It pops a window to enter the register/memory address and the value as shown in Figure G.7. In this example, we are setting the value of the `$a2` register to 7FFFF000H.
- **Display Symbol Table:** This command displays the simulator symbol table in the message window.
- **Settings...:** This opens the `Settings` dialog box shown on page 972. We have discussed the simulator settings in detail in Section G.2.

When single-stepping your program, the instructions you see do not exactly correspond to your source code for two reasons: the system might have introduced some code (e.g., the startup code mentioned before), or because the pseudoinstructions are translated into processor instructions. For some pseudoinstructions, there is a single processor instruction. However, other pseudoinstructions may get translated into more than one processor instruction.

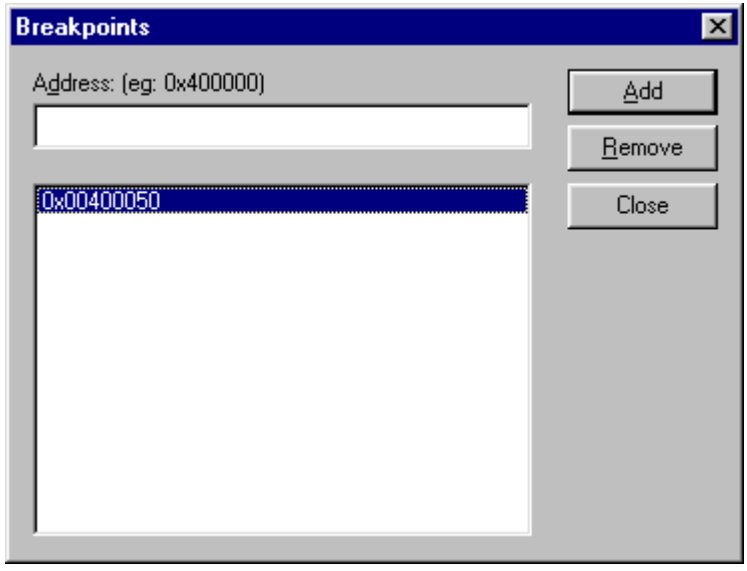


Figure G.5 Breakpoints dialog box.

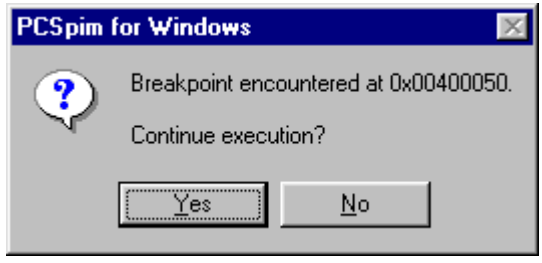


Figure G.6 Breakpoint query window.

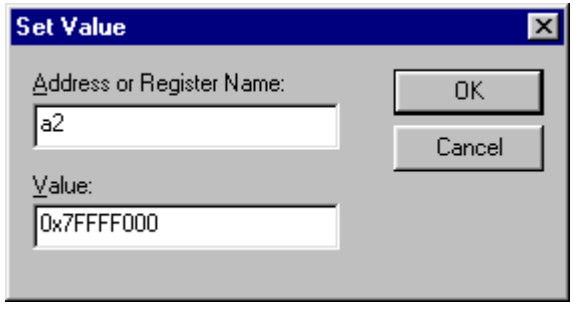


Figure G.7 Set value dialog box.

G.4 Summary

We have introduced the MIPS simulator SPIM. SPIM is a convenient tool to experience RISC assembly language programming. SPIM is available for a variety of platforms. It includes a simple debugger to facilitate single-stepping and setting breakpoints. In the last section, we have presented an overview of its debugging facilities.

G.5 Exercises

G–1 Discuss the situations where the `Multiple Step` command is useful in debugging programs.

G–2 In our setup, the run command displays the execution start address as 0x00400000. Explain why.

G–3 SPIM programs can specify the starting address either by `__start` or by `main`. Our programs used the `main` label. Discuss the differences between these two methods of specifying the execution start address.

G.6 Programming Exercises

G–P1 Take a program from Chapter 15, and ask a friend to deliberately introduce some logical errors into the program. Then use the SPIM debugger to locate and fix the errors.

G–P2 Discuss your experience debugging the MIPS and Pentium assembly language programs. Make sure to include similarities and differences between the two.

Appendix H

The SPARC Architecture

Objectives

- To discuss the 64-bit SPARC architecture;
- To describe the instruction set of SPARC processors;
- To give details about procedure invocation and parameter passing mechanisms used by SPARC processors.

This appendix gives details about the SPARC processors. Like the Itanium and MIPS processors discussed in Chapters 14 and 15, it is a RISC processor. After a brief introduction, we describe the SPARC architecture in detail. We start our discussion with a description of the register architecture in Section H.2. The following section describes its addressing modes. The SPARC instruction set details are presented in Section H.4. Section H.5 describes how procedures are invoked in the SPARC architecture. This section also provides information on SPARC's parameter passing mechanism and window management.

H.1 Introduction

The SPARC architecture was initially developed by Sun and is based on the RISC II design from the University of California, Berkeley. SPARC stands for scalable processor architecture. Unlike other companies, Sun was wise to make this an open standard and decided not to manufacture the processor itself. Sun licensed different chip manufacturers to fabricate the processor. Since SPARC is a specification at the ISA level, manufacturers can choose to design their version to suit the target price range and to improve efficiency. For example, implementation of cache is completely transparent at the ISA level.

The initial SPARC specification, introduced in 1987, was a 32-bit processor. The 64-bit SPARC version (Version 9) was introduced in 1995. In this section, we present details about the 64-bit version. Since it is a RISC processor, it uses the load/store architecture we discussed in Chapter 14.

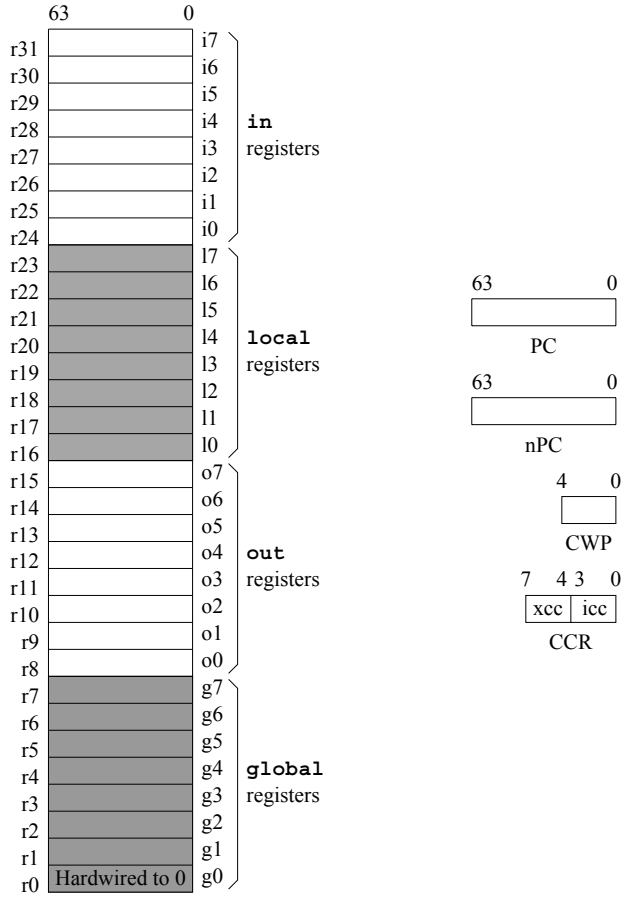


Figure H.1 Register set in the SPARC V9 processor.

H.2 Registers

At any time, a user's program sees 32 general-purpose 64-bit registers `r0` through `r31`. These 32 registers are divided into four groups of 8 registers each, as shown in Figure H.1. Conceptually, there are many similarities between the register architectures of the SPARC and the Itanium. General-purpose registers `r0` to `r7` are used as the global registers. The SPARC also maintains another set of 8 alternate global registers. The AG (alternate global) field of the processor state (PSTATE) register determines which global register set is used. As we have seen in Chapter 14, the Itanium also uses global registers (128 of them).

As does the Itanium processor, the SPARC uses the register windows described in Chapter 14 (see page 577). A register window consists of 24 registers consisting of `in`, `local`, and `out` registers. As shown in Figure H.2, the `out` registers of one set overlap with the `in` regis-

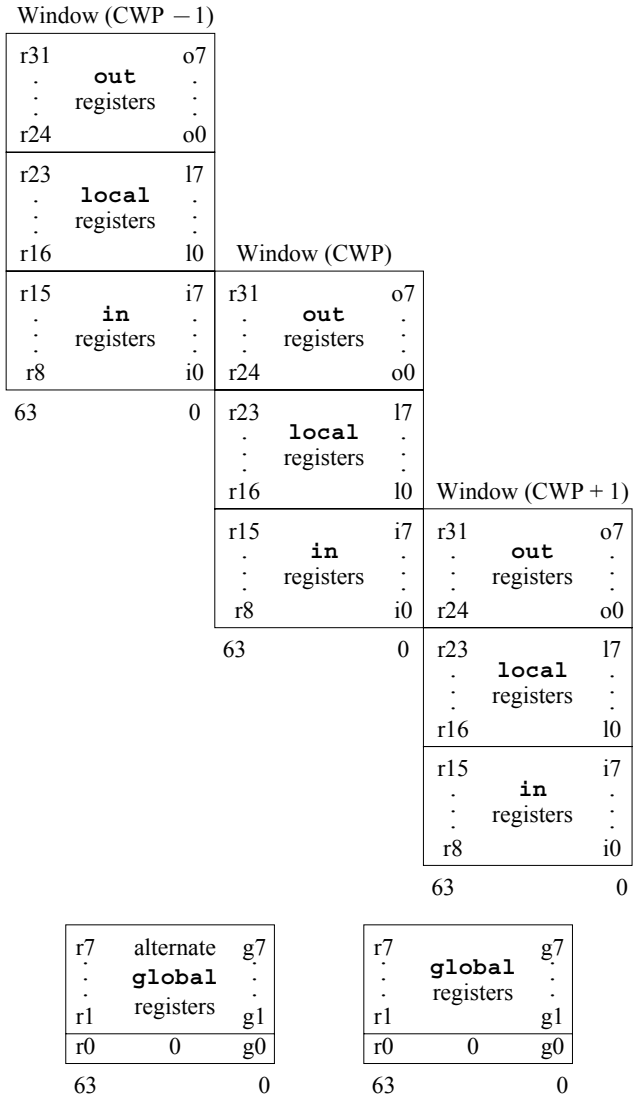


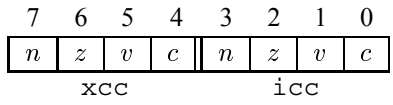
Figure H.2 Register windows in the SPARC architecture.

ters of the adjacent set. The current window pointer (CWP) register gives the current window information. The value of CWP can be incremented or decremented by two instructions: the `restore` instruction decrements the CWP register, and the `save` instruction increments it. We show details of these two instructions in Section H.5.4.

An implementation may have from 64 to 528 registers. Thus, in an implementation with the

minimum number of 64 registers, we can have three register sets (8 global, 8 alternate global, three sets of 16 registers each). The maximum number of registers allows for 32 register sets. An implementation can define NWINDOWS to specify the number of windows.

Condition Code Register: This register is analogous to the flags register in the Pentium processor, and provides two 4-bit integer condition code fields: `xcc` and `icc`. Each field consists of four bits, as shown below:



The n bit is similar to the sign flag in the Pentium. It records whether the result of the last instruction that affects the condition codes is negative ($n = 1$) or positive ($n = 0$). The `xcc` condition codes record status information about the result of an operation when the operands are 64 bits (the "x" stands for extended). The `icc` records similar information for 32-bit operands. For example, if the result is 0000 0000 F000FFFFH, the n bit of `icc` is set because the 32-bit result is a negative number. However, the 64-bit result is positive; therefore, the n bit is cleared.

The zero bit indicates whether the result is zero. As in the Pentium, $z = 1$ if the result is zero; otherwise, $z = 0$. The overflow bit v is similar to the Pentium's overflow flag. It indicates whether the result, when treated as a signed number, is within the range of 64 bits (`xcc`) or 32 bits (`icc`). The carry bit c keeps information on whether there was a carry-out from bit 63 (`xcc`) or bit 31 (`icc`). This is similar to the carry flag in the Pentium. Because of these similarities, we do not further elaborate on these bits. You can find more details about these condition code bits in Section 12.1 on page 472.

H.3 Addressing Modes

The SPARC supports only the two of the three addressing modes supported by the PowerPC and Itanium. It supports the register indirect with index and register indirect with immediate index addressing modes.

- *Register Indirect with Immediate:* This addressing mode computes the effective address as

  Effective address = contents of `Rx` + `imm13`.

  The base register `Rx` can be any general-purpose register. A 13-bit signed constant `imm13` can be specified as the displacement. This constant is sign-extended to 64 bits and added to the `Rx` register.
- *Register Indirect with Index:* This addressing mode computes the effective address as

  Effective address = contents of `Rx` + contents of `Ry`.

  The base register `Rx` and the index register `Ry` can be any general-purpose registers.

There is no register indirect addressing mode. But we can emulate this addressing mode by making the constant zero in the first addressing mode, or using `r0` as the index register in the second addressing mode. Note that `r0` is hardwired to read zero.

The main reason for not providing the register indirect addressing is the inability of the SPARC to handle 32- and 64-bit constants. As we show in the next section, SPARC instructions use the 32-bit format. Thus, we cannot even specify 32-bit constants, as we need a few bits for other fields such as the opcode. As we have seen in Chapter 14, the Itanium gets around this problem by using the extended instruction format, which uses two instructions.

To specify constants in registers, the SPARC provides a special instruction, `sethi` (set high), which stores a 22-bit value in the upper 22 bits of the destination register. To facilitate constant manipulation, SPARC assemblers provide several unary operators to extract parts of a word or extended word. These operators are as follows:

`%uhi`	Extracts bits 63 to 42 of a 64-bit extended word (i.e., extracts the high-order 22 bits of the upper word);
`%ulo`	Extracts bits 41 to 32 of a 64-bit extended word (i.e., extracts the low-order 10 bits of the upper word);
`%hi`	Extracts bits 31 to 10 of its operand (i.e., extracts the upper 22 bits);
`%lo`	Extracts bits 9 to 0 of its operand (i.e., extracts the lower 10 bits).

We are now ready to see how we can use the `sethi` instruction to load integer constants into registers. First, let us look at 32-bit words. The instruction sequence

```
sethi   %hi(value),Rd
or      Rd,%lo(value),Rd
```

stores the 32-bit constant `value` in the `Rd` register. In fact, SPARC assemblers provide the pseudoinstruction

```
set     value,Rd
```

to represent this two-instruction code sequence. If the upper 22 bits are zero, `set` uses

```
or    %g0,value,Rd
```

On the other hand, if the lower 10 bits are zero, `set` uses

```
sethi   %hi(value),Rd
```

The `setx` is similar to the `set` instruction except that it allows a 64-bit constant. The syntax is

```
setx    value,Rt,Rd
```

This instruction stores the 64-bit constant `value` in `Rd` using `Rt` as a temporary register. This pseudoinstruction can be translated into a sequence of SPARC processor instructions by using `%uhi`, `%ulo`, `%hi`, and `%lo` unary operators along with `or`, `sethi`, and `sllx` instructions. The `or` and `sllx` instructions are discussed in the next section.

H.4 Instruction Set

This section gives details about the SPARC instruction format and its instruction set. Procedure invocation and parameter passing details are discussed in the next section.

H.4.1 Instruction Format

All SPARC instructions are 32-bits long. As in the Itanium, the SPARC instruction format uses two opcode fields: The most significant two bits `op` (bits 30 and 31) identify a major operation group and the second field, `op2` or `op3`, specifies the actual instruction. A sample of the instruction formats is shown in Figure H.3. The first two formats show how most instructions that use three addresses are encoded. If an immediate value is used, the second format is used. The `i` bit specifies whether one of the source operands is in a register (`i` = 0) or a constant (`i` = 1).

The next three formats are used for `sethi` and conditional branch instructions. As we have seen before, the `sethi` instruction moves a 22-bit constant into a register. The SPARC supports two types of conditional branch instructions. These branch instructions are discussed on page 989.

The final format is used for procedure calls. We can specify a 30-bit signed displacement value as part of this instruction.

H.4.2 Data Transfer Instructions

Since the SPARC follows the load/store architecture, only load and store instructions can move data between a register and memory. The load instruction

```
ldsb    [address],Rd
```

loads the signed byte in memory at `address` into the `Rd` register. Since it is a signed byte, it is sign-extended to 64 bits and loaded into `Rd`. To load signed halfword and word, use `ldsh` and `ldsw`, respectively. The corresponding unsigned load instructions are `ldub`, `lduh`, and `lduw`. These instructions zero-extend the data to 64 bits before loading into the `Rd` register. To load a 64-bit extended word, use `ldx`.

Unlike the load instructions, store instructions do not need to be sign- or zero-extended. Thus, we do not need separate signed and unsigned versions. The store instruction

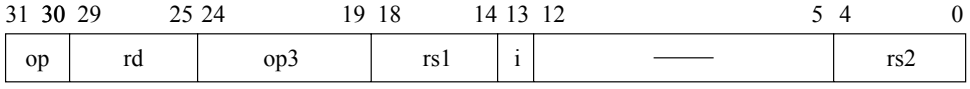
```
stb     Rs,[address]
```

stores the lower byte of `rs` in memory at `address`. To store a halfword, word, and extended word, use `sth`, `stw`, and `stx`, respectively.

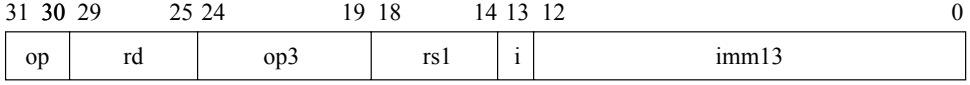
In addition to these load and store instructions, the SPARC provides two groups of conditional data movement instructions. One group moves data if the register contents satisfy a certain condition; the other group checks the condition codes. We briefly describe these two groups of instructions next.

There are five instructions in the first group. We describe one instruction as the others follow the same behavior except for the condition tested. The instruction

General Format

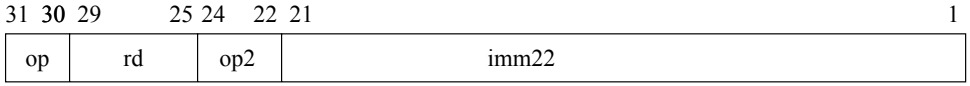


Register-register instructions (i = 0)

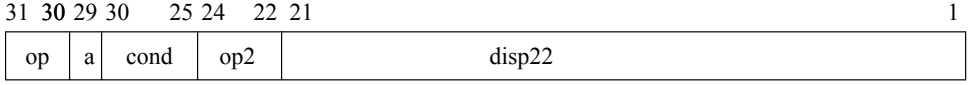


Register-immediate instructions (i = 1)

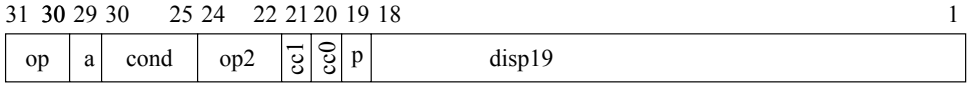
SETHI and Branch Format



SETHI instruction

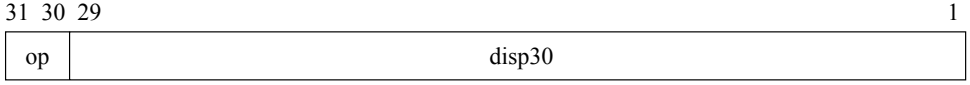


Branch instructions



Branch with prediction instructions

CALL Format



Call instructions

Figure H.3 Some sample SPARC instruction formats.

```
movrz    Rs1,Rs2,Rd    or    movrz    Rs1,imm10,Rd
```

copies the second operand (either `Rs2` or `imm10`) if the contents of `Rs1` are zero. The other instructions in this group test conditions such as "less than zero," and "greater than or equal to zero" as shown in Table H.1.

The second group of move instructions tests the condition codes. The format is

Table H.1 The SPARC's condition-testing instructions

Mnemonic	Operation	Test condition
movrz	Move if register zero	Rs1 = 0
movrnz	Move if register not zero	Rs1 $\neq$ 0
movrlz	Move if register less than zero	Rs1 $<$ 0
movrlez	Move if register less than or equal to zero	Rs1 $\leq$ 0
movrgz	Move if register greater than zero	Rs1 $>$ 0
movrgez	Move if register greater than or equal to zero	Rs1 $\geq$ 0

```
movXX    i_or_x_cc,Rs1,Rd     or     movXX    i_or_x_cc,imm11,Rd
```

which moves contents of the second operand (Rs1 or imm11) to Rd if the condition XX is satisfied. The instruction can specify whether to look at xcc or icc condition codes. The conditions tested are the same ones used in the conditional branch instruction that we discuss later. For this reason, we just give one example instruction to explain the format and semantics. The move (move if equal) instruction

```
move    i_or_x_cc,Rs1,Rd
```

moves contents of Rs1 to Rd if $z = 1$. For a full list of conditions tested by these instructions, see the conditional branch instructions in Table H.3.

H.4.3 Arithmetic Instructions

The SPARC supports the standard four types of arithmetic operations. Most arithmetic operations have two versions: one version updates the condition codes and the other does not. In most instructions, the second operand can be a register or a signed 13-bit immediate imm13.

Add Instructions: As do the other processors we have seen in this book, SPARC provides add instructions with and without the carry. The instruction

```
add    Rs1,Rs2,Rd
```

adds contents of Rs1 and Rs2 and stores the result in the Rd register. The immediate version of add is

```
add    Rs1,imm13,Rd
```

The immediate value is sign-extended to 64 bits and added to the contents of Rs1. The add instruction does not update the condition codes. If you want the operation to update the condition codes, use addcc instead of add. It updates the four integer condition codes mentioned before. In this sense, it is similar to the add instruction in the Pentium and PowerPC processors.

If you want to add the carry bit, use addc as shown below:

```
addc    Rs1,Rs2,Rd
addcc   Rs1,Rs2,Rd
```

These instructions add the *icc* carry bit. As usual, the second operand can be a signed 13-bit immediate value. This instruction is similar to the `adc` Pentium instruction.

Subtract Instructions: The SPARC provides four subtract instructions corresponding to the four `add` instructions: `sub`, `subcc`, `subc`, and `subccc`. The last two subtract instructions subtract the *icc* carry bit. The format of these instructions is similar to that of the `add` instructions. The instruction

```
sub     Rs1,Rs2,Rd
```

stores the result of `Rs1`−`Rs2` in the destination register `Rd`.

Multiplication Instructions: Unlike the other processors we have seen, the SPARC provides a single multiplication instruction for both signed and unsigned multiplication. The multiplication instruction

```
mulx    Rs1,Rs2,Rd
```

multiplies two 64-bit values in `Rs1` and `Rs2` and places the 64-bit result in `Rd`. Strictly speaking, multiplying two 64-bit values can result in a 128-bit result. This instruction, however, provides only a 64-bit result. The input operands, obviously, should be restricted to get a valid result. The SPARC-V8, which is a 32-bit processor, has two multiply instructions—one for the signed numbers and the other for unsigned numbers—as in the other processors. The multiplication instructions do not modify any condition codes.

Division Instructions: Two division instructions are available: `udivx` for unsigned numbers and `sdivx` for signed numbers. These instructions divide two 64-bit numbers and produce a 64-bit quotient. The format is

```
udivx   Rs1,Rs2,Rd
sdivx   Rs1,Rs2,Rd
```

These instructions place the result of `Rs1`÷`Rs2` in the destination register `Rd`. As with the multiplication instructions, these instructions do not modify any of the condition codes. However, these instructions generate divide-by-zero exceptions. There is no remainder computed. As we have seen before, the PowerPC also does not provide the remainder. However, we can easily compute the remainder as shown on page 586.

H.4.4 Logical Instructions

The three basic logical operations—`and`, `or`, and `xor`—are supported. These perform the bitwise logical operations. The instruction

```
and     Rs1,Rs2,Rd
```

performs the bitwise AND operation (`Rs1` AND `Rs2`) and stores the result in `Rd`.

The SPARC provides three additional logical operations: `andn`, `orn`, and `xnor`. The `andn` and `orn` operations negate the second operand before applying the specified logical operation. The `xnor` is similar to the equivalence (`equ`) instruction of the PowerPC. It is equivalent to a NOT operation followed by an XOR operation.

If you want to update the condition codes, use the `cc` versions of these instructions. As before, the second operand can be a 13-bit signed constant.

H.4.5 Shift Instructions

Two types of shift instructions are provided: 32- and 64-bit versions. The 64-bit versions use the suffix "x." For example, `sll` is the 32-bit left-shift instruction, whereas `sllx` is the 64-bit version. As in the Pentium and other processors, both left- and right-shifts are supported. The instruction

```
sll     Rs1,Rs2,Rd
```

left-shifts the lower 32 bits of `Rs1` by the shift count specified by `Rs2` and places the result in `Rd`. Note that only the least significant 5 bits of `Rs2` are taken as the shift count.

The 64-bit version

```
sllx    Rs1,Rs2,Rd
```

left-shifts the 64 bits of `Rs1`. The least significant 6 bits of `Rs2` are taken as the shift count.

For all shift instructions, the second operand can also be a constant specifying the shift count. The format is

```
sll     Rs1,count,Rd
```

The `count` is 5 bits long for 32-bit instructions and 6 bits long for the 64-bit versions.

Use `srl` and `srlx` for logical right-shift and `sra` and `srax` for arithmetic right-shift. For a discussion of the difference between the logical and arithmetic right-shifts, see our discussion in Section 9.6.5 on page 357. Unlike the Pentium, no rotate instructions are available.

H.4.6 Compare Instructions

The SPARC does not provide any compare instructions. However, SPARC assemblers provide a compare pseudoinstruction. The compare instruction

```
cmp     Rs1,Rs2       or      cmp     Rs1,imm13
```

is implemented using the `subcc` instruction as

```
subcc   Rs1,Rs2       or      subcc   Rs1,imm13
```

Table H.2 The SPARC's test-and-jump branch instructions

Mnemonic	Operation	Test condition
`brz`	Branch on register zero	`Rs1` = 0
`brlz`	Branch on register less than zero	`Rs1` $<$ 0
`brlez`	Branch on register less than or equal to zero	`Rs1` $\leq$ 0
`brnz`	Branch on register not zero	`Rs1` $\neq$ 0
`brgz`	Branch on register greater than zero	`Rs1` $>$ 0
`brgez`	Branch on register greater than or equal to zero	`Rs1` $\geq$ 0

H.4.7 Branch Instructions

The SPARC provides test-and-jump as well as set-then-jump types of branch instructions. The first group has six branch instructions. The simplest of these is shown below:

```
brz     Rs1,target
```

This instruction jumps to the specified `target` if the contents of `Rs1` are equal to zero. This transfer is achieved by updating the nPC register with the `target` address. When comparing, the values are treated as signed integers. The branch instructions are summarized in Table H.2.

The set-then-jump branch instructions check the `icc` or `xcc` condition codes. The syntax is

```
bxxx    i_or_x_cc,target
```

The `xxx` identifies the branch condition. The first operand specifies whether the `icc` or `xcc` condition codes should be used. The target address is specified as in the other branch instructions. Table H.3 shows the branch instructions in this group.

It is a good time to talk about *branch delay slots* used by most RISC processors. Consider the following C program fragment:

```
i = 10;
x = 0;
while (i >= 0)
    x = x + 35;  /* loop body */
x = 2*x;
```

This code is translated into the following assembly language version:

```
100:     add    %g0,#11,%i0   ; i = 11 (i is in i0)
104:     xor    %i1,%i1,%i1   ; x = 0 (x is in i1)
108:     brz    %g0,test      ; jump to test
  top:
```

Table H.3 SPARC's set-then-jump branch instructions

Mnemonic	Operation	Test condition
`ba`	Branch always	1 (always true)
`bn`	Branch never	0 (always false)
`bne`	Branch on not equal	NOT (Z)
`be`	Branch on equal	Z
`bg`	Branch on greater	NOT(Z OR (N XOR V))
`ble`	Branch on less or equal	Z OR (N XOR V)
`bge`	Branch on greater or equal	NOT (N XOR V)
`bl`	Branch on less	Z OR (N XOR V)
`bgu`	Branch on greater unsigned	NOT (C OR Z)
`bleu`	Branch on less or equal unsigned	C OR Z
`bcc`	Branch on carry clear (greater than or equal, unsigned)	NOT C
`bcs`	Branch on carry set (less than, unsigned)	C
`bpos`	Branch on positive	NOT N
`bneg`	Branch on negative	N
`bvc`	Branch on overflow clear	NOT V
`bvs`	Branch on overflow set	V

```
112:      add     %i1,#35,%i1   ; x = x + 35
   test:
116:      sub     %i0,#1,%i0    ; i = i - 1
120:      brgez   %i0,top       ; jump if i ≥ 0
124:      add     %i1,%i1,%i1   ; x = 2 * x
```

We use registers `i0` and `i1` for variables i and x. We should have used the local registers for this purpose but `l0` and `l1` make for confusing reading as `l` and `1` look very similar in our font. The first column gives the memory address of each instruction, assuming that the first instruction is located at address 100. The third instruction at address 108 is essentially an unconditional branch as the `g0` register is hardwired to zero. The while loop condition is tested by the other conditional branch instruction at address 120. The last instruction uses addition to multiply `x` by 2.

Table H.4 shows how this assembly code is executed. We give the contents of PC and nPC along with the instruction that is currently being executed and the one that is being fetched. You can see from this execution table that the code is not executed as intended. The two deviations are as follows:

Table H.4 A branch execution example

PC	nPC	Executing		Fetching	
100	104	`add`	`%g0,#11,%i0`	`xor`	`%i1,%i1,%i1`
104	108	`xor`	`%i1,%i1,%i1`	`brz`	`%g0,test`
108	112	`brz`	`%g0,test`	`add`	`%i1,#35,%i1`
112	116	`add`	`%i1,#35,%i1`	`sub`	`%i0,#1,%i0`
116	120	`sub`	`%i0,#1,%i0`	`brgez`	`%i0,top`
120	124	`brgez`	`%i0,top`	`add`	`%i1,%i1,%i1`
124	128	`add`	`%i1,%i1,%i1`	···	

1. The loop body instruction (at address 112, which adds constant 35 to `x`) is executed even before testing the loop condition.
2. The final instruction (at address 124), which should have been executed once, is executed during each iteration.

These two problems are caused by the execution of the instruction following the branch instruction. The reason is that, by the time the processor decodes the branch instruction, the next instruction has already been fetched. As we have seen in Chapter 8, we can improve efficiency by executing this instruction. The instruction slot after a branch is called the *delay slot*. Delay slots, however, require program modifications.

One simple solution to our problem is to do nothing (i.e., no operation) in the delay slot. We can correct our code to include a `nop` (no operation) instruction after each branch instruction, as shown below:

```
100:      add     %g0,#11,%i0   ; i = 11 (i is in i0)
104:      xor     %i1,%i1,%i1   ; x = 0 (x is in i1)
108:      brz     %g0,test      ; jump to test
112:      nop                   ; fill delay slot with a nop
   top:
116:      add     %i1,#35,%i1   ; x = x + 35
   test:
120:      sub     %i0,#1,%i0    ; i = i - 1
124:      brgez   %i0,top       ; jump if i ≥ 0
128:      nop                   ; another delay slot with a nop
132:      add     %i1,%i1,%i1   ; x = 2 * x
```

Even though we solved the problem, we defeated the purpose of the delay slot. The `nop` unnecessarily consumes processor cycles. This overhead can be substantial if the loop count is

large. Since branch instructions typically occur about 20% of the time, we would be wasting a lot of processor cycles executing `nop` instructions.

We can avoid using the `nops` if we could move the instruction before the branch to after the branch instruction. In our code we could apply this strategy to the unconditional branch instruction `brz`. However, we cannot move the `sub` instruction after the conditional branch instruction `brgez` due to the dependence on `i0` register. The resulting code is shown below:

```
100:      add     %g0,#11,%i0   ; i = 11 (i is in i0)
104:      brz     %g0,test      ; jump to test
108:      xor     %i1,%i1,%i1   ; x = 0 (x is in i1)
   top:
112:      add     %i1,#35,%i1   ; x = x + 35
   test:
116:      sub     %i0,#1,%i0    ; i = i - 1
120:      brgez   %i0,top       ; jump if i ≥ 0
124:      nop                   ; another delay slot with a nop
128:      add     %i1,%i1,%i1   ; x = 2 * x
```

This is not a great improvement because we still have the main `nop` instruction in the loop body. We could improve this code further by noticing that the `add` and `sub` instructions at addresses 112 and 116 can be interchanged. Then we could move the `add` instruction after the `brgz` branch instruction, as shown below:

```
100:      add     %g0,#11,%i0   ; i = 11 (i is in i0)
104:      brz     %g0,test      ; jump to test
108:      xor     %i1,%i1,%i1   ; x = 0 (x is in i1)
   test:
112:      sub     %i0,#1,%i0    ; i = i - 1
116:      brgez   %i0,test      ; jump if i ≥ 0
120:      add     %i1,#35,%i1   ; x = x + 35
124:      add     %i1,%i1,%i1   ; x = 2 * x
```

Although we eliminated the `nop`, we have a slight semantic problem. That is, the `add` instruction at address 120 is executed one more time than needed. We don't want to execute this instruction if the branch at 116 is not taken. We have no problem in executing this instruction when the branch is taken. Since this requirement is very common, the branch instruction can optionally specify whether the delay slot should be executed when the branch is not taken. Note that the delay slot instruction is always executed when a branch is taken. In the SPARC, we can append "`,a`" to the branch mnemonic to specify that the delay slot instruction should *not* be executed when the branch is *not* taken. The correct code is shown below:

```
100:      add      %g0,#11,%i0   ; i = 11 (i is in i0)
104:      brz      %g0,test      ; jump to test
```

```
108:      xor      %i1,%i1,%i1   ; x = 0 (x is in i1)
   test:
112:      sub      %i0,#1,%i0    ; i = i - 1
116:      brgez,a  %i0,test      ; jump if i ≥ 0
120:      add      %i1,#35,%i1   ; x = x + 35
124:      add      %i1,%i1,%i1   ; x = 2 * x
```

Note that the specification of "`,a`" does not change the behavior when the branch is taken; in this case, the delay slot instruction is always executed. But specifying "`,a`" annuls the delay slot instruction only when the branch is not taken.

As in the Itanium, the SPARC allows providing a hint to the hardware as to whether the branch is likely to be taken. To convey this information, append "`pt`" for branch taken hint or "`pn`" for branch not taken hint. The default is branch taken. Thus `brgez,a,pt` is equivalent to `brgez,a`. If you want to give the branch not taken hint, use `brgez,a,pn` instead. If you skipped the Itanium details presented in Chapter 14, at least read Section 8.4.2 for details on branch prediction strategies and their impact on performance.

H.5 Procedures and Parameter Passing

This section presents details about procedure invocation, parameter passing, and register window management.

H.5.1 Procedure Instructions

Procedures in the SPARC can be invoked either by the `call` or `jmpl` instruction. The call instruction is similar to that supported by other processors we have discussed. It takes a label identifying the called procedure. The format is

```
call     procName
```

As shown in Figure H.3, the called procedure's displacement is expressed as a 30-bit signed number. This displacement is PC-relative. The SPARC requires procedures to be word-aligned (i.e., the procedure address is a multiple of 4). It multiplies the 30-bit displacement value by 4 and adds to the contents of PC to get the procedure address. As in the branch instructions, the call is also delayed. Thus, before the procedure is invoked, the instruction in the delay slot is executed. This delayed execution is achieved by placing the target address in nPC.

To facilitate return from a procedure, the `call` instruction stores the PC value (i.e., address of the `call` instruction itself) in `o7` (really `r15`). The following summarizes the actions taken by the `call` instruction:

```
nPC = PC + 4*30-bit displacement
r15 = PC
```

The `call` instruction allows only direct addressing. The `jmpl` instruction allows more flexibility. It allows indirect procedure calls (as in the Pentium) as well as the specification of a 32-bit target address. The format is

```
jmpl    address,register
```

The target `address` can be specified in either of the two addressing modes allowed by the SPARC. It jumps to the 32-bit address given by `address` and leaves the current PC value, which is the address of the `jmpl` instruction itself, in `register`. To call a procedure indirectly, use

```
jmpl    register,%r15
```

This causes transfer of control to the address in `register` and leaves the return address in `r15` as in the `call` instruction.

We can also use the `jmpl` instruction to return from a procedure. For example, the instruction

```
jmpl    %r15+8,%g0
```

adds 8 to the contents of `r15` and delay jumps to that address. The current PC address is written to `g0`, which means it is ignored. We add 8 to the return address in `r15` because `r15` points to the `call`/`jmpl` instruction that called the procedure. Also, we have to skip the following delay-slot instruction. Assemblers typically provide a `ret` pseudoinstruction, which is translated into this particular `jmpl` instruction.

The SPARC also provides a return instruction that takes a return address as an operand. The format is

```
return    address
```

For example, instead of the assembler-provided `ret` instruction, we can also use

```
return    %r15+8
```

SPARC assemblers provide two pseudoinstructions (also called synthetic instructions) to facilitate return from procedures. The first return instruction

```
ret       is implemented as    jmpl    %i7+8,%g0
```

There is a special return instruction `retl` to return from a leaf procedure. A leaf procedure is a procedure that does not call any other procedure. The

```
retl      is implemented as    jmpl    %o7+8,%g0
```

We show an example use of these instructions later.

H.5.2 Parameter Passing

By convention, the first six arguments are passed in the `out` registers. The remaining arguments, if any, are passed via the stack. Recall that the caller's eight `out` registers become the callee's `in` registers. The following summarizes the usage of these registers:

Caller	Callee	Usage
`%o0`	`%i0`	First argument
`%o1`	`%i1`	Second argument
`%o2`	`%i2`	Third argument
`%o3`	`%i3`	Fourth argument
`%o4`	`%i4`	Fifth argument
`%o5`	`%i5`	Sixth argument
`%o6`	`%i6`	Stack pointer/frame pointer
`%o7`	`%i7`	Return address − 8

The `o6` is referred to as the stack pointer and can be accessed by its `sp` alias. The `i6` is referred to as the frame pointer by the callee and we can use the alias `fp` to access it.

We can return up to six values via the registers as shown below:

Caller	Callee	Usage
`%o0`	`%i0`	First return value
`%o1`	`%i1`	Second return value
`%o2`	`%i2`	Third return value
`%o3`	`%i3`	Fourth return value
`%o4`	`%i4`	Fifth return value
`%o5`	`%i5`	Sixth return value

As with the parameter passing, we have to use the stack to return the remaining return values.

H.5.3 Stack Implementation

We briefly describe the SPARC stack implementation. For more details on the stack, see our discussion in Chapter 10. As in the Pentium, the stack grows downward (i.e., from a higher memory address to a lower address). However, there are no explicit stack push and pop instructions. Instead, these instructions can be synthesized by manipulating the stack pointer `sp`. For example, to push or pop the contents of `i0`, we can use the following code:

`push` operation

```
sub     %sp,4,%sp
st      %i0,[%sp]
```

`pop` operation

```
add     %sp,4,%sp
ld      [%sp],%i0
```

As in the Pentium, to allocate an N-byte stack frame, we can use

```
sub     %sp,N,%sp
```

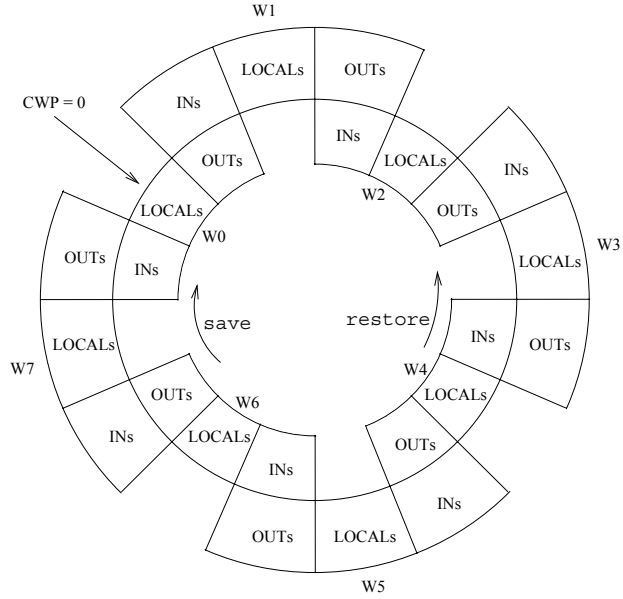


Figure H.4 The register windows are organized as a circular buffer.

H.5.4 Window Management

SPARC processors can have up to 32 register windows. The number of windows available on a specific implementation is given by NWINDOWS. Note that NWINDOWS can range from 3 to 32. As noted, the current window pointer (CWP) points to the current register set. These window sets are organized as a circular buffer (see Figure H.4). Thus, the CWP arithmetic can be done modulo NWINDOWS.

With each procedure call, a new register window is assigned. This is done by the `save` instruction. This instruction can also allocate space on the stack for the stack frame. The `save` instruction

```
save    %sp,−N,%sp
```

slides the register window by incrementing CWP (mod NWINDOWS) and allocates N bytes of stack space. If no errors occur, `save` acts as the `add` instruction does. Thus, by specifying `sp` and a negative N value, it allocates N bytes of stack space. As in the `add` instruction, the second operand can also be a register.

The `restore` instruction restores the register window saved by the last `save` instruction. Its format is similar to that of the `save`. It also performs addition on its operands as does the `save` instruction. A trivial `restore` pseudoinstruction is defined as

```
restore    %g0,%g0,%g0
```

A typical procedure looks like

```
proc-name:
        save    %sp,−N,%sp
        . . .
        procedure body
        . . .
        ret
        restore
```

Note that the `restore` instruction is executed in the delay slot. Since the `restore` does not add N to `sp`, you might be wondering about how the stack allocation is released. To understand this, we should look at the way the `save` instruction performs the `add` operation on its operands. For this `add` operation, `save` uses the *old window* for the two source operands and stores the result in the *new window*. In our example, it adds $-N$ to the `sp` value from the previous window and stores the result in the new window's `sp` register. Thus, when we `restore` the previous window, we automatically see the previous `sp` value.

A leaf procedure does not use a new register window. It uses the registers from the caller's window. A typical leaf procedure looks like

```
proc-name:
        . . .
        procedure body
        . . .
        retl /* use retl, not ret */
```

What happens if the `save` instruction cannot get a new window of registers? For this reason, the stack frame maintains space for the in, local, and six arguments. In addition, there is a "hidden parameter" to return a structure pointer. Thus, a minimum of $(16 + 1 + 6) * 8 = 184$ bytes of stack frame is needed. Additional space may be needed for storing temporaries, more arguments, and so on, as shown in Figure H.5.

We give an example procedure to illustrate how these instructions are used. We use the following C code consisting of three procedures:

```
        . . .
 i = condSum (1, 2, 3, 4)
        . . .

int condSum(int a, int b, int c, int d)
{
    int  t1;

    t1 = a;
```

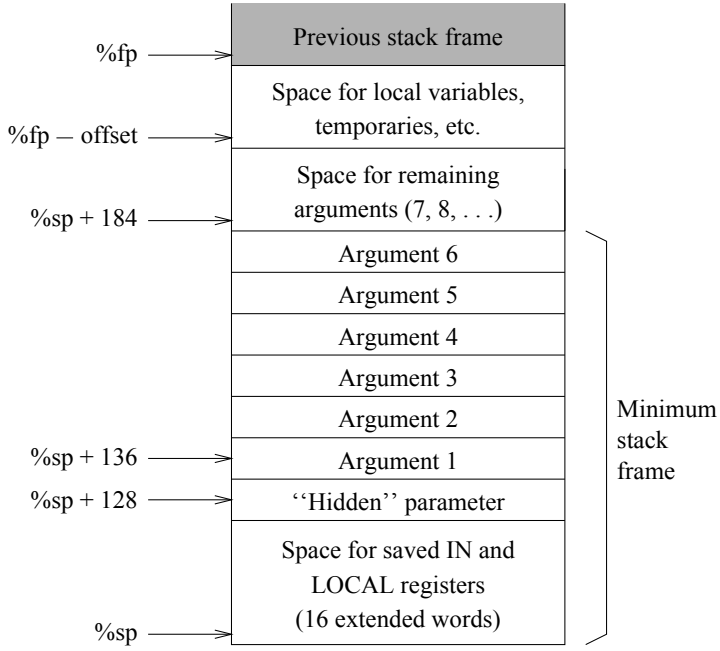


Figure H.5 SPARC's stack frame.

```
    if (a < b)
        t1 = b;
    return(sum(t1,c,d));
}

int sum(int x, int y, int z)
{
    return(x+y+z);
}
```

The corresponding SPARC assembly code is shown in three procedures. In the `main` procedure, the four arguments of `condSum` are moved to the first four `out` registers: `o0` through `o3`.

```
        . . .
mov     1,%o0       ; first argument
mov     2,%o1       ; second argument
mov     3,%o2       ; third argument
call    condSum
mov     4,%o3       ; fourth argument in delay slot
```

```
    ; condSum returns result in %o0. When condSum returns,
    ; the following instruction is executed
    mov     %o0,%l0     ; return result moved to %l0
            . . .
```

Since the `call` is a delayed instruction, we use the delay slot to move the fourth argument to `o3`. Note that the `condSum` call should return to the instruction

```
mov     %o0,%l0
```

This is the reason for adding 8 to the return address. This procedure returns the sum in the `o0` register. The above `mov` instruction copies this value to the `l0` local register.

The first instruction in the `condSum` procedure is the `save` instruction. As we have discussed, it allocates a new register window by incrementing CWP. We also allocate 184 bytes of stack frame, which is the minimum size. The next four instructions select the minimum of the first two arguments. Note that the `out` registers of the previous window are referred to as `in` registers in the current window. We move the three arguments to `out` registers to pass them on to the `sum` procedure. When `sum` returns the total, this value is moved from `o0` to `i0` so that the result is available in `o0` in the main procedure.

```
;*********** condSum procedure **********
condSum:
    save    %sp,-184,%sp; allocate min. stack frame
    mov     %i0, %o0
    cmp     %i1, %i0    ; if %i1 is less/equal,
    ble     skip        ; skip the following mov
    mov     %i1, %o0
skip:
    mov     %i2, %o1    ; second argument
    call    sum
    mov     %i3, %o2    ; third argument in delay slot
    mov     %o0, %i0    ; move the result returned by sum to %o0
    ret
    restore             ; trivial restore in delay slot of ret
;******* end of condSum procedure *******
```

The `sum` procedure is a leaf procedure as it does not call any other procedure. We can optimize a leaf procedure by not requesting a new window; instead it uses the registers from the caller's window. Thus, there is no need for the `save` and `restore` instructions. The only instruction that needs special care is the return: we have to use `retl` rather than the `ret` instruction.

```
;*********** sum procedure ***********
sum:
     add     %o0, %o1, %o0 ; first addition
     retl                  ; leaf procedure, use retl
     add     %o0, %o2, %o0 ; final add in delay slot
     ; result returned in %o0
;******* end of sum procedure ********
```

H.6 Summary

We have briefly presented the architecture and programming of SPARC processors. The SPARC's register window mechanism is similar to that of the Itanium processor discussed in Chapter 14. Note that the SPARC specification preceded the design of the Itanium. A notable difference is that it is a *specification* at the ISA level that is available to chip manufacturers, which means several implementations are possible. For example, an implementation can choose to have a number of register windows between 3 and 32. There are also several instructions that have implementation-defined semantics. If you are interested in more details, several reference documents are available at the SPARC Web site.

H.7 Web Resources

Full specifications and other reference material on the SPARC architecture are available form `www.sparc.org`. Sun also maintains information on their SPARC processors at `www.sun.com/microelectronics/sparc`.

H.8 Exercises

H–1 Discuss the similarities and differences between the register architectures of the SPARC and Itanium.

H–2 Unlike the Itanium and MIPS processors, the SPARC does not provide the register indirect addressing mode. Explain how we can emulate this addressing mode in the SPARC.

H–3 Discuss the need for the operators like `%uhi` and `%ulo`.

H–4 The SPARC does not provide rotate instructions as does the Pentium. Write a SPARC code fragment to rotate the contents of register `i1` right by one bit position.

H–5 Write a code fragment that implements the pseudoinstruction `setx` given on page 983.

Appendix I

Pentium Instruction Set

Objectives

- To describe the Pentium instruction format;
- To present selected Pentium instructions.

Instruction format and encoding encompass a variety of factors: addressing modes, number of operands, number of registers, sources of operands, and so on. Instructions can be of fixed length or variable length. The Pentium uses variable-length instructions. The instruction length is varied to accommodate the complexity of the instruction. Section I.1 discusses the instruction format of the Pentium processor. A subset of the Pentium instruction set is presented in Section I.2.

I.1 Pentium Instruction Format

The Pentium uses variable-length instructions. Instruction length can range between 1 and 16 bytes. The instruction format of the Pentium is shown in Figure I.1. The next two subsections discuss the instruction format in detail.

I.1.1 Instruction Prefixes

There are four instruction prefixes, as shown in Figure I.1*a*. These prefixes can appear in any order. All four prefixes are optional. When a prefix is present, it takes a byte.

- *Instruction Prefixes*: Instruction prefixes such as `rep` were discussed in Chapter 12. This group of prefixes consists of `rep`, `repe/repz`, `repne/repnz`, and `lock`. The three repeat prefixes were discussed in detail in Chapter 12. The `lock` prefix is useful in multiprocessor systems to ensure exclusive use of shared memory.
- *Segment Override Prefixes*: These prefixes are used to override the default segment association. For example, DS is the default segment for accessing data. We can override this

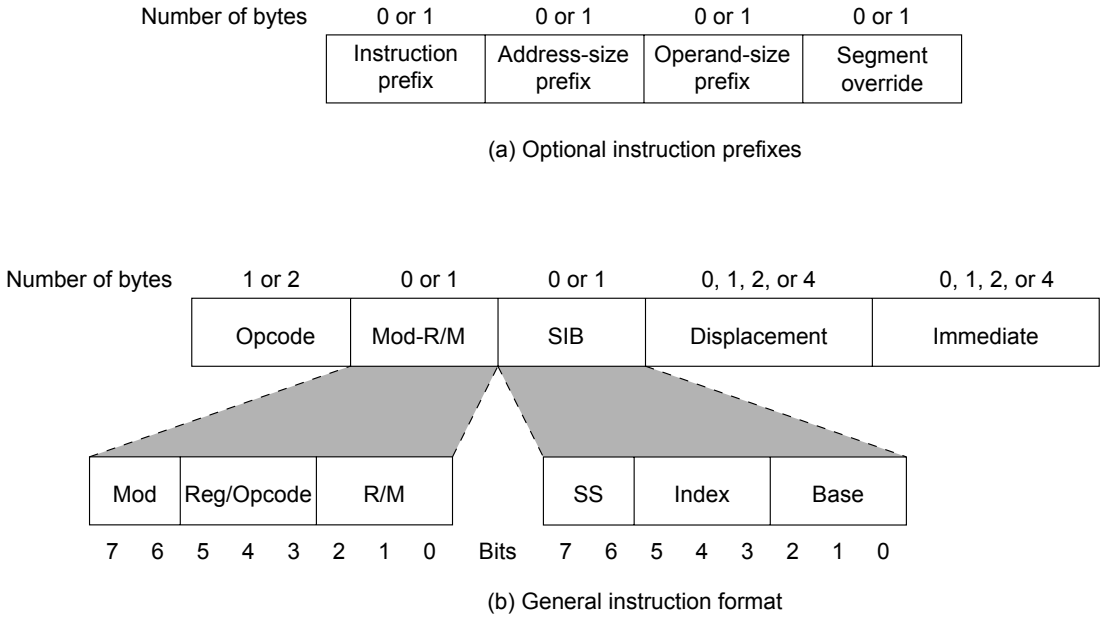


Figure I.1 Pentium instruction format.

by using a segment prefix. We saw an example of this in Chapter 10 (see Program 10.6 on page 419). The following segment override prefixes are available: CS, SS, DS, ES, FS, and GS. Chapter 9 gives details on segment override prefixes (see page 337).

- *Address-Size Override Prefix*: This prefix is useful in overriding the default address size. As discussed on page 267, the D bit indicates the default address and operand size. This prefix is used to switch between 16- and 32-bit addresses.
- *Operand-Size Override Prefix*: The use of this prefix allows us to switch from the default operand size to the other. For example, in the 16-bit operand mode, we can use a 32-bit register by prefixing the instruction with the operand-size override prefix. Chapter 11 gives details on the operand- and address-size override prefixes (see page 437).

I.1.2 General Instruction Format

The general instruction format shown in Figure I.1*b* consists of five fields.

- *Opcode*: This field can be one or two bytes long. This is the only field that must be present in every instruction. For example, the opcode for the `popa` instruction is 61H, and takes only one byte. On the other hand, the opcode for the `shld` instruction takes two bytes (the opcode is 0FA4H). The opcode field also contains other smaller encoding fields. These fields include register encoding, direction of operation (to or from memory), size

of displacement, and whether the immediate data must be sign-extended. For example, the instructions

```
push    AX
push    CX
push    DX
push    BX
```

are encoded as 50H, 51H, 52H, and 53H, respectively. Each instruction takes only one byte that includes the operation code (push) as well as the register encoding (AX, CX, DX, or BX).

- *Mod R/M*: This field along with the SIB byte provides addressing information. The Mod R/M byte consists of three fields.
  - *Mod*: This field (two bits) along with the R/M field (three bits) specifies one of 32 possible choices: eight registers and 24 indexing modes.
  - *Reg/Opcode*: This field (three bits) specifies either a register number or three more bits of opcode information. The first byte of the instruction determines the meaning of this field.
  - *R/M*: This field (three bits) either specifies a register as the location of the operand or forms part of the addressing-mode encoding along with the Mod field.
- *SIB*: The based-indexed and scaled-indexed modes of 32-bit addressing require this byte. Certain encodings of the Mod R/M byte indicate the presence of the SIB byte. The SIB byte consists of three fields, as shown in Figure I.1. The SS field (two bits) specifies the scale factor (1, 2, 4, or 8). The index and base fields (three bits each) specify the index and base registers, respectively.
- *Displacement*: For instructions that need a displacement, this field provides the required value. When present, it is an 8-, 16-, or 32-bit signed integer. For example,

```
        jg    SHORT done
        pop   BX
done:
```

  generates the code 7F 01 for the `jg` conditional jump instruction. The opcode for `jg` is 7FH and the displacement is 01 because the `pop` instruction encoding takes only a single byte.
- *Immediate*: The immediate field is present in those instructions that specify an immediate operand. When present, it is an 8-, 16-, or 32-bit operand. For example,

```
mov    AX,256
```

  is encoded as `B8 0100`. Note that the first byte B8 not only identifies the instruction as `mov` but also specifies the destination register AX (by the least significant three bits of the opcode byte). The Pentium uses the following encoding for the 16-bit registers:

AX = 0	SP = 4
CX = 1	BP = 5
DX = 2	SI = 6
BX = 3	DI = 7.

The last two bytes represent the immediate value 256, which is equal to 100H. If we change the register from AX to BX, the opcode byte changes from B8 to BB.

I.2 Selected Pentium Instructions

This section gives selected Pentium instructions in alphabetical order. For each instruction, the instruction mnemonic, flags affected, format, and a description are given. For a more detailed discussion, please refer to the *Pentium Processor Family Developer's Manual—Volume 3: Architecture and Programming Manual*. Although most of the components are self-explanatory, the flags section requires some explanation regarding the notation used. An instruction can affect a flag bit in one of several ways. We use the following notation to represent the effect of an instruction on a flag bit.

0	—	Cleared;
1	—	Set;
–	—	Unchanged;
M	—	Updated according to the result;
*	—	Undefined.

adc — Add with carry

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
adc     dest,src
```

Description:

Performs integer addition of `src` and `dest` with the carry flag. The result (`dest` + `src` + CF) is assigned to `dest`. Clock cycles: 1 to 3.

add — Add without carry

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
add     dest,src
```

Description:

Performs integer addition of `src` and `dest`. The result (`dest` + `src`) is assigned to `dest`. Clock cycles: 1 to 3.

and — Logical bitwise and

C	O	Z	S	P	A
0	0	M	M	M	*

Format:

```
and     dest,src
```

Description:

Performs logical bitwise **and** operation. The result `src` **and** `dest` is stored in `dest`. Clock cycles: 1 to 3

bsf — Bit scan forward

C	O	Z	S	P	A
*	*	M	*	*	*

Format:

```
bsf     dest,src
```

Description:

Scans the bits in `src` starting with the least significant bit. The ZF flag is set if all bits are 0; otherwise, ZF is cleared and the `dest` register is loaded with the bit index of the first set bit. Note that `dest` and `src` both must be either 16- or 32-bit operands. Although the `src` operand can be either in a register or memory, `dest` must be a register. Clock cycles: 6 to 35 for 16-bit operands and 6 to 43 for 32-bit operands.

bsr — Bit scan reverse

C	O	Z	S	P	A
*	*	M	*	*	*

Format:

```
bsr     dest,src
```

Description:

Scans the bits in `src` starting with the most significant bit. The ZF flag is set if all bits are 0; otherwise, ZF is cleared, and the `dest` register is loaded with the bit index of the first set bit when scanning `src` in the reverse direction. Note that `dest` and `src` both must be either 16- or 32-bit operands. Although the `src` operand can be either in a register or memory, `dest` must be a register. Clock cycles: 7 to 40 for 16-bit operands and 7 to 72 for 32-bit operands.

bswap — Byte swap

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
bswap    src
```

Description:

Reverses the byte order of a 32-bit register `src`. This effectively converts a value from little-endian to big-endian and vice versa. Note that `src` must be a 32-bit register. Result is undefined if a 16-bit register is used. Clock cycles: 1.

bt — Bit test

C	O	Z	S	P	A
M	–	–	–	–	–

Format:

```
bt    src1,src2
```

Description:

The value of the bit in `src1`, whose position is indicated by `src2`, is saved in the carry flag. The first operand `src1` can be a 16- or 32-bit value that is either in a register or memory. The second operand `src2` can be a 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 4 to 9.

btc — Bit test and complement

C	O	Z	S	P	A
M	–	–	–	–	–

Format:

```
btc    src1,src2
```

Description:

The value of the bit in `src1`, whose position is indicated by `src2`, is saved in the carry flag and then the bit in `src1` is complemented. The first operand `src1` can be a 16- or 32-bit value that is either in a register or memory. The second operand `src2` can be a 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 7 to 13.

btr — Bit test and reset

C	O	Z	S	P	A
M	–	–	–	–	–

Format:

```
btr     src1,src2
```

Description:

The value of the bit in `src1`, whose position is indicated by `src2`, is saved in the carry flag and then the bit in `src1` is reset (i.e., cleared). The first operand `src1` can be a 16- or 32-bit value that is either in a register or memory. The second operand `src2` can be a 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 7 to 13.

bts — Bit test and set

C	O	Z	S	P	A
M	–	–	–	–	–

Format:

```
bts     src1,src2
```

Description:

The value of the bit in `src1`, whose position is indicated by `src2`, is saved in the carry flag and then the bit in `src1` is set (i.e., stores 1). The first operand `src1` can be a 16- or 32-bit value that is either in a register or memory. The second operand `src2` can be a 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 7 to 13.

call — Call procedure

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
call    dest
```

Description:

The `call` instruction causes the procedure in the operand to be executed. There is a variety of call types. We indicated that the flags are not affected by `call`. This is true only if there is no task switch. For more details on the `call` instruction, see Chapter 10. For details on other forms of call, see the Pentium data book. Clock cycles: vary depending on the type of call.

cbw — Convert byte to word

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
cbw
```

Description:

Converts the signed byte in AL to a signed word in AX by copying the sign bit of AL (the most significant bit) to all bits of AH. Clock cycles: 3.

cdq — Convert doubleword to quadword

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
cdq
```

Description:

Converts the signed doubleword in EAX to a signed quadword in EDX:EAX by copying the sign bit of EAX (the most significant bit) to all bits of EDX. Clock cycles: 2.

clc — Clear carry flag

C	O	Z	S	P	A
0	–	–	–	–	–

Format:

```
clc
```

Description:

Clears the carry flag. Clock cycles: 2.

cld — Clear direction flag

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
cld
```

Description:

Clears the direction flag. Clock cycles: 2.

cli — Clear interrupt flag

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
cli
```

Description:

Clears the interrupt flag. Note that maskable interrupts are disabled when the interrupt flag is cleared. Clock cycles: 7.

cmc — Complement carry flag

C	O	Z	S	P	A
M	–	–	–	–	–

Format:

```
cmc
```

Description:

Complements the carry flag. Clock cycles: 2.

cmp — Compare two operands

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
cmp     dest,src
```

Description:

Compares the two operands specified by performing `dest – src`. However, the result of this subtraction is not stored (unlike the `sub` instruction) but only the flags are updated to reflect the result of the subtract operation. This instruction is typically used in conjunction with conditional jumps. If an operand greater than 1 byte is compared to an immediate byte, the byte value is first sign-extended. Clock cycles: 1 if no memory operand is involved; 2 if one of the operands is in memory.

cmps — Compare string operands

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
cmps    dest,src
cmpsb
cmpsw
cmpsd
```

Description:

Compares the byte, word, or doubleword pointed by the source index register (SI or ESI) with an operand of equal size pointed by the destination index register (DI or EDI). If the address size is 16 bits, SI and DI registers are used; ESI and EDI registers are used for 32-bit addresses. The comparison is done by subtracting the operand pointed by the DI or EDI register from that by the SI or ESI register. That is, the `cmps` instructions performs either [SI] − [DI] or [ESI] − [EDI]. The result is not stored but used to update the flags, as in the `cmp` instruction. After the comparison, both source and destination index registers are automatically updated. Whether these two registers are incremented or decremented depends on the direction flag (DF). The registers are incremented if DF is 0 (see the `cld` instruction to clear the direction flag); if the DF is 1, both index registers are decremented (see the `std` instruction to set the direction flag). The two registers are incremented or decremented by 1 for byte comparisons, 2 for word comparisons, and 4 for doubleword comparisons.

Note that the specification of the operands in `cmps` is not really required as the two operands are assumed to be pointed by the index registers. The `cmpsb`, `cmpsw`, and `cmpsd` are synonyms for the byte, word, and doubleword `cmps` instructions, respectively.

The repeat prefix instructions (i.e., `rep`, `repe`, or `repne`) can precede the `cmps` instructions for array or string comparisons. See the `rep` instruction for details. Clock cycles: 5.

cwd — Convert word to doubleword

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
cwd
```

Description:

Converts the signed word in AX to a signed doubleword in DX:AX by copying the sign bit of AX (the most significant bit) to all bits of DX. In fact, `cdq` and this instruction use the same opcode (99H). Which one is executed depends on the default operand size. If the operand size is 16 bits, `cwd` is performed; `cdq` is performed for 32-bit operands. Clock cycles: 2.

cwde — Convert word to doubleword

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
cwde
```

Description:

Converts the signed word in AX to a signed doubleword in EAX by copying the sign bit of AX (the most significant bit) to all bits of the upper word of EAX. In fact, `cbw` and `cwde` are the same instructions (i.e., share the same opcode of 98H). The action performed depends on the operand size. If the operand size is 16 bits, `cbw` is performed; `cwde` is performed for 32-bit operands. Clock cycles: 3.

dec — Decrement by 1

C	O	Z	S	P	A
–	M	M	M	M	M

Format:

```
dec     dest
```

Description:

The `dec` instruction decrements the `dest` operand by 1. The carry flag is not affected. Clock cycles: 1 if `dest` is a register; 3 if `dest` is in memory.

div — Unsigned divide

C	O	Z	S	P	A
*	*	*	*	*	*

Format:

```
div     divisor
```

Description:

The `div` instruction performs unsigned division. The divisor can be an 8-, 16-, or 32-bit operand, located either in a register or in memory. The dividend is assumed to be in AX (for byte divisor), DX:AX (for word divisor), or EDX:EAX (for doubleword divisor). The quotient is stored in AL, AX, or EAX for 8-, 16-, and 32-bit divisors, respectively. The remainder is stored in AH, DX, or EDX for 8-, 16-, and 32-bit divisors, respectively. It generates interrupt 0 if the result cannot fit the quotient register (AL, AX, or EAX), or if the divisor is zero. See Chapter 12 for details. Clock cycles: 17 for an 8-bit divisor, 25 for a 16-bit divisor, and 41 for a 32-bit divisor.

enter — Create procedure stack frame

C	O	Z	S	P	A
*	*	*	*	*	*

Format:

```
enter    bytes,level
```

Description:

The `enter` instruction creates stack frames for procedures. The first operand `bytes`, which is a 16-bit immediate value, specifies the number of bytes of local storage space allocated for the procedure. The second operand `level` gives the nesting level of the procedure. The nesting level, which can range from 0 to 31, determines the number of stack frame pointers copied into the new stack frame. More details on this instruction are in Chapter 10 (see pages 407 and 424). Clock cycles: 11 for `level` = 0, 15 for `level` = 1, and 15 + 2*`level` for higher levels.

hlt — Halt

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
hlt
```

Description:

This instruction halts instruction execution indefinitely. An interrupt or a reset will enable instruction execution. Clock cycles: ∞.

idiv — Signed divide

C	O	Z	S	P	A
*	*	*	*	*	*

Format:

```
idiv    divisor
```

Description:

Similar to `div` instruction except that `idiv` performs signed division. The divisor can be an 8-, 16-, or 32-bit operand, located either in a register or in memory. The dividend is assumed to be in AX (for byte divisor), DX:AX (for word divisor), or EDX:EAX (for doubleword divisor). The quotient is stored in AL, AX, or EAX for 8-, 16-, and 32-bit divisors, respectively. The remainder is stored in AH, DX, or EDX for 8-, 16-, and 32-bit divisors, respectively. It generates interrupt 0 if the result cannot fit the quotient register (AL, AX, or EAX), or if the divisor is zero. See Chapter 12 for details. Clock cycles: 22 for an 8-bit divisor, 30 for a 16-bit divisor, and 46 for a 32-bit divisor.

imul — Signed multiplication

C	O	Z	S	P	A
M	M	*	*	*	*

Format:

```
imul    src
imul    dest,src
imul    dest,src,constant
```

Description:

This instruction performs signed multiplication. The number of operands for `imul` can be between one and three, depending on the format used. In the one-operand format, the other operand is assumed to be in the AL, AX, or EAX register depending on whether the `src` operand is 8-, 16-, or 32-bits long, respectively. The `src` operand can be either in a register or in memory. The result, which is twice as long as the `src` operand, is placed in AX, DX:AX, or EDX:EAX for 8-, 16-, or 32-bit `src` operands, respectively. In the other two forms, the result is of the same length as the input operands.

The two-operand format specifies both operands required for multiplication. In this case, `src` and `dest` both must be either 16- or 32-bit operands. Although `src` can be either in a register or memory, `dest` must be a register.

In the three-operand format, a constant can be specified as an immediate operand. The result (`src` × `constant`) is stored in `dest`. As in the two-operand format, the `dest` operand must be a register. The `src` can be either in a register or memory. The immediate constant can be an 8-, 16-, or 32-bit value. For additional restrictions, refer to the Pentium data book. Clock cycles: 10 (11 if the one-operand format is used with either 8- or 16-bit operands).

in — Input from a port

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
in    dest,port
in    dest,DX
```

Description:

This instruction has two formats. In both formats, `dest` must be in the AL, AX, or EAX register. In the first format, it reads a byte, word, or doubleword from `port` into the AL, AX, or EAX register, respectively. Note that `port` is an 8-bit immediate value. This format is restrictive in the sense that only the first 256 ports can be accessed. The other format is more flexible and allows access to the complete I/O space (i.e., any port between 0 and 65,535). In this format, the port number is assumed to be in the DX register. Clock cycles: varies; see Pentium data book.

inc — Increment by 1

C	O	Z	S	P	A
–	M	M	M	M	M

Format:

```
inc     dest
```

Description:

The `inc` instruction increments the `dest` operand by 1. The carry flag is not affected. Clock cycles: 1 if `dest` is a register; 3 if `dest` is in memory.

ins — Input from a port to string

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
insb
insw
insd
```

Description:

This instruction transfers 8-, 16-, or 32-bit data from the input port specified in the DX register to a location in memory pointed by ES:(E)DI. The DI index register is used if the address size is 16 bits and the EDI index register for 32-bit addresses. Unlike the `in` instruction, the `ins` instruction does not allow the specification of the port number as an immediate value. After the data transfer, the index register is updated automatically. The index register is incremented if DF is 0; it is decremented if DF is 1. The index register is incremented or decremented by 1, 2, or 4 for byte, word, or doubleword operands, respectively. The repeat prefix can be used along with the `ins` instruction to transfer a block of data (the number of data transfers is indicated by the CX register; see the `rep` instruction for details). Clock cycles: varies; see Pentium data book.

int — Interrupt

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
int     interrupt-type
```

Description:

The `int` instruction calls an interrupt service routine or handler associated with `interrupt-type`. The `interrupt-type` is an immediate 8-bit operand. This value is used as an index into the interrupt descriptor table (IDT). See Chapter 20 for details on the interrupt invocation mechanism. Clock cycles: varies; see Pentium data book.

into — Interrupt on overflow

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
into
```

Description:

The `into` instruction is a conditional software interrupt identical to `int 4` except that the `int` is implicit and the interrupt handler is invoked conditionally only when the overflow flag is set. Clock cycles: varies; see Pentium data book.

iret — Interrupt return

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
iret
iretd
```

Description:

The `iret` instruction returns control from an interrupt handler. In real address mode, it loads the instruction pointer and the flags register with values from the stack and resumes the interrupted routine. Both `iret` and `iretd` are synonymous (and use the opcode CFH). The operand size in effect determines whether the 16- or 32-bit instruction pointer (IP or EIP) and flags (FLAGS or EFLAGS) are be used. See Chapter 20 for more details. This instruction affects all flags as the flags register is popped from stack. Clock cycles: varies; see Pentium data book.

jcc — Jump if condition cc is satisfied

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
jcc     target
```

Description:

The `jcc` instruction alters program execution by transferring control conditionally to the `target` location in the same segment. The `target` operand is a relative offset (relative to the instruction following the conditional jump instruction). The relative offset can be a signed 8-, 16-, or 32-bit value. Most efficient instruction encoding results if 8-bit offsets are used. With 8-bit offsets, the target should be within −128 to +127 of the first byte of the next instruction. For 16- and 32-bit offsets, the corresponding values are 2^{15} to $2^{15}-1$ and 2^{31} to $2^{31}-1$, respectively. When the target is in another segment, test for the opposite condition, and use the unconditional `jmp` instruction, as explained in Chapter 9. See Chapter 12 for details on the various conditions tested such as `ja`, `jbe`, and so on. The `jcxz` instruction tests the contents of the CX or ECX register and jumps to the target location only if (E)CX = 0. The default operand size determines whether CX or ECX is used for comparison. Clock cycles: 1 for all conditional jumps (except `jcxz`, which takes 5 or 6 cycles).

jmp — Unconditional jump

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
jmp     target
```

Description:

The `jmp` instruction alters program execution by transferring control unconditionally to the `target` location. This instruction allows jumps to another segment. In direct jumps, the `target` operand is a relative offset (relative to the instruction following the `jmp` instruction). The relative offset can be an 8-, 16-, or 32-bit value as in the conditional jump instruction. In addition, the relative offset can be specified indirectly via a register or memory location. See Chapter 12 for an example. For other forms of the `jmp` instruction, see the Pentium data book. Note: Flags are not affected unless there is a task switch, in which case all flags are affected. Clock cycles: 1 for direct jumps; 2 for indirect jumps (more clock cycles for other types of jumps).

lahf — Load flags into AH register

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
lahf
```

Description:

The `lahf` instruction loads the AH register with the low byte of the flags register. AH := SF, ZF, ∗, AF, ∗, PF, ∗, CF where ∗ represents indeterminate value. Clock cycles: 2.

lds/les/lfs/lgs/lss — Load full pointer

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
lds     dest,src
les     dest,src
lfs     dest,src
lgs     dest,src
lss     dest,src
```

Description:

These instructions read a full pointer from memory (given by the `src` operand) and loads corresponding segment register (e.g., DS register for the `lds` instruction, ES register for the `les` instruction, etc.) and the `dest` register. The `dest` operand must be a 16- or 32-bit register. The first two or four bytes (depending on whether the `dest` is a 16- or 32-bit register) at the effective address given by the `src` operand are loaded into the `dest` register and the next two bytes into the corresponding segment register. Clock cycles: 4 (except `lss`).

lea — Load effective address

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
lea     dest,src
```

Description:

The `lea` instruction computes the effective address of a memory operand given by `src` and stores it in the `dest` register. The `dest` must be either a 16- or 32-bit register. If the `dest` register is a 16-bit register and the address size is 32, only the lower 16 bits are stored. On the other hand, if a 32-bit register is specified when the address size is 16 bits, the effective address is zero-extended to 32 bits. Clock cycles: 1.

leave — Exit a procedure

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
leave
```

Description:

The `leave` instruction reverses the actions of the `enter` instruction. It copies (E)BP to (E)SP to release any stack space used for local variables. The old frame pointer is popped from the stack into the (E)BP register. More details on this instruction are in Chapter 10 (see page 408). Clock cycles: 3.

lods — Load string operand

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
lodsb
lodsw
lodsd
```

Description:

The `lods` instruction loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed by DS:SI or DS:ESI. The address size attribute determines whether the SI or ESI register is used. The `lodsw` and `loadsd` instructions share the same opcode (ADH). The operand size is used to load either a word or doubleword. After loading, the source index register is updated automatically. The index register is incremented if DF is 0; it is decremented if DF is 1. The index register is incremented or decremented by 1, 2, or 4 for byte, word, or doubleword operands, respectively. The `rep` prefix can be used with this instruction but is not useful as explained in Chapter 12. This instruction is typically used in a loop (see the `loop` instruction). Clock cycles: 2.

loop/loope/loopne — Loop control

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
loop    target
loope/loopz    target
loopne/loopnz    target
```

Description:

The `loop` instruction decrements the count register (CX if the address size attribute is 16 and ECX if it is 32) and jumps to `target` if the count register is not zero. This instruction decrements the (E)CX register without changing any flags. The operand `target` is a relative 8-bit offset (i.e., the target must be in the range −128 to +127 bytes).

The `loope` instruction is similar to `loop` except that it also checks the ZF value to jump to the `target`. That is, control is transferred to `target` if, after decrementing the (E)CX register, the count register is not zero and ZF = 1. `loopz` is a synonym for the `loope` instruction.

The `loopne` instruction is similar to `loope` except that it transfers control to `target` if ZF is 0 (instead of 1 as in the `loope` instruction). See Chapter 9 for more details on these instructions. Clock cycles: 5 or 6 for `loop` and 7 or 8 for the other two.

Note that the unconditional `loop` instruction takes longer to execute than a functionally equivalent two-instruction sequence that decrements the (E)CX register and jumps conditionally.

mov — Copy data

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
mov    dest,src
```

Description:

Copies data from `src` to `dest`. Clock cycles: 1 for most `mov` instructions except when copying into a segment register, which takes more clock cycles.

movs — Copy string data

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
movs    dest,src
movsb
movsw
movsd
```

Description:

Copies the byte, word, or doubleword pointed by the source index register (SI or ESI) to the byte, word, or doubleword pointed by the destination index register (DI or EDI). If the address size is 16 bits, SI and DI registers are used; ESI and EDI registers are used for 32-bit addresses. The default segment for the source is DS and ES for the destination. The segment override prefix can be used only for the source operand. After the move, both source and destination index registers are automatically updated as in the `cmps` instruction.

The `rep` prefix instruction can precede the `movs` instruction for block movement of data. See the `rep` instruction for details. Clock cycles: 4.

movsx — Copy with sign extension

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
movsx    reg16,src8
movsx    reg32,src8
movsx    reg32,src16
```

Description:

Copies the sign-extended source operand `src8/src16` into the destination `reg16/reg32`. The destination can be either a 16- or 32-bit register only. The source can be a register, memory byte, or word operand. Note that `reg16` and `reg32` represent a 16- and 32-bit register, respectively. Similarly, `src8` and `src16` represent a byte and word operand, respectively. Clock cycles: 3.

movzx — Copy with zero extension

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
movzx    reg16,src8
movzx    reg32,src8
movzx    reg32,src16
```

Description:

Similar to `movsx` instruction except `movzx` copies the zero-extended source operand into destination. Clock cycles: 3.

mul — Unsigned multiplication

C	O	Z	S	P	A
M	M	*	*	*	*

Format:

```
mul     AL,src8
mul     AX,src16
mul     EAX,src32
```

Description:

Performs unsigned multiplication of two 8-, 16-, or 32-bit operands. Only one of the operands need be specified; the other operand, matching in size, is assumed to be in the AL, AX, or EAX register.

- For 8-bit multiplication, the result is in the AX register. CF and OF are cleared if AH is zero; otherwise, they are set.
- For 16-bit multiplication, the result is in the DX:AX register pair. The higher-order 16 bits are in DX. CF and OF are cleared if DX is zero; otherwise, they are set.
- For 32-bit multiplication, the result is in the EDX:EAX register pair. The higher-order 32 bits are in EDX. CF and OF are cleared if EDX is zero; otherwise, they are set.

Clock cycles: 11 for 8- or 16-bit operands and 10 for 32-bit operands.

neg — Negate sign (2's complement)

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
neg     operand
```

Description:

Performs 2's complement negation (sign reversal) of the operand specified. The operand specified can be 8, 16, or 32 bits in size and can be located in a register or memory. The operand is subtracted from zero and the result is stored back in the operand. The CF flag is set for nonzero result; cleared otherwise. Other flags are set according to the result. Clock cycles: 1 for register operands and 3 for memory operands.

nop — No operation

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
nop
```

Description:

Performs no operation. Interestingly, the `nop` instruction is an alias for the `xchg (E)AX,(E)AX` instruction. Clock cycles: 1.

not — Logical bitwise not

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
not     operand
```

Description:

Performs 1's complement bitwise **not** operation (a 1 becomes 0 and vice versa). Clock cycles: 1 for register operands and 3 for memory operands.

or — Logical bitwise or

C	O	Z	S	P	A
0	0	M	M	M	*

Format:

```
or     dest,src
```

Description:

Performs bitwise **or** operation. The result (`dest` **or** `src`) is stored in `dest`. Clock cycles: 1 for register and immediate operands and 3 if a memory operand is involved.

out — Output to a port

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
out     port,src
out     DX,src
```

Description:

As does the `in` instruction, this instruction has two formats. In both formats, the `src` operand must be in the AL, AX, or EAX register. In the first format, it outputs a byte, word, or doubleword from `src` to the I/O port specified by the first operand `port`. Note that `port` is an 8-bit immediate value. This format limits access to the first 256 I/O ports in the I/O space. The other format is more general and allows access to the full I/O space (i.e., any port between 0 and 65,535). In this format, the port number is assumed to be in the DX register. Clock cycles: varies; see Pentium data book.

outs — Output from a string to a port

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
outsb
outsw
outsd
```

Description:

This instruction transfers 8-, 16-, or 32-bit data from a string (pointed by the source index register) to the output port specified in the DX register. Similar to the `ins` instruction, it uses the SI index register for 16-bit addresses and the ESI register if the address size is 32. The (E)SI register is automatically updated after the transfer of a data item. The index register is incremented if DF is 0; it is decremented if DF is 1. The index register is incremented or decremented by 1, 2, or 4 for byte, word, or doubleword operands, respectively. The repeat prefix can be used with `outs` for block transfer of data. Clock cycles: varies; see Pentium data book.

pop — Pop a word from the stack

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
pop     dest
```

Description:

Pops a word or doubleword from the top of the stack. If the address size attribute is 16 bits, SS:SP is used as the top of the stack pointer; otherwise, SS:ESP is used. `dest` can be a register or memory operand. In addition, it can also be a segment register DS, ES, SS, FS, or GS (e.g., `pop DS`). The stack pointer is incremented by 2 (if the operand size is 16 bits) or 4 (if the operand size is 32 bits). Note that `pop CS` is not allowed. This can be done only indirectly by the `ret` instruction. Clock cycles: 1 if `dest` is a general register; 3 if `dest` is a segment register or memory operand.

popa — Pop all general registers

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
popa
popad
```

Description:

Pops all eight 16-bit (`popa`) or 32-bit (`popad`) general registers from the top of the stack. The `popa` loads the registers in the order DI, SI, and BP, discards the next two bytes (to skip loading into SP), then BX, DX, CX, and AX. That is, DI is popped first and AX last. The `popad` instruction follows the same order on the 32-bit registers. Clock cycles: 5.

popf — Pop flags register

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
popf
popfd
```

Description:

Pops the 16-bit (`popf`) or 32-bit (`popfd`) flags register (FLAGS or EFLAGS) from the top of the stack. Bits 16 (VM flag) and 17 (RF flag) of the EFLAGS register are not affected by this instruction. Clock cycles: 6 in the real mode and 4 in the protected mode.

push — Push a word onto the stack

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
push    src
```

Description:

Pushes a word or doubleword onto the top of the stack. If the address size attribute is 16 bits, SS:SP is used as the top of the stack pointer; otherwise, SS:ESP is used. `src` can be (i) a register, (ii) a memory operand, (iii) a segment register (CS, SS, DS, ES, FS, or GS), or (iv) an immediate byte, word, or doubleword operand. The stack pointer is decremented by 2 (if the operand size is 16 bits) or 4 (if the operand size is 32 bits). The `push ESP` instruction pushes the ESP register value before it is decremented by the `push` instruction. On the other hand, `push SP` pushes the decremented SP value onto the stack. Clock cycles: 1 (except when the operand is in memory, in which case it takes 2 clock cycles).

pusha — Push all general registers

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
pusha
pushad
```

Description:

Pushes all eight 16-bit (`pusha`) or 32-bit (`pushad`) general registers onto the stack. The `pusha` pushes the registers onto the stack in the order AX, CX, DX, BX, SP, BP, SI, and DI. That is, AX is pushed first and DI last. The `pushad` instruction follows the same order on the 32-bit registers. It decrements the stack pointer SP by 16 for word operands and decrements ESP by 32 for doubleword operands. Clock cycles: 5.

pushf — Push flags register

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
pushf
pushfd
```

Description:

Pushes the 16-bit (`pushf`) or 32-bit (`pushfd`) flags register (FLAGS or EFLAGS) onto the stack. Decrements SP by 2 (`pushf`) for word operands and decrements ESP by 4 (`pushfd`) for doubleword operands. Clock cycles: 4 in the real mode and 3 in the protected mode.

rep/repe/repz/repne/repnz — Repeat instruction

C	O	Z	S	P	A
–	–	M	–	–	–

Format:

```
rep     string-inst
repe/repz     string-inst
repne/repnz     string-inst
```

Description:

These three prefixes repeat the specified string instruction until the conditions are met. The `rep` instruction decrements the count register (CX or ECX) each time the string instruction is executed. The string instruction is repeatedly executed until the count register is zero. The `repe` (repeat while equal) has an additional termination condition: ZF = 0. The `repz` is an alias for the `repe` instruction. The `repne` (repeat while not equal) is similar to `repe` except that the additional termination condition is ZF =1. The `repnz` is an alias for the `repne` instruction. The ZF flag is affected by `rep cmps` and `rep scas` instructions. For more details, see Chapter 12. Clock cycles: varies; see Pentium data book for details.

ret — Return from a procedure

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
ret
ret value
```

Description:

Transfers control to the instruction following the corresponding `call` instruction. The optional immediate `value` specifies the number of bytes (for 16-bit operands) or number of words (for 32-bit operands) that are to be cleared from the stack after the return. This parameter is usually used to clear the stack of the input parameters. See Chapter 10 for more details. Clock cycles: 2 for near return and 3 for far return; if the optional `value` is specified, add one more clock cycle. Changing privilege levels takes more clocks; see Pentium data book.

rol/ror/rcl/rcr — Rotate instructions

C	O	Z	S	P	A
M	M	–	–	–	–

Format:

```
rol/ror/rcl/rcr    src,1
rol/ror/rcl/rcr    src,count
rol/ror/rcl/rcr    src,CL
```

Description:

This group of instructions supports rotation of 8-, 16-, or 32-bit data. The `rol` (rotate left) and `ror` (rotate right) instructions rotate the `src` data as explained in Chapter 9. The second operand gives the number of times `src` is to be rotated. This operand can be given as an immediate value (a constant 1 or a byte value `count`) or preloaded into the CL register. The other two rotate instructions `rcl` (rotate left including CF) and `rcr` (rotate right including CF) rotate the `src` data with the carry flag (CF) included in the rotation process, as explained in Chapter 9. The OF flag is affected only for single bit rotates; it is undefined for multibit rotates. Clock cycles: `rol` and `ror` take 1 (if `src` is a register) or 3 (if `src` is a memory operand) for the immediate mode (constant 1 or `count`) and 4 for the CL version; for the other two instructions, it can take as many as 27 clock cycles; see Pentium data book for details.

sahf — Store AH in flags register

C	O	Z	S	P	A
M	–	M	M	M	M

Format:

```
sahf
```

Description:

The AH register bits 7, 6, 4, 2, and 0 are loaded into flags SF, ZF, AF, PF, and CF, respectively. Clock cycles: 2.

sal/sar/shl/shr — Shift instructions

C	O	Z	S	P	A
M	M	M	M	M	–

Format:

```
sal/sar/shl/shr    src,1
sal/sar/shl/shr    src,count
sal/sar/shl/shr    src,CL
```

Description:

This group of instructions supports shifting of 8-, 16-, or 32-bit data. The format is similar to the rotate instructions. The `sal` (shift arithmetic left) and its synonym `shl` (shift left) instructions shift the `src` data left. The shifted out bit goes into CF and the vacated bit is cleared, as explained in Chapter 9. The second operand gives the number of times `src` is to be shifted. This operand can be given as an immediate value (a constant 1 or a byte value `count`) or preloaded into the CL register. The `shr` (shift right) is similar to `shl` except for the direction of shift. The `sar` (shift arithmetic right) is similar to `sal` except for two differences: the shift direction is right and the sign bit is copied into the vacated bits. If the shift count is zero, no flags are affected. The CF flag contains the last bit shifted out. The OF flag is defined only for single shifts; it is undefined for multibit shifts. Clock cycles: 1 (if `src` is a register) or 3 (if `src` is a memory operand) for the immediate mode (constant 1 or `count`) and 4 for the CL version.

sbb — Subtract with borrow

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
sbb     dest,src
```

Description:

Performs integer subtraction with borrow. The `dest` is assigned the result of `dest - (src+CF)`. Clock cycles: 1–3.

scas — Compare string operands

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
scas    operand
scasb
scasw
scasd
```

Description:

Subtracts the memory byte, word, or doubleword pointed by the destination index register (DI or EDI) from the AL, AX, or EAX register, respectively. The result is not stored but used to update the flags. The memory operand must be addressable from the ES register. Segment override is not allowed in this instruction. If the address size is 16 bits, the DI register is used; the EDI register is used for 32-bit addresses. After the subtraction, the destination index register is updated automatically. Whether the register is incremented or decremented depends on the direction flag (DF). The register is incremented if DF is 0 (see the `cld` instruction to clear the direction flag); if the DF is 1, the index register is decremented (see the `std` instruction to set the direction flag). The amount of increment or decrement is 1 (for byte operands), 2 (for word operands), or 4 (for doubleword operands).

Note that the specification of the operand in `scas` is not really required as the memory operand is assumed to be pointed by the index register. The `scasb`, `scasw`, and `scasd` are synonyms for the byte, word, and doubleword `scas` instructions, respectively.

The repeat prefix instructions (i.e., `repe` or `repne`) can precede the `scas` instructions for array or string comparisons. See the `rep` instruction for details. Clock cycles: 4.

setCC — Byte set on condition operands

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
setCC    dest
```

Description:

Sets `dest` byte to 1 if the condition CC is met; otherwise, sets to zero. The operand `dest` must be either an 8-bit register or memory operand. The conditions tested are similar to the conditional jump instruction (see `jcc` instruction). The conditions are: A, AE, B, BE, E, NE, G, GE, L, LE, NA, NAE, NB, NBE, NG, NGE, NL, NLE, C, NC, O, NO, P, PE, PO, NP, O, NO, S, NS, Z, NZ. The conditions can specify signed and unsigned comparisons as well as flag values. Clock cycles: 1 for register operand and 2 for memory operand.

shld/shrd — Double precision shift

C	O	Z	S	P	A
M	M	M	M	M	*

Format:

```
shld/shrd    dest,src,count
```

Description:

The `shld` instruction performs left-shift of `dest` by `count` times. The second operand `src` provides the bits to shift in from the right. In other words, the `shld` instruction performs a left-shift of `dest` concatenated with `src` and the result in the upper half is copied into `dest`. `dest` and `src` operands can both be either 16- or 32-bit operands. Although `dest` can be a register or memory operand, `src` must be a register of the same size as `dest`. The third operand `count` can be an immediate byte value or the CL register can be used as in the shift instructions. The contents of the `src` register are not altered.

The `shrd` instruction (double precision shift right) is similar to `shld` except for the direction of shift.

If shift count is zero, no flags are affected. The CF flag contains the last bit shifted out. The OF flag is defined only for single shifts; it is undefined for multibit shifts. The SF, ZF, and PF flags are set according to the result.

Clock cycles: 4 (5 if `dest` is a memory operand and the CL register is used for `count`).

stc — Set carry flag

C	O	Z	S	P	A
1	–	–	–	–	–

Format:

```
stc
```

Description:

Sets the carry flag to 1. Clock cycles: 2.

std — Set direction flag

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
std
```

Description:

Sets the direction flag to 1. Clock cycles: 2.

sti — Set interrupt flag

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
sti
```

Description:

Sets the interrupt flag to 1. Clock cycles: 7.

stos — Store string operand

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
stosb
stosw
stosd
```

Description:

Stores the contents of the AL, AX, or EAX register at the memory byte, word, or doubleword pointed by the destination index register (DI or EDI), respectively. If the address size is 16 bits, the DI register is used; the EDI register is used for 32-bit addresses. After the load, the destination index register is automatically updated. Whether this register is incremented or decremented depends on the direction flag (DF). The register is incremented if DF is 0 (see the `cld` instruction to clear the direction flag); if the DF is 1, the index register is decremented (see the `std` instruction to set the direction flag). The amount of increment or decrement depends on the operand size (1 for byte operands, 2 for word operands, and 4 for doubleword operands).

The repeat prefix instruction `rep` can precede the `stos` instruction to fill a block of CX/ECX bytes, words, or doublewords. Clock cycles: 3.

sub — Subtract

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
sub     dest,src
```

Description:

Performs integer subtraction. The `dest` is assigned the result of `dest − src`. Clock cycles: 1 to 3.

test — Logical compare

C	O	Z	S	P	A
0	0	M	M	M	*

Format:

```
test    dest,src
```

Description:

Performs logical **and** operation (`dest` **and** `src`). However, the result of the **and** operation is discarded. The `dest` operand can be either in a register or memory. The `src` operand can be either an immediate value or a register. Both `dest` and `src` operands are not affected. Sets SF, ZF, and PF flags according to the result. Clock cycles: 1 if `dest` is a register operand and 2 if it is a memory operand.

xchg — Exchange data

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
xchg    dest,src
```

Description:

Exchanges the values of the two operands `src` and `dest`. Clock cycles: 2 if both operands are registers or 3 if one of them is a memory operand.

xlat — Translate byte

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
xlat    table-offset
xlatb
```

Description:

Translates the data in the AL register using a table lookup. It changes the AL register from the table index to the corresponding table contents. The contents of the BX (for 16-bit addresses) or EBX (for 32-bit addresses) register are used as the offset to the translation table base. The contents of the AL register are treated as an index into this table. The byte value at this index replaces the index value in AL. The default segment for the translation table is DS. This is used in both formats. However, in the operand version, a segment override is possible. Clock cycles: 4.

xor — Logical bitwise exclusive-or

C	O	Z	S	P	A
0	0	M	M	M	*

Format:

```
xor     dest,src
```

Description:

Performs logical bitwise exclusive-or (xor) operation (`dest` **xor** `src`) and the result is stored in `dest`. Sets the SF, ZF, and PF flags according to the result. Clock cycles: 1 to 3.

Bibliography

[1] D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro,* June 1993, pp. 11–21.

[2] D. Anderson, *PCMCIA System Architecture: 16-Bit Cards,* Second edition, Addison-Wesley, Reading, MA, 1995.

[3] D. Anderson, *Universal Serial Bus System Architecture,* Addison-Wesley, Reading, MA, 1997.

[4] D. Anderson, *PCI System Architecture,* Fourth edition, Addison-Wesley, Reading, MA, 1999.

[5] D. Anderson, *FireWire System Architecture,* Second edition, Addison-Wesley, Reading, MA, 1999.

[6] D. Anderson and T. Shanley, *ISA System Architecture,* Third edition, Addison-Wesley, Reading, MA, 1995.

[7] D. Anderson and T. Shanley, *CardBus System Architecture,* Addison-Wesley, Reading, MA, 1996.

[8] C.G. Bell, "The Mini and Micro Industries," *IEEE Computer,* Vol. 17, No. 10, October 1984, pp. 14–30.

[9] M. Campbell-Kelly and W. Aspray, *Computer: A History of the Information Machine,* Basic Books, New York, 1996.

[10] Compaq, "PCI-X Architectural Overview," 2000. Available from `www.compaq.com/products/servers/technology/pci-x-enablement.html`.

[11] S.P. Dandamudi, *Introduction to Assembly Language Programming,* Springer-Verlag, New York, 1998.

[12] D. Dzatko and T. Shanley, *AGP System Architecture,* Second edition, Addison-Wesley, Reading, MA, 2000.

[13] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys,* Vol. 23, No. 1, March 1991, pp. 5–48.

[14] H. Goldstein, *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ, 1972.

[15] T. Graham, *Unicode: A Primer,* M&T Books, New York, 2000.

[16] V.C. Hamacher, Z.G. Vranesic, and S.G. Zaky, *Computer Organization* (Fourth edition), McGraw-Hill, New York, 1996.

[17] J. Handy, *The Cache Memory Book*, Academic Press, San Diego, CA, 1998.

[18] J.L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, July 2000, pp. 28–35.

[19] R.N. Ibbett and N.P. Topham, *Architecture of High Performance Computers* (Volume 1), Springer-Verlag, New York, 1989.

[20] Intel, *Accelerated Graphics Port Interface Specification,* Revision 2.0, May 1998. (Available from `developer.intel.com/technology/agp/agp_index.htm.`)

[21] Intel, *Itanium Architecture Software Developer's Manual*, 2001. This four-volume set is available from `developer.intel.com/design/ia-64`.

[22] B. Jacob and T. Mudge, "Virtual Memory: Issues of Implementation," *Computer*, June 1998, pp. 33–43.

[23] B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors," *IEEE Micro*, July–August 1998, pp. 60–75.

[24] J.R. Larus, *SPIM S20: A MIPS R2000 Simulator,* 1997. Available from `http://www.cs.wisc.edu/~larus/SPIM_manual/spim-manual.html`.

[25] J.K. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, Vol. 17, No. 1, 1984, pp. 6–22.

[26] Motorola, *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Processors,* 1997. Available from `http://www.motorola.com/SPS/PowerPC`.

[27] *NASM Manual,* `http://www.octium.net/nasm`.

[28] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* Second edition, Morgan Kaufmann, San Francisco, CA, 1998.

[29] D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer,* Vol. 15, No. 9, 1982, pp. 8–21.

[30] K.A. Robbins and S. Robbins, *The Cray X-MP/Model 24: A Case Study in Pipelined Architecture and Vector Processing,* Lecture Notes in Computer Science 375, Springer-Verlag, New York, 1987.

[31] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures: A Design Space Approach,* Addison-Wesley, New York, 1997.

[32] A.J. Smith, "Cache Memories," *ACM Computing Surveys,* Vol. 14, 1982, pp. 473–530.

[33] W. Stallings, *Computer Organization and Architecture*, Fifth edition, Prentice-Hall, Englewood Cliffs, NJ, 2000.

[34] Standard Performance Evaluation Corporation, `http://www.spec.org`.

[35] A.S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Communications of the ACM,* Vol. 21, No. 3, 1978, pp. 237–246.

[36] A.S. Tanenbaum, *Modern Operating Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[37] A.S. Tanenbaum, *Structured Computer Organization,* Fourth edition, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[38] Unicode Consortium, *The Unicode Standard: A Technical Introduction*. Available from `http://www.unicode.org/unicode/standard/principles.html`.

[39] M.V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," *Proceedings of the Manchester University Computer Inaugural Conference,* 1951.

[40] M.V. Wilkes and J.B. Stinger, "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer," *Proceedings of the Cambridge Philosophical Society,* 1953, pp. 230–238.

Index