TEXTS IN COMPUTER SCIENCE

Editors
David Gries
Fred B. Schneider

Sivarama P. Dandamudi

Introduction to Assembly Language Programming

For Pentium and RISC Processors

With 75 Illustrations

Springer

Sivarama P. Dandamudi
School of Computer Science
Carleton University
1125 Colonel By Drive
Ottawa, K1S 5B6
Canada
sivarama@scs.carleton.ca

Series Editors:
David Gries
Fred B. Schneider
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Library of Congress Cataloging-in-Publication Data
Dandamudi, Sivarama P., 1955–
Introduction to Assembly language programming : Pentium and RISC processors / Sivarama P. Dandamudi.— 2nd ed.
p. cm. — (Texts in computer science)
Includes bibliographical references and index.
ISBN 0-387-20636-1 (alk. paper)
1. Assembler language (Computer program language) 2. Microprocessors—Programming. I. Title. II. Series.
QA76.73.A87D36 2004
005.13′6—dc22 2004049182

ISBN 0-387-20636-1 Printed on acid-free paper.

Printed in the United States of America. (HAM)

9 8 7 6 5 4 3 2 1 SPIN 10949580

springeronline.com

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

Preface

The objective of this book is to introduce assembly language programming. Assembly language is very closely linked to the underlying processor architecture and design. Popular processor designs can be broadly divided into two categories: Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC). The dominant processor in the PC market, Pentium, belongs to the CISC category. However, the recent design trend is to use the RISC designs. Some example RISC processors include the MIPS, SPARC, PowerPC, and ARM. Even Intel's 64-bit processor Itanium is a RISC processor. Thus, both types of processors are important candidates for our study.

This book covers assembly language programming of both CISC and RISC processors. We use the Intel Pentium processor as the representative of the CISC category. We have selected the Pentium processor because of its market dominance. To explore RISC assembly language, we selected the MIPS processor. The MIPS processor is appealing as it closely adheres to the RISC principles. Furthermore, the availability of the SPIM simulator allows us to use a Pentium-based PC to learn MIPS assembly language.

New in the Second Edition

The second edition has been substantially revised to reflect the changes that have taken place since the publication of the first edition. The major changes are listed below:

- We introduced RISC assembly language programming so that the reader can benefit from learning both CISC and RISC assembly languages. As mentioned before, Pentium and MIPS processors are used to cover CISC and RISC processors.

- The first edition used MASM/TASM assemblers. In this edition, we use the NASM assembler. The syntax of NASM is slightly different from that of MASM/TASM assemblers. The advantage is that NASM is free! Another advantage is that it works with both Microsoft Windows and Linux operating systems.
- Consistent with our shift to NASM, we moved away from DOS to Linux. Since NASM is available for Windows and Linux, most of the programs in this book can be used with either Windows or Linux. However, we clearly indicate our preference to Linux. This preference is exposed in chapters like "High-Level Language Interface" that deal with mixed-mode programming involving C and assembly language. For example, in Chapter 17, we use the GNU C compiler (`gcc`) rather than the Microsoft or Borland C compiler. Similarly, in Appendix C we use the GNU debugger (`gdb`) to explore the debugging process.
- The "Basic Computer Organization" chapter (Chapter 2) has been completely rewritten to give a general background on computer organization. The Pentium processor details are moved to a new chapter (Chapter 4).
- A completely new chapter has been added to discuss Pentium's protected mode interrupt processing.
- We have added a new chapter on recursion. This chapter discusses how we can implement recursive procedures in the Pentium and MIPS assembly languages.
- We have augmented the Pentium assembly language programming by describing its floating-point instructions. This entire chapter is new in this edition.

In addition to these major changes, all chapters have gone through extensive revision. Some chapters have been reorganized to eliminate the duplication present in the first edition.

Intended Use

Assembly language programming is part of several undergraduate curricula in computer science, computer engineering, and electrical engineering departments. This book can be used as a text for those courses that teach assembly language.

It can also be used as a companion text in a computer organization course for teaching the assembly language. Because we cover both CISC and RISC processors, the instructor can select the assembly language that best fits her or his course.

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.

Instructional Support

The book's Web site (`www.scs.carleton.ca/~sivarama/asm_book`) has complete chapter-by-chapter PowerPoint slides for instructors. Instructors can use these slides directly in their classes or can modify them to suit their needs. In addition, instructors can obtain the solutions manual by contacting the publisher. For more up-to-date details, please see the book's Web site.

Prerequisites

The student is assumed to have had some experience in a structured, high-level language such as C. However, the book does not assume extensive knowledge of any high-level language—only the basics are needed. Furthermore, it is assumed that the student has a rudimentary background in the software development cycle, as is obtained in a typical high-level programming course.

Features

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

- This is probably the only book to cover the assembly language programming of both CISC and RISC processors.
- This book uses NASM and Linux as opposed to scores of other books that use MASM and Windows.
- The book is self-contained and does not assume a background in computer organization. All necessary background material on computer organization is presented in the book.
- This book contains a methodical organization of chapters for a step-by-step introduction to the assembly language.
- Extensive examples are used in each chapter to illustrate the points discussed in the chapter. Our objective is not just to explain how an instruction works but also to provide the rationale as to why the instruction has been designed the way it is.
- Procedures are introduced early on to encourage modular programming in developing assembly language programs.
- A set of input and output routines is provided so that the student can focus on developing assembly language programs rather than spending time in understanding how the input and output are done using the basic I/O functions provided by the operating system.
- This book does not use fragments of code in examples. All examples are complete in the sense that they can be assembled and run, giving a better feeling as to how these programs work.
- All examples and other required software are available from the book's Web site (`www.scs.carleton.ca/~sivarama/asm_book`) 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 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 at the end.
- Each chapter contains two types of exercises—review and programming—to reinforce the concepts discussed in the chapter.
- The appendices provide special reference material that contains a thorough treatment of various topics.

Overview and Organization

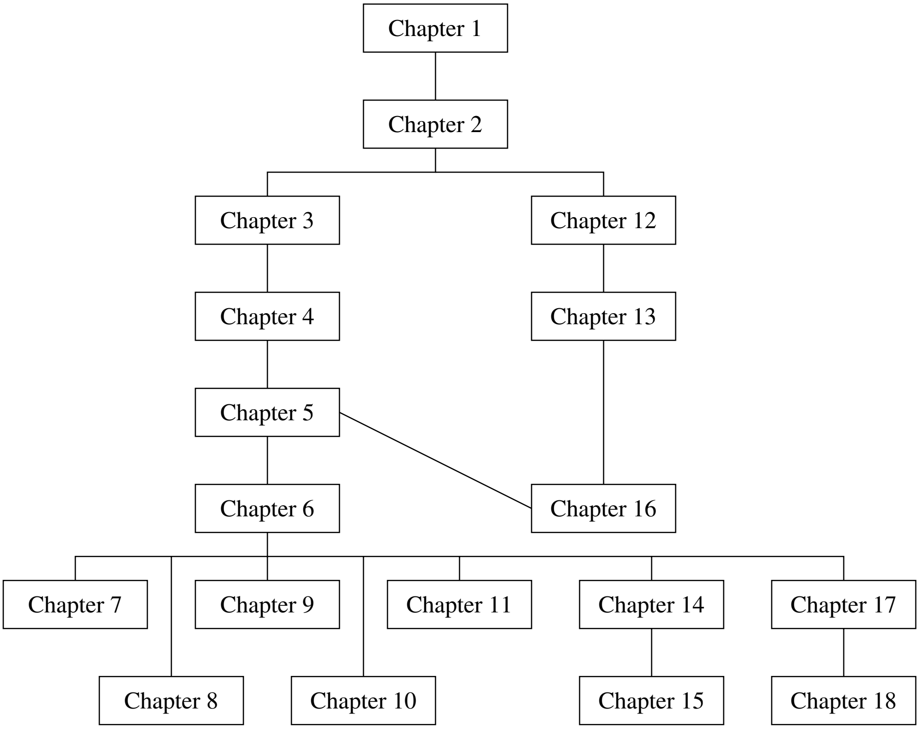
The 18 chapters in the book are divided into 6 parts (see the figure on the next page for chapter dependencies). Part I presents introductory topics and consists of the first two chapters. Chapter 1 provides introduction to the assembly language and gives reasons for programming in the assembly language. Chapter 2 presents the basics of computer organization with a focus on three system components: processor, memory, and I/O.

Part II is dedicated to Pentium assembly language programming. It consists of nine chapters—Chapters 3 through 11. This part begins with a description of the Pentium processor organization (Chapter 3). In particular, this chapter gives sufficient details on the 16- and 32-bit Intel processors so that the student can effectively program in the assembly language. Chapter 4 gives an overview of the assembly language. After covering these two chapters, one can write simple standalone assembly language programs.

To emphasize the importance of modular programming, procedures are introduced early on (in Chapter 5). The other chapters in this part expand on the overview given in Chapter 4. Chapter 6 presents the addressing modes supported by the Intel 16- and 32-bit processors. This chapter also contains a detailed discussion on the motivation for providing the various addressing modes. Addressing modes are one of the differentiating characteristics of CISC processors. Chapter 7 discusses the arithmetic instructions and the use of the flags register. Chapters 8 and 9 present conditional and bit manipulation instructions. A feature of these two chapters is that they relate how high-level language statements can be implemented using the instructions discussed in these two chapters. Chapter 10 discusses the string processing instructions in detail. ASCII and BCD arithmetic instructions are presented in Chapter 11.

The first four chapters of this part—Chapters 3 to 6—should be covered in some detail for proper grounding in assembly language programming. However, the remaining five chapters can be studied in any order. In addition, the depth at which these five chapters are covered can be varied without sacrificing the effectiveness, depending on the time available and importance to the course objective.

Part III is dedicated to the MIPS assembly language programming. Chapter 12 describes the RISC design principles; it also covers MIPS processor details. The MIPS assembly language is presented in Chapter 13. This chapter also gives details on the SPIM simulator. All



the programming examples given in this chapter can be run on a Pentium-based PC using the SPIM simulator. The SPIM simulator details are given in Appendix D.

Part IV focuses on Pentium's interrupt processing mechanism. We cover both protected-mode and real-mode interrupt processing. Chapter 14 gives details on protected-mode interrupt processing. This chapter uses Linux system calls to facilitate our discussion of software interrupts. The next chapter discusses the real-mode interrupt processing. This is the only chapter that uses DOS to explore how programmed I/O and interrupt-driven I/O are done.

The remaining 3 of the 18 chapters constitute Part V. These chapters deal with advanced topics. Chapter 16 focuses on how recursive procedures are implemented in Pentium and MIPS assembly languages. The next chapter deals with the high-level language interface, which allows mixed-mode programming. We use C and assembly language to cover the principles involved in mixed-mode programming. The last chapter discusses Pentium's floating-point instructions. To follow the programming examples of this chapter, you need to understand the high-level language interface details presented in Chapter 17.

The seven appendices provide a wealth of reference material the student needs. Appendix A primarily discusses the number systems and their internal representation. Appendix B gives information on the use of I/O routines provided with this book and the assembler software. The debugging aspect of assembly language programming is discussed in Appendix C. The SPIM simulator details are given in Appendix D. Selected Pentium and MIPS instructions are given in Appendices E and F, respectively. Finally, Appendix G gives the standard ASCII table.

Acknowledgments

Several people have contributed, either directly or indirectly, in writing this book. First and foremost, I would like to thank my family for enduring my preoccupation with this project. My heartfelt thanks to Sobha and Veda for their understanding and patience!

I want to thank Ann Kostant, Executive Editor at Springer, for her positive feedback on the proposal for the revision. A very special thanks to Wayne Wheeler, Associate Editor, for handling various aspects of the project in a timely manner. I would also like to express my appreciation to the staff at the Springer production department for converting my camera-ready copy into the book in front of you.

I also express my appreciation to the School of Computer Science at Carleton University for providing a great atmosphere to complete this book.

Feedback

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

Ottawa, Canada
January 2004

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

Contents

PART I

Overview

This part consists of two chapters. The first chapter gives an introduction to assembly language along with reasons for programming in assembly language. This chapter also informally introduces the two main processor designs: CISC and RISC.

The second chapter presents the basics of computer organization with a focus on three system components: processor, memory, and I/O. This chapter also explains why data alignment improves performance.

These two chapters set the stage for our explorations of assembly languages of CISC and RISC processors in the remainder of the book.

Chapter 1

Introduction

Objectives

- To introduce the assembly language and explain where it fits in the hierarchy of computer languages
- To discuss the advantages and disadvantages associated with programming in the assembly language
- To provide motivation to learn the assembly language
- To demonstrate performance advantages of the assembly language

Users of a computer system can interact with the system at several different levels. At the highest level, the interaction could be through an application program such as a word processor. The next two levels use a programming language to facilitate interaction at a lower level. The hierarchy of levels is discussed in Section 1.1.

High-level programming languages such as C and Java can be used to develop modular programs. These languages provide several high-level constructs such as `if-then-else` *and* `while` *that facilitate program development and maintenance. After giving a brief introduction to the assembly language in Section 1.2, we elaborate on the main advantages of high-level languages in Section 1.3. The need for programming in the assembly language is discussed in Section 1.4. Section 1.5 identifies some typical application areas that benefit from programming in the assembly language. Section 1.6 discusses some reasons for learning the assembly language. The performance advantage of the assembly language over C is demonstrated in Section 1.7. A summary of the chapter is given in the last section.*

1.1 A User's View of Computer Systems

A user's view of a computer system depends on the degree of abstraction provided by the underlying software. Figure 1.1 shows a hierarchy of levels at which users can interact with a computer system. Moving to the top of the hierarchy shields the user from the lower-level details. At the highest level, the user interaction is limited to the interface provided by application software such as a spreadsheet, word processor, and so on. The user is expected to have only a rudimentary knowledge of how the system operates. Problem solving at this level, for example, involves composing a letter using the word processor software.

At the next level, problem solving is done in one of the *high-level languages* such as C and Java. A user interacting with the system at this level should have detailed knowledge of software development. Typically, these users are application programmers. Level 4 users are knowledgeable about the application and the high-level language that they would use to write the application software. They may not, however, know internal details of the system unless they also happen to be involved in developing system software such as device drivers, assemblers, linkers, and so on.

Both levels 4 and 5 are *system-independent*, i.e., independent of a particular processor used in the system. For example, an application program written in C can be executed on a system with an Intel processor or a PowerPC processor without modifying the source code. All we have to do is recompile the program with a C compiler native to the target system. By contrast, software development done at all levels below level 4 is *system-dependent*.

Assembly language programming is referred to as *low-level programming* because each assembly language instruction performs a much lower-level task compared to an instruction in a high-level language. As a consequence, to perform the same task, assembly language code tends to be much larger than the equivalent high-level language code. Assembly language instructions are native to the processor used in the system. For example, a program written in the Pentium assembly language cannot be executed on the PowerPC processor. Programming in the assembly language also requires knowledge about system internal details such as the processor architecture, memory organization, and so on.

Machine language is a close relative of the assembly language. Typically, there is a one-to-one correspondence between the assembly language and machine language instructions. The processor understands only the machine language, whose instructions consist of strings of 1's and 0's. We say more on these two languages in the next section.

Even though the assembly language is considered a low-level language, programming in the assembly language will not expose you to all the nuts and bolts of the system. Our operating system hides several of the low-level details so that the assembly language programmer can breathe easily. For example, if we want to read input from the keyboard, we can rely on the services provided by the operating system.

Well, ultimately there has to be something to execute the machine language instructions. This is the system hardware, which consists of digital logic circuits and the associated support electronics. A detailed discussion of this topic is beyond the scope of this book. Books on computer organization discuss this topic in detail.

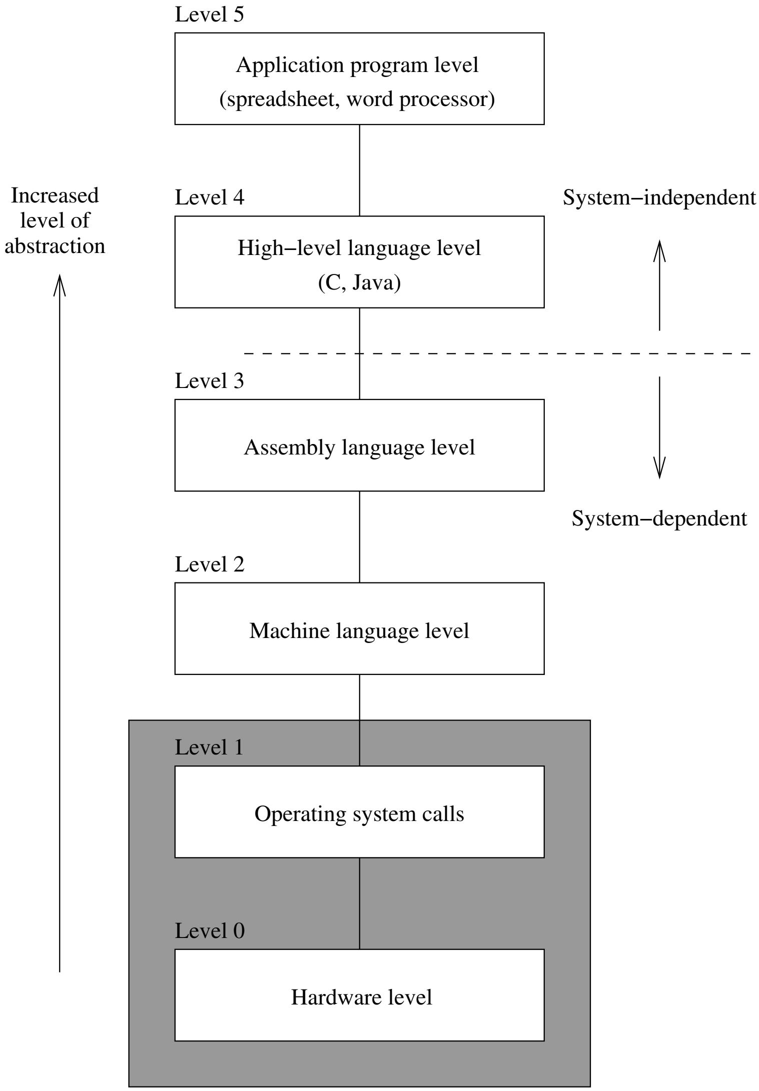


Figure 1.1 A user's view of a computer system.

1.2 What Is Assembly Language?

Assembly language is directly influenced by the instruction set and architecture of the processor. There are two basic types of processors: CISC (Complex Instruction Set Computers) and RISC (Reduced Instruction Set Computers). The Pentium is an example of a CISC processor.

Most current processors, however, follow the RISC design philosophy. In this book, we use the MIPS processor to represent the RISC category.

As the name suggests, CISC processors 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. Furthermore, RISC systems assume that the required operands are in the processor's registers, not in the main memory. We discuss processor registers in the next chapter. For now, think of them as a scratchpad inside the processor.

A CISC processor, on the other hand, 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 12.

The assembly language code must be processed by a program in order to generate the machine language code. *Assembler* is the program that translates assembly language code into the machine language. NASM (Netwide Assembler), MASM (Microsoft Assembler), and TASM (Borland Turbo Assembler) are some of the popular assemblers for the Pentium processors. In this book, we focus on the NASM assembler. In addition, we use the SPIM simulator to run the MIPS assembly language programs.

Here are some Pentium language examples:

```
inc   result
mov   class_size,45
and   mask1,128
add   marks,10
```

The first instruction increments the variable `result`. This assembly language instruction is equivalent to

```
result++;
```

in C. The second instruction initializes `class_size` to 45. The equivalent statement in C is

```
class_size = 45;
```

The third instruction performs the bitwise `and` operation on `mask1` and can be expressed in C as

```
mask1 = mask1 & 128;
```

The last instruction updates `marks` by adding 10. This is equivalent to

```
marks = marks + 10;
```

in C.

As you can see from these examples, most Pentium instructions use two addresses. In these instructions, one operand doubles as a source and destination (for example, marks and class_size). In contrast, the MIPS instructions use three addresses as shown below:

```
andi   $t2,$t1,15
addu   $t3,$t1,$t2
move   $t2,$t1
```

The operands in these instructions are in processor registers. The processor registers t1, t2, and t3 are identified by $. The andi instruction performs bitwise and of t1 contents with 15 and writes the result in t2. The second instruction adds contents of t1 and t2 and stores the result in t3.

The last instruction copies the t1 value into t2. In contrast to our claim that MIPS uses three addresses, this instruction seems to use only two addresses. This is not really an instruction supported by MIPS processor—it is a synthesized assembly language instruction. When translated by the MIPS assembler, it is replaced by

```
addu    $t2,$0,$t1
```

The second operand in this instruction is a special register that holds constant zero. Thus, copying the t1 value is treated as adding zero to it.

These examples illustrate several points:

1. Assembly language instructions are cryptic.
2. Assembly language operations are expressed by using mnemonics (like and, inc, addu, and so on).
3. Assembly language instructions are low-level. For example, we cannot write the following in the Pentium assembly language:

   ```
   add    marks,value
   ```

   This instruction is invalid because two variables, marks and value, cannot be used in a single instruction. In MIPS, for example, we cannot even write something like

   ```
   addu   class_size,45
   ```

   as it expects all operands in the processor's internal registers.

We appreciate the readability of the assembly language instructions by looking at the equivalent machine language instructions. Here are some Pentium and MIPS machine language examples:

Pentium examples

Assembly language		Operation	Machine language (in hex)
`nop`		No operation	`90`
`inc`	`result`	Increment	`FF060A00`
`mov`	`class_size, 45`	Copy	`C7060C002D00`
`and`	`mask, 128`	Logical and	`80260E0080`
`add`	`marks, 10`	Integer addition	`83060F000A`

MIPS examples

Assembly language		Operation	Machine language (in hex)
`nop`		No operation	`00000000`
`move`	`$t2,$t15`	Copy	`000A2021`
`andi`	`$t2,$t1,15`	Logical and	`312A000F`
`addu`	`$t3,$t1,$t2`	Integer addition	`012A5821`

In the above tables, machine language instructions are written in the hexadecimal number system. If you are not familiar with this number system, consult Appendix A for a detailed discussion of various number systems. These examples visibly demonstrate one of the key differences between CISC and RISC processors: RISC processors use fixed-length machine language instructions whereas the machine language instructions of CISC processors vary in length.

It is obvious from these examples that understanding the code of a program in the machine language is almost impossible. Since there is a one-to-one correspondence between the instructions of assembly language and machine language, it is fairly straightforward to translate instructions from the assembly language to the machine language. 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!

1.3 Advantages of High-Level Languages

High-level languages are preferred to program applications, as they provide a convenient abstraction of the underlying system suitable for problem solving. Here are some advantages of programming in a high-level language:

1. *Program development is faster.*
   Many high-level languages provide structures (sequential, selection, iterative) that facilitate program development. Programs written in a high-level language are relatively

small compared to the equivalent programs written in an assembly language. These programs are also easier to code and debug.

2. *Programs are easier to maintain.*
   Programming a new application can take from several weeks to several months and the life cycle of such an application software can be several years. Therefore, it is critical that software development be done with a view of 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 are portable.*
   High-level language programs contain very few processor-dependent details. As a result, they can be used with little or no modification on different computer systems. By contrast, assembly language programs are processor-specific.

To illustrate the differences between programs written in C and assembly languages, Section 1.7 presents a concrete example that multiplies two 16-bit integers. You can get an idea of how readable and compact the code written in C is by comparing the C `mult` procedure (see Program 1.2 on page 17) with the Pentium assembly language version (see Program 1.3 on page 17). A more detailed discussion is deferred until Section 1.7.

1.4 Why Program in the Assembly Language?

The previous section gives enough reasons to discourage you from programming in the assembly language. However, there are two main reasons why programming is still done in the assembly language: (1) efficiency, and (2) 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 executable 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 be more compact than those written in a high-level language.

Time efficiency refers to the time taken to execute a program. Obviously a program that runs faster is said to be better from the time-efficiency point of view. If we craft assembly language programs carefully, they tend to run faster than their high-level language counterparts. Section 1.7 demonstrates this advantage through an example.

As an aside, we can also define a third objective: how fast a program can be developed (i.e., write code and debug). This objective is related to the *programmer productivity*, and the assembly language loses the battle to high-level languages, as discussed in the last section.

The superiority of the 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 the assembly language may not be substantial. Second, the cost of memory 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 is both space- and time-efficient. However, there are systems such as embedded controllers and handheld devices in which space efficiency is important.

One of the main reasons for writing programs in the assembly language is to generate code that is time-efficient. The superiority of assembly language programs in producing efficient code is a direct manifestation of *specificity*. That is, assembly language programs contain only the code that is necessary 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. Although the gap is narrowing with improvements in compiler technology, the assembly language still retains its advantage for now.

The other main reason for writing assembly language programs is to have direct control over 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 a device driver to a new scanner on the market almost certainly requires programming in the assembly language. Since the assembly language does not impose any restrictions, you can have direct control over the system hardware. If you are developing system software, you cannot avoid writing assembly language programs.

1.5 Typical Applications

We have identified three advantages to programming in an assembly language.

1. Time efficiency
2. Accessibility to hardware
3. Space efficiency

Time efficiency: Applications for which the execution speed is important fall under two categories:

1. Time convenience (to improve performance)
2. Time-critical (to satisfy functionality)

Applications in the first category benefit from time-efficient programs because it is convenient or desirable. However, time efficiency is not absolutely necessary for their operation. For example, a graphics package that scales an object instantaneously is more pleasant to use than the one that takes noticeable time.

In *time-critical applications*, tasks have to be completed within a specified time period. These applications, also called *real-time applications*, include aircraft navigation systems, process control systems, robot control software, communications software, and target acquisition (e.g., missile tracking) software.

Accessibility to hardware: System software often requires direct control over the system hardware. Examples include operating systems, assemblers, compilers, linkers, loaders, device drivers, and network interfaces. Some applications also require hardware control. Video games are an obvious example.

Space efficiency: As indicated in Section 1.4, for most systems, compactness of application code is not a major concern. However, in portable and handheld devices, code compactness is an important factor. Space efficiency is also important in spacecraft control systems.

1.6 Why Learn the Assembly Language?

Programming in the assembly language is a tedious and error-prone process. The natural preference of a programmer is to program in some high-level language. We discussed a few good reasons why some applications cannot be programmed in a high-level language. Even these applications do not require the whole program to be written in the assembly language. In such instances, a small part of the program is written in the assembly language and the rest is written in some high-level language. Such programs are called *hybrid* or *mixed-mode* programs. In Chapter 17, we discuss how we can write such hybrid programs.

Learning the assembly language has both practical and educational purposes. Even if you don't intend to program in an assembly language, studying it gives you a good understanding of computer systems. When you program in a high-level language, you are provided only a "black-box" view of the system. When programming in the assembly language, you need to understand the internal details of the system (for example, you need to know about processor internal registers). To understand the assembly language is to understand the computer system itself!

This book exposes you to the assembly languages of both CISC and RISC processors. We use the popular Pentium processor to represent the CISC category. We study RISC processors by looking at the MIPS assembly language. Studying these two assembly languages gives you a solid foundation to understand the differences between the CISC and RISC design philosophies and how they impact execution speed of your programs.

A final reason to learn the assembly language is the personal satisfaction that comes with learning something complex. Sure, learning the assembly language is more difficult than learning Java. But the assembly language gives you complete control over the system hardware. You feel powerful with the assembly language on your side, making the time spent learning assembly language worth your while. The insights provided by the assembly language will benefit you even if you program only in high-level languages.

1.7 Performance: C Versus Assembly Language

Now let's see how much better we can do by writing programs in assembly language. As an example, consider multiplying two 16-bit integers. Our strategy is to write the multiplication procedure in C (a representative high-level language) and in the Pentium assembly language and compare their execution times.

1.7.1 Multiplication Algorithm

The Pentium instruction set has two instructions for multiplication: one for unsigned integers and the other for signed integers. These instructions can be used to multiply two integers that can take up to 32 bits to represent them. For multiplying larger numbers, we have to use one of the algorithms discussed in Chapter 7 (see Section 7.4.2 on page 225).

Here we consider the algorithm that is based on the longhand multiplication. This algorithm, shown below, takes two n-bit unsigned integers and produces a $2n$-bit product.

```
product := 0
for ($i = 1$ to $n$)
    if (least significant bit of the multiplier = 1)
    then
        product := product + multiplicand
    end if
    Left shift the multiplicand by one bit position
    Right shift the multiplier by one bit position
end for
```

More details on this algorithm are given in Appendix A.

The main program is shown in Program 1.1 on page 16. To avoid the influence of I/O (i.e., the `printf` and `scanf` statements), we time only the `mult` procedure. To do this, we use `clock()`, which is defined in the `time.h` header file. When `clock()` is invoked, it gives the current clock value in terms of number of clock ticks. The number of clock ticks per second is defined by `CLOCKS_PER_SEC`. Thus, to obtain the multiplication time in seconds, we have to divide the clock ticks by `CLOCKS_PER_SEC`.

The C version of the `mult` procedure is given in Program 1.2 (page 17). This procedure multiplies two 16-bit integers. As you can see from this program listing, it directly follows the algorithm described before.

The assembly language version of the procedure is shown in Program 1.3. As you can see from this code, the assembly language statements are inserted into the procedure using the `asm` construct. For this reason, this method is called *inline assembly*. We give more details on this method in Chapter 17. At this time, you are not expected to make any sense out of this program.

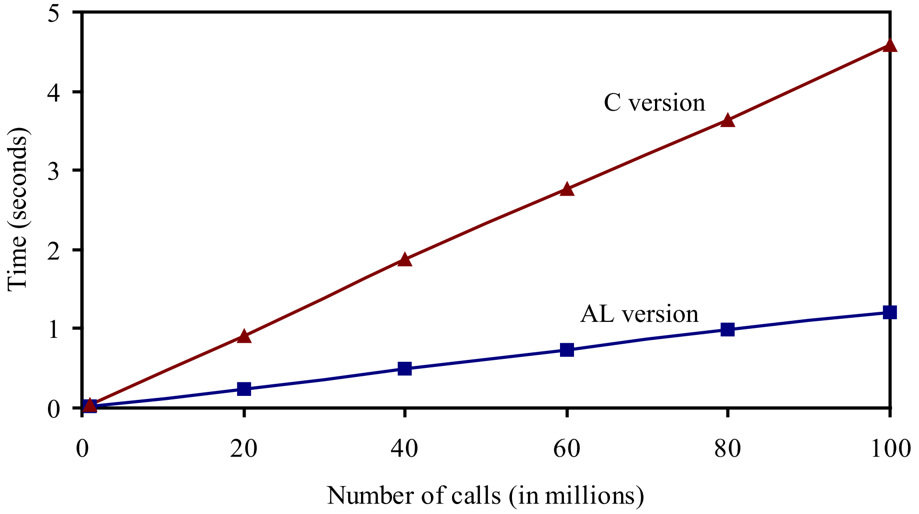


Figure 1.2 Multiplication time comparison on a 2.4-GHz Pentium 4 system: C version uses the multiplication procedure shown in Program 1.2; the assembly language (AL) version uses the assembly language procedure shown in Program 1.3.

Speedup

Let us now look at the potential performance benefit we can get from using the assembly language. Toward this end, we present the multiplication times for the C and assembly language versions in Figure 1.2. These timings were obtained on a 2.4-GHz Pentium 4 system running Red Hat Linux 8.0. The y-axis gives the multiplication time in seconds.

It is clear from this plot that the assembly language version runs substantially faster than the C version. This plot substantiates our claim that assembly language programs are time-efficient. For example, to execute the procedure 100 million times, the C version takes about 3.5 seconds more than the assembly language version.

To quantify the performance difference, let us look at the speedup. We define speedup as

$$\text{Speedup} = \frac{\text{Execution time of the C version}}{\text{Execution time of the assembly language version}}$$

Speedup values greater than 1 indicate that performance of the assembly language version is better—the higher the speedup, the better the assembly language performance. For our application, we get a speedup of about 4.

The reader should be cautioned that the improvement obtained by the assembly language programs depends on the application, compiler, and the type of processor, and so on. It is also important to write an efficient assembly language code in order to get better performance. If we write sloppy assembly language code, it may run slower than the compiler-generated code. This implies that critical analysis and efficient coding are very important to realize the potential performance gains from the assembly language.

In practice, assembly language programming is limited to critical sections of a program. When we say critical, we mean either due to the nature of the application (e.g., real-time constraints) or due to performance reasons.

1.8 Summary

We introduced assembly language and discussed where it fits in the hierarchy of computer languages. Our discussion focused on the usefulness of high-level languages vis-à-vis the assembly language. We noted that high-level languages are preferred, as their use aids in faster program development, program maintenance, and portability. Assembly language, however, provides two chief benefits: faster program execution, and access to system hardware.

In the last section, we used an example to demonstrate the performance advantage of programming in 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:

- Assemblers
- Assembly language
- CISC
- Machine language
- Mixed-mode programs
- Programmer productivity
- RISC
- Space efficiency
- Speedup
- Time efficiency

1.9 Exercises

1–1 What is the relationship between assembly language and machine language? Under what circumstances, if any, do you consider programming in machine language?

1–2 Accessibility to hardware is touted as one of the reasons for programming in assembly language. Discuss why we can't have full control over hardware by using a high-level language.

1–3 Why is assembly language called a low-level language and C a high-level language?

1–4 Why is portability of programs important? When portability is important, would you choose C or assembly language?

1–5 We briefly introduced the CISC and RISC processor types in this chapter. From this discussion, give some of the differences between these two processor types.

1–6 What is programmer's productivity? Discuss how a programming language can affect programmer's productivity.

1.10 Programming Exercises

1–P1 Compile and run the C and assembly language versions of the multiplication program on your machine. The next section gives details on compiling these programs. Compare the relative performance of the C and assembly language versions.

1.11 Program Listings

This section gives the source code listings of

`mult16m.c`	main program
`mult16c.c`	multiplication procedure—C version
`mult16inline.c`	multiplication procedure—assembly language version

Compilation is straightforward. The command

```
gcc -O2 -o c.out mult16m.c mult16c.c
```

can be used to compile the C version. This produces the executable file `c.out`. We can compile the assembly language version using the command

```
gcc -O2 -o asm.out mult16m.c mult16inline.c
```

This produces the `asm.out` executable file.

Program 1.1 The C main program

```
/*****************************************************
 * This program calls the multiply procedure a large *
 * number of times and prints the execution time.    *
 *****************************************************/
#include        <stdio.h>
#include        <time.h>

int main(void)
{
        clock_t start, finish;
        int value1=1000, value2=4096;
        int i, j, n;

        extern long mult(int, int);

        printf ("Please input repeat count: ");
        scanf("%d", &n);

        start = clock();   /* start clock */
        for (j=0; j<n; j++)
            for (i=0; i<1000000; i++)
                mult(value1, value2);
        finish = clock();  /* stop clock */

        printf("Multiplication took %f seconds to finish.\n",
         ((double)(finish-start))/ CLOCKS_PER_SEC);

        return 0;
}
```

Program 1.2 The `mult` procedure—C version

```
/**************************************************************
 * This procedure multiplies two 16-bit integers and returns *
 * their product. It uses the algorithm given in Chapter 1.  *
 **************************************************************/
 long mult(int value1, int value2)
 {
     long  product=0;
     int  i;

     for (i=0; i < 16; i++)
     {
         if (value2 & 1)
             product += value1;
         value1 <<= 1;
         value2 >>= 1;
     }
     return(product);
 }
```

Program 1.3 The `mult` procedure—Assembly language version

```
/***************************************************************
 * This procedure uses inline assembly code to multiply two    *
 * 16-bit integers. It uses the algorithm given in Chapter 1.  *
 ***************************************************************/
 long mult (int value1, int value2)
 {
     long  product=0;
     asm("repeat1: bsfl   %2,%%ecx;  "
         "         jz     done;      "
         "         shll   %%cl,%1;   "
         "         addl   %1,%0;     "
         "         btcl   %%ecx,%2;  "
         "         jmp    repeat1;   "
         "done:                      "
         :"=r"(product)
         :"r"(value1), "r"(value2), "0"(product)
         :"%ecx","%edx","cc");
     return(product);
 }
```

Chapter 2

Basic Computer Organization

Objectives

- To provide a high-level view of computer organization
- To describe processor organization details
- To discuss memory organization and structure
- To introduce how input/output devices are interfaced
- To illustrate the importance of data alignment

Programming in a high-level language does not require a detailed knowledge of the system hardware. Assembly language programmers, however, should have some basic understanding of the underlying system architecture. A high-level view of computer systems, presented in Section 2.1, consists of three major components: a processor, a memory unit, and input/output devices.

The next three sections discuss these three components in detail. Section 2.2 discusses the basic processor execution cycle. The following two sections look at the number of addresses used in the instructions and how the control flow is altered. Section 2.5 presents some basic concepts about the memory system. Section 2.6 gives a brief overview of how input/output devices are interfaced to the system.

Section 2.7 discusses how data alignment affects execution time of programs. We use the bubble sort example discussed in Chapter 5 to illustrate the impact of data alignment. Section 2.8 concludes the chapter with a summary.

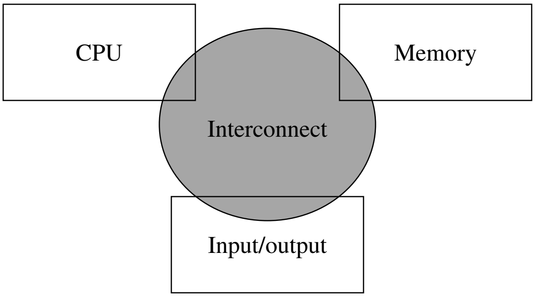


Figure 2.1 High-level view of a computer system.

2.1 Basic Components of a Computer System

A computer system has three main components: a central processing unit (CPU) or processor, a memory unit, and input/output (I/O) devices (see Figure 2.1). These three components are interconnected by a *system bus*. The term “bus” is used to represent a group of electrical signals or the wires that carry these signals. Figure 2.2 shows details of how they are interconnected and what actually constitutes the system bus. As shown in this figure, the three major components of the system bus are the address bus, data bus, and control bus.

The width of the address bus determines the memory addressing capacity of the processor. The width of the data bus indicates the size of the data transferred between the processor and memory or I/O device. For example, the 8086 processor has a 20-bit address bus and a 16-bit data bus. The amount of physical memory that this processor can address is 2^{20} bytes, or 1 MB, and each data transfer involves 16 bits. The Pentium, on the other hand, has 32 address lines and 64 data lines. Thus, the Pentium can address up to 2^{32} bytes, or a 4-GB memory. Furthermore, each data transfer can move 64 bits. In comparison to the Pentium, Intel’s 64-bit processor Itanium uses 64 address lines and 128 data lines.

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 asserted. Similarly, when the processor is reading from an I/O device, the I/O read signal is asserted.

The system memory, also called *main memory* or *primary memory*, is used to store both program instructions and data. I/O devices such as the keyboard and display are used to provide user interface. I/O devices are also used to interface with secondary storage devices such as disks.

The system bus is the communication medium for data transfers. Such data transfers are called the *bus transactions*. Some examples of bus transactions are memory read, memory write, I/O read, I/O write, and interrupt. Depending on the processor and the type of bus used,

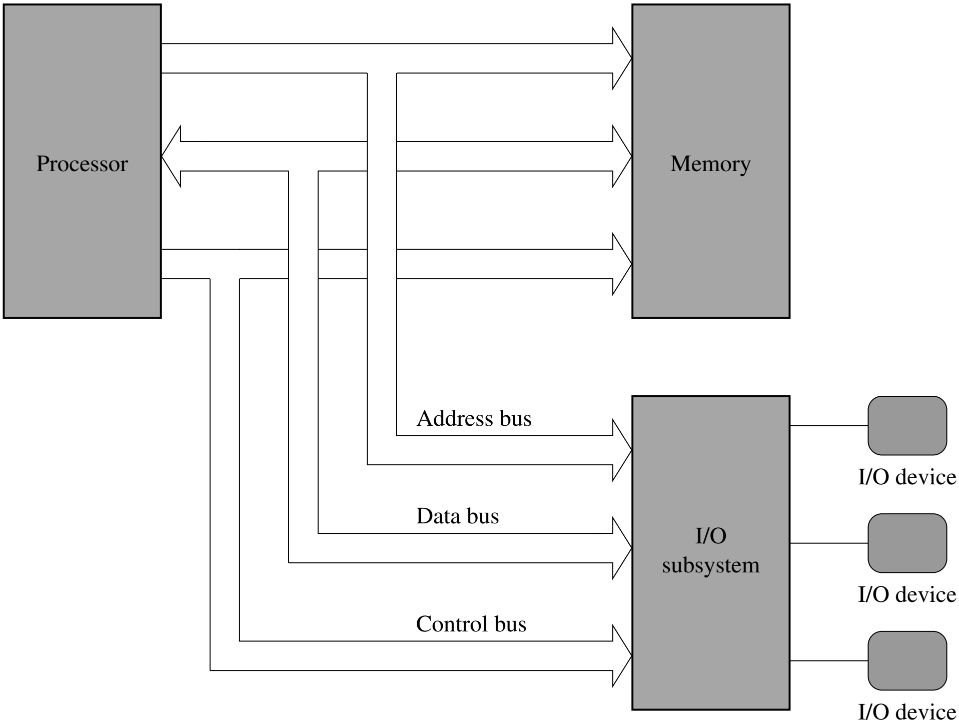


Figure 2.2 Simplified block diagram of a computer system.

there may be other types of transactions. For example, Pentium supports a burst mode of data transfer in which up to four 64 bits of data can be transferred in a burst cycle.

Every bus transaction involves a *master* and a *slave*. The master is the initiator of the transaction and the slave is the target of the transaction. For example, when the processor wants to read data from the memory, it initiates a bus transaction, also called a *bus cycle*, in which the processor is the bus master and memory is the slave. The processor usually acts as the master of the system bus, while components like memory are usually slaves. Some components may act as slaves for some transactions and as masters for other transactions.

When there is more than one master device, which is typically the case, the device requesting the use of the bus sends a *bus request* signal to the bus arbiter using the bus request control line. If the bus arbiter grants the request, it notifies the requesting device by sending a signal on the *bus grant* control line. The granted device, which acts as the master, can then use the bus for data transfer. The bus-request-grant procedure is called the *bus protocol*. Different buses use different bus protocols. In some protocols, permission to use the bus is granted for only one bus cycle; in others, permission is granted until the bus master relinquishes the bus.

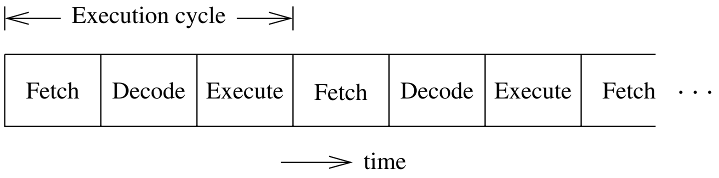


Figure 2.3 Execution cycle of a typical computer system.

2.2 The Processor

The processor acts as the controller of all actions or services provided by the system. It can be thought of as executing the following cycle forever:

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

This process is often referred to as the *fetch-decode-execute* cycle, or simply the *execution* cycle.

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

When we write programs—whether in a high-level language or in an assembly language—we are providing a sequence of instructions to perform a particular task (i.e., solving a problem). These instructions are translated by a compiler/assembler 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.

2.2.1 The Execution Cycle

The execution cycle of a processor is shown in Figure 2.3. *Fetching* an instruction from the main memory involves placing the appropriate address on the address bus and activating the memory read signal on the control bus to indicate to the memory unit that an instruction should be read from that location. The memory unit requires time to read the instruction at the addressed location. This time is called the *access time*. The memory then places the instruction on the data bus. The processor, after instructing the memory unit to read, waits until the instruction is available on the data bus and then reads the instruction.

Decoding involves identifying the instruction that has been fetched from the memory. To facilitate the decoding process, machine language instructions follow a particular instruction-encoding scheme.

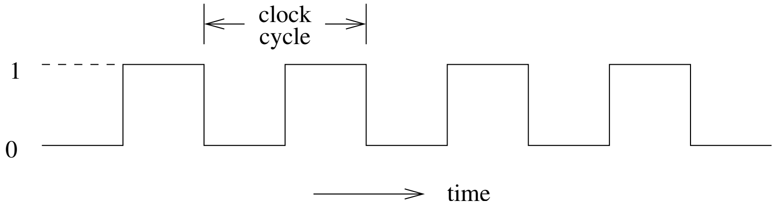


Figure 2.4 Clock signal of a computer system.

To *execute* an instruction, the processor contains hardware consisting of control circuitry and an arithmetic and logic unit (ALU). The control circuitry is needed to provide timing controls as well as to instruct the internal hardware components to perform a specific operation. The ALU is mainly responsible for performing arithmetic operations (such as `add`, `divide`) and logical operations (such as `and`, `or`) on data.

In practice, instructions and data are not fetched, most of the time, from the main memory. There is a high-speed cache memory that provides faster access to instructions and data than the main memory. For example, the Pentium provides a 16 KB on-chip cache. This is divided equally into data cache and instruction cache. The presence of the on-chip cache is transparent to application programs—it helps improve application performance.

2.2.2 The System Clock

The system clock provides a timing signal to synchronize the operations of the system. A clock is a sequence of 1's and 0's, as shown in Figure 2.4. The clock frequency is measured in the number of cycles per second. This number is referred to as Hertz (Hz). We often use the abbreviations MHz and GHz to represent 10^6 and 10^9 cycles per second, respectively.

The system clock defines the *speed* at which the system operates. All processor operations take multiple clock cycles. For example, transfer of data from a memory location to Pentium takes three clock cycles. Thus, the higher the clock rate, the faster the system can work.

The clock period is defined as the length of time taken by one *clock cycle.*

$$\text{Clock period} = \frac{1}{\text{Clock frequency}}$$

For example, a clock frequency of 1 GHz yields a clock period of

$$\frac{1}{1 \times 10^9} = 1 \text{ ns}$$

If it takes three clock cycles to execute an instruction, it takes 3×1 ns = 3 ns.

One way to increase the speed of a computer system is to use a higher clock frequency. For example, if we use a clock of 2 GHz, the instruction execution time reduces from 3 ns to 1.5 ns. Clock frequency increases with improvements in technology. The original IBM PC used a clock of 4.77 MHz. Current technology allows clock frequencies higher than 3 GHz.

Table 2.1 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`
`sub`	`dest,src1,src2`	Subtracts the second source operand at `src2` from the first at `src1` and stores the result in `dest`
`mult`	`dest,src1,src2`	Multiplies the two values at `src1` and `src2` and stores the result in `dest`

2.3 Number of Addresses

One of the characteristics that shapes the architecture of a processor is the number of addresses used in its instructions. 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 processors use either two or three addresses. For example, the MIPS processor uses three addresses whereas the Pentium uses two addresses. However, it is possible to design systems with one or even zero address. In the rest of this section, we give details on these machines.

2.3.1 Three-Address Machines

In three-address machines, instructions carry all three addresses explicitly. Most current processors use three addresses. The MIPS processor we discuss in Chapter 12, for example, uses three addresses. Table 2.1 gives some sample instructions of a three-address machine.

On 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
```

Table 2.2 Sample Two-Address Machine Instructions

Instruction	Semantics
`load dest,src`	Copies the value at `src` to `dest`
`add dest,src`	Adds the two values at `src` and `dest` and stores the result in `dest`
`sub dest,src`	Subtracts the second source operand at `src` from the first at `dest` and stores the result in `dest`
`mult dest,src`	Multiplies the two values at `src` and `dest` and stores the result in `dest`

```
add   A,A,T    ; A = B + C*D - E + F + A
```

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.

2.3.2 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 the Pentium processor details in the next few chapters. Table 2.2 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
```

Since we use only two addresses, we use a load instruction to first copy the C value into a temporary address 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.

2.3.3 One-Address Machines

In the early machines, when memory was expensive and slow, a special set of registers was used to provide one of the input operands as well as to receive the result of the operation. Because of this, these registers are called the *accumulators*. In most machines, there is just a single accumulator register. This kind of design, called the *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 and speeds up the computation by reducing the number of memory accesses.

2.3.4 Zero-Address Machines

In zero-address machines, the 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 5.1 on page 118).

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.

2.3.5 The Load/Store Architecture

In this architecture, instructions operate on values stored in internal processor registers. Only load and store instructions move data between the registers and memory. Table 2.3 gives some sample instructions for the load/store machines. On 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
```

Table 2.3 Sample Load/Store Machine Instructions

Instruction		Semantics
`load`	`Rd,addr`	Loads the `Rd` register with the value at address `addr`
`store`	`addr,Rs`	Stores the value in `Rs` register at address `addr`
`add`	`Rd,Rs1,Rs2`	Adds the two values in `Rs1` and `Rs2` registers and places the result in `Rd` register
`sub`	`Rd,Rs1,Rs2`	Subtracts the value in `Rs2` from that in `Rs1` and places the result in `Rd` register
`mult`	`Rd,Rs1,Rs2`	Multiplies the two values in `Rs1` and `Rs2` and places the result in `Rd` register

In this code, we assume 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 processors tend to have many more registers than CISC processors. For example, the MIPS processor has 32 registers and the Intel Itanium processor has 128 registers. Compared to this, the Pentium has only 10 registers.

RISC machines as well as vector processors use this architecture, which reduces the instruction size substantially. If we assume that memory 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 2.5. In this figure, we use 5 bits to specify a register as we assume that there are 32 registers as in the MIPS processors. MIPS processor details are given in Chapters 12 and 13.

2.3.6 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, in three- and two-address machines, there is no need for the internal data registers. However, having a few internal registers improves performance by cutting down the number of memory accesses required to execute a program. RISC processors typically have a large number of registers.

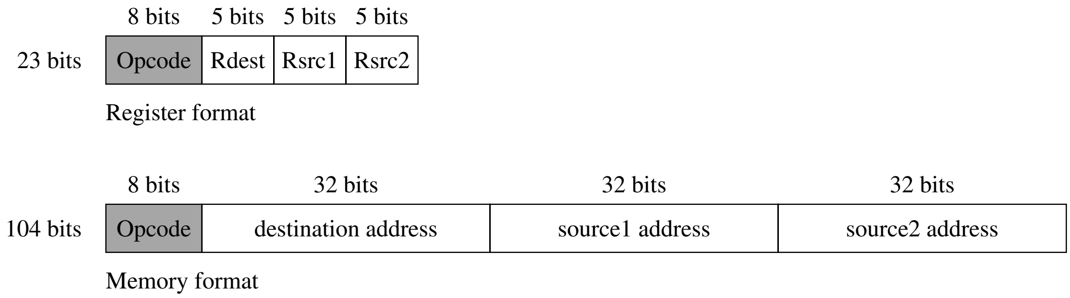


Figure 2.5 A comparison of the instruction size when the operands are in registers versus memory.

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.

Registers available in the Pentium processor are described in the next chapter. MIPS processor registers are discussed in Chapter 12.

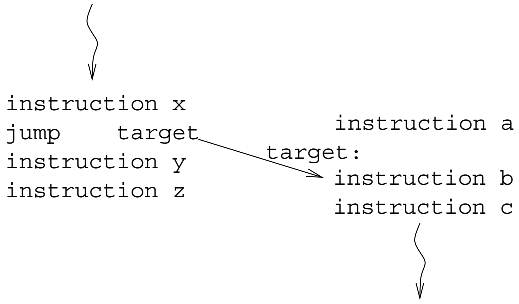
2.4 Flow of Control

Program execution, by default, proceeds sequentially. This default behavior is due to the semantics associated with the execution cycle described in Section 2.2.1. 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. 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 the MIPS 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, we can use procedure calls to 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 14.

2.4.1 Branching

Branching is implemented by means of a branch instruction. This instruction carries the address of the target instruction explicitly. Branch instructions in processors such as the Pentium are also called the jump instructions. Processors suppport two types of branches: uncondi-



```
instruction x
jump    target              instruction a
instruction y          target:
instruction z               instruction b
                            instruction c
```

Figure 2.6 Control flow in branching.

tional and conditional. In both cases, the transfer control mechanism remains the same (see Figure 2.6).

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:

```
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. The processor only supports the `j` instruction.

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 only if 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. Then we can use a conditional jump instruction to jump to the target location if the specified condition bit is set. The Pentium jump instructions are discussed in detail in Part II.
- *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 processor provides several branch instructions that test and branch (for a quick peek, see Table 13.7 on page 374). 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. More details on the MIPS processor branch instructions are given in Chapter 13.

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 processors, for example, do not use condition registers. Instead, it uses exceptions to flag the overflow condition. On the other hand, the Pentium uses condition registers, which are called the flags register.

Some instruction sets provide branches based on comparisons to zero. Some example processors that provide this type of branch instructions include the MIPS processors.

2.4.2 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 2.6. 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 2.7.

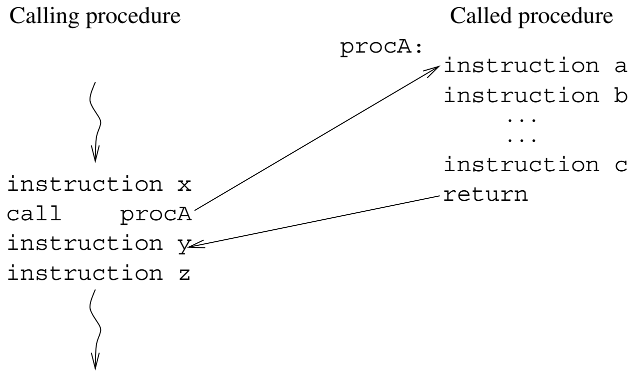


Figure 2.7 Control flow in procedure calls.

From Figures 2.6 and 2.7, 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. In the assembly language, we must specify the end of a procedure by using the return instruction.

- *Return Address:* How does the processor know where to return after completing a procedure? 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 the 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. Both MIPS and Pentium processors 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 control back to the instruction following the procedure call. A more detailed description of the procedure call mechanism is found in Chapter 5.

  MIPS processors allow any general-purpose register to store the return address. The return statement specifies this register. The format of the return statement is

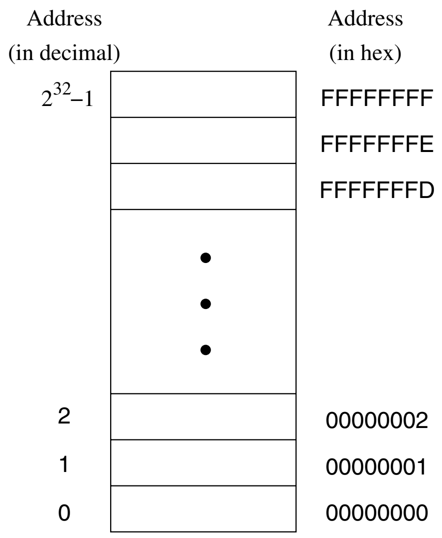


Figure 2.8 Logical view of the system memory.

```
jr    $ra
```

where ra is the register that contains the return address. Chapter 13 gives more details on this instruction.

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 the processor's internal 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 read 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 the 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 5.

2.5 Memory

The memory of a computer system consists of tiny electronic switches, with each switch set in one of two states: *open* or *closed*. It is, however, more convenient to think of these states

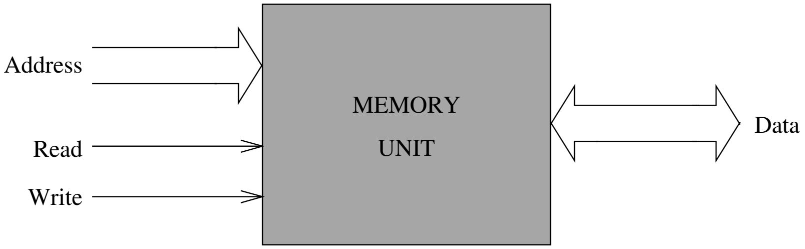


Figure 2.9 Block diagram of the system memory.

as 0 and 1 rather than open and closed. A single such switch can be used to represent two (i.e., binary) numbers: a zero and a one. Thus, each switch can represent a *binary digit* or *bit*, as it is known. The memory unit consists of millions of such bits. In order to make memory more manageable, bits are organized into groups of eight bits called *bytes*. Memory can then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory can be identified by its sequence number starting with 0, as shown in Figure 2.8. This is referred to as the *memory address* of the byte. Such memory is called *byte addressable* memory.

The Pentium can address up to 4 GB (2^{32} bytes) of main memory (see Figure 2.8). 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* (MAS). The memory address space of a system is determined by the address bus width of the processor used in the system. Typically, 32-bit processors support 32-bit addresses. For example, the MIPS processor we discuss in Chapter 12 also supports 4-GB memory address space.

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.

2.5.1 Two Basic Memory Operations

The memory unit supports two fundamental operations: *read* and *write*. The *read operation* reads a previously stored data and the *write operation* stores a value in memory. Both of these operations require an address in memory from which to read a value or to which to write a value. In addition, the write operation requires specification of the data to be written. The block diagram of the memory unit is shown in Figure 2.9. The address and data of the memory unit are connected to the address and data buses, 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.

Memory transfer rates can be measured by the bandwidth metric. It specifies the number of bytes transferred per second. For example, a Pentium system with the PC133 memory can transfer 8 bytes at a frequency of 133 times per second. This gives us a bandwidth of $8 * 133 = 1064$ MB/s.

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, on the other hand, is destructive, as writing a value into a location destroys the old contents of that memory location.

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 it 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. Step 2 is performed during the second clock cycle. 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.

2.5.2 Types of Memory

The memory unit can be implemented using a variety of memory chips—different speeds, different manufacturing technologies, and different sizes. The two basic types of memory are the *read-only memory* and *read/write memory*.

A basic property of memory systems is that they are random access memories in that accessing any memory location (for reading or writing) takes the same time. Contrast this with data stored on a magnetic tape. Access time on the tape depends on the location of the data.

Volatility is another important property of a memory unit. A *volatile* memory requires power to retain its contents. A *nonvolatile* memory can retain its values even in the absence of power.

Read-Only Memories

Read-only memory (ROM) allows only read operations to be performed. As the name suggests, we cannot write into this memory. The main advantage of ROM is that it is nonvolatile. Most ROM is factory-programmed and cannot be altered. The term *programming* in this context refers to writing values into a ROM. This type of ROM is cheaper to manufacture in large quantities than other types of ROM. The program that controls the standard input and output functions (called BIOS), for instance, is kept in ROM. Current systems use the flash memory rather than a ROM (see our discussion later).

Other types of ROM include *programmable ROM* (PROM) and *erasable PROM* (EPROM). PROM is useful in situations where the contents of ROM are not yet fixed. For instance, when the program is still in the development stage, it is convenient for the designer to be able to program the ROM locally rather than at the time of manufacture.

In PROM, a fuse is associated with each bit cell. If the fuse is on, the bit cell supplies a 1 when read. The fuse has to be burned to read a 0 from that bit cell. When PROM is manufactured, its contents are all set to 1. To program PROM, selective fuses are burned (to introduce 0's) by sending high current. This is the writing process and is not reversible (i.e., a burned fuse cannot be restored). EPROM offers further flexibility during system prototyping. Contents of an EPROM can be erased by exposing it to ultraviolet light for a few minutes. Once erased, the EPROM can be reprogrammed.

Electrically erasable PROMs (EEPROMs) allow further flexibility. By exposing to ultraviolet light, we erase *all* the contents of an EPROM. EEPROMs, on the other hand, allow the user to selectively erase contents. Furthermore, erasing can be done in place; there is no need to place it in a special ultraviolet chamber.

Flash memory is a special kind of EEPROM. One main difference between the EEPROM and flash memory lies in how the memory contents are erased. The EEPROM is byte-erasable whereas flash memory is block-erasable. Thus, writing in the flash memory involves erasing a block and rewriting it.

Current systems use flash memory for BIOS so that changing BIOS versions is fairly straightforward (you just have to "flash" the new version). Flash memory is also becoming very popular as a removable media. The SmartMedia, CompactFlash, and Sony's Memory Stick are all examples of various forms of removable flash media.

Flash memory, however, is slower than the RAMs we discuss next. For example, flash memory cycle time is about 80 ns whereas the corresponding value for RAMs is about 10 ns. Nevertheless, since flash memories are nonvolatile, they are used in applications where this property is important. Apart from BIOS, we see them in devices like digital cameras and video game systems.

Read/Write Memory

Read/write memory is commonly referred to as *random access memory* (RAM), even though ROM is also random access memory. This terminology is so entrenched in the literature that we follow it here with a cautionary note that RAM actually refers to RWM.

Read/write memory can be divided into *static* and *dynamic* categories. Static random access memory (SRAM) retains the data, once written, without further manipulation so long as the source of power holds its value. SRAM is typically used for implementing the processor registers and cache memories.

The bulk of main memory in a typical computer system, however, consists of dynamic random access memory (DRAM). DRAM is a complex memory device that uses a tiny capacitor to store a bit. A charged capacitor represents 1 bit. Since capacitors slowly lose their charge due to leakage, they must be *refreshed* periodically to replace the charges representing 1 bit. A typical refresh period is about 64 ms. Reading from DRAM involves testing to see if the corresponding bit cells are charged. Unfortunately, this test destroys the charges on the bit cells. Thus, DRAM is a destructive read memory.

For proper operation, a read cycle is followed by a restore cycle. As a result, the DRAM cycle time, the actual time necessary between accesses, is typically about twice the read access time, which is the time necessary to retrieve a datum from the memory.

Several types of DRAM chips are available. We briefly describe some of the most popular types next.

FPM DRAMs Fast page-mode (FPM) DRAMs are an improvement over the previous generation DRAMs. FPM DRAMs exploit the fact that we access memory sequentially, most of the time. To know how this access pattern characteristic is exploited, we have to look at how the memory is organized. Internally, the memory is organized as a matrix of bits. For example, a 32-Mb memory could be organized as 8 K rows (i.e., 8192 since K = 1024) and 4-K columns. To access a bit, we have to supply a row address and a column address. In the FPM DRAM, a page represents part of the memory with the same row address. To access a page, we specify the row address only once; we can read the bits in the specified page by changing the column addresses. Since the row address is not changing, we save on the memory cycle time.

EDO DRAMs Extended Data Output (EDO) DRAM is another type of FPM DRAM. It also exploits the fact that we access memory sequentially. However, it uses pipelining to speed up memory access. That is, it initiates the next request before the previous memory access is completed. A characteristic of pipelining inherited by EDO DRAMs is that single memory reference requests are not sped up. However, by overlapping multiple memory access requests, it improves the memory bandwidth.

SDRAMs Both FPM DRAMs and EDO DRAMs are asynchronous in the sense that their data output is not synchronized to a clock. The synchronous DRAM (SDRAM) uses an external clock to synchronize the data output. This synchronization reduces delays and thereby improves the memory performance. The SDRAM memories are used in systems that require memory satisfying the PC100/PC133 specification. SDRAMs are dominant in low-end PC market and are cheap.

DDR SDRAMs The SDRAM memories are also called single data rate (SDR) SDRAMs as they supply data once per memory cycle. However, with increasing processor speeds, the processor bus (also called front-side bus or FSB) frequency is also going up. For example, Pentium systems now have 533-MHz FSB that supports a transfer rate of about 4.2 GB/s. To satisfy this transfer rate, SDRAMs have been improved to provide data at both rising and falling edges of the clock. This effectively doubles the memory bandwidth and satisfies the high data transfer rates of faster processors.

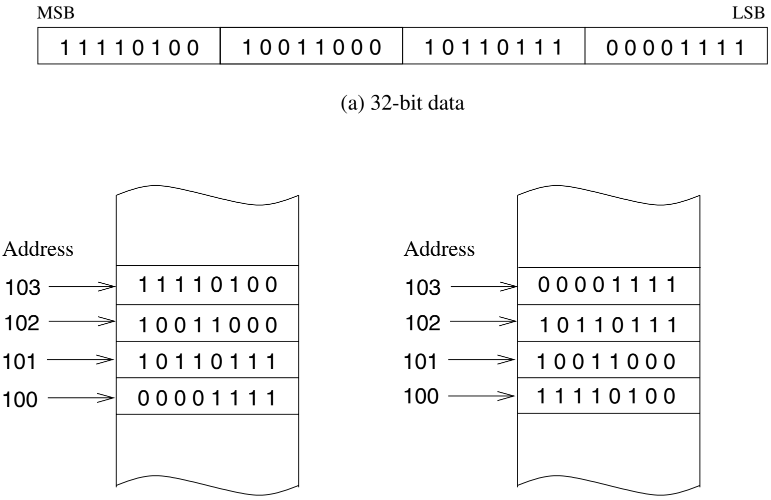
RDRAMs Rambus DRAM (RDRAM) takes a completely different approach to increase the memory bandwidth. A technology developed and licensed by Rambus, it is a memory subsystem that consists of the RAM, RAM controller, and a high-speed bus called the Rambus channel. Like the DDR DRAM, it also performs two transfers per cycle. In contrast to the 8-byte-wide data bus of DRAMs, Rambus channel is a 2-byte data bus. However, by using multiple channels, we can increase the bandwidth of RDRAMs. For example, a dual-channel RDRAM operating at 533 MHz provides a bandwidth of $533 * 2 * 4 = 4.2$ GB/s, sufficient for the 533-MHz FSB systems.

From this brief discussion it should be clear that DDR SDRAMs and RDRAMs compete with each other in the high-end market. The race between these two DRAM technologies continues as Intel boosts its FSB to 800 MHz.

2.5.3 Storing Multibyte Data

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 2.10a.

Suppose we want to store these four bytes of data in memory at locations 100 through 103. How do we store them? Figure 2.10 shows two possibilities: least significant byte



(a) 32-bit data

(b) Little-endian byte ordering

(c) Big-endian byte ordering

Figure 2.10 Two byte-ordering schemes.

(Figure 2.10b) or most significant byte (Figure 2.10c) 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. Later chapters give details on these instructions.

2.6 Input/Output

Input/output (I/O) devices provide the means by which a computer system can interact with the outside world. An I/O device can be purely an input device (e.g., keyboard, mouse), purely

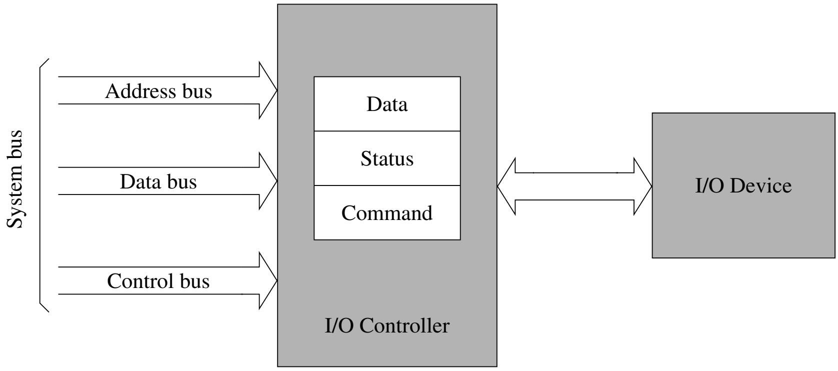


Figure 2.11 Block diagram of a generic I/O device interface.

an output device (e.g., printer, display screen), or both an input and output device (e.g., disks). Here we present a brief overview of the I/O device interface. Chapters 14 and 15 provide more details on I/O interfaces.

Computers use I/O devices (also called *peripheral devices*) for two major 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 and the I/O device.

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 processor would have to understand and respond appropriately to each I/O device. This would cause the processor 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, special I/O controller chips are 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 the 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 2.11. When the processor 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 processor 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 data register holds the character to be printed, and the command register tells the controller the operation requested by the processor (for example, 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;
- Place a character to be printed in the data register;
- Set the command register to initiate the transfer.

The processor accesses the internal registers of an I/O controller through what are known as *I/O ports*. An I/O port is simply the address of a register associated with an I/O controller.

There are two ways of mapping I/O ports. Some processors such as the MIPS map I/O ports to memory addresses. This is called *memory-mapped I/O*. In these systems, writing to an I/O port is similar to writing to a memory address. Other processors like the Pentium have an *I/O address space* that is separate from the memory address space. This technique is called *isolated I/O*. In these systems, to access the I/O address space, special I/O instructions are needed. Pentium provides two instructions—`in` and `out`—to access I/O ports. The `in` instruction can be used to read from an I/O port and the `out` for writing to an I/O port. Chapter 15 gives more details on these instructions.

Pentium provides 64 KB of I/O address space. This address space can be used for 8-bit, 16-bit, and 32-bit I/O ports. However, the combination cannot be more than the 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.

Systems designed with processors supporting the isolated I/O have the flexibility of using either the memory-mapped I/O or the isolated I/O. Typically, both strategies are used. For instance, devices like the printer or keyboard could be mapped to the I/O address using the isolated I/O strategy; the display screen could be mapped to a set of memory addresses using the memory-mapped I/O.

Accessing I/O Devices As a programmer, you can have direct control on any of the I/O devices (through their associated I/O controllers) when you program in the assembly language. However, it is often a difficult task to access an I/O device without any help. Furthermore, it is a waste of time and effort if everyone has to develop his or her own routine to access I/O devices (called *device drivers*). In addition, system resources could be abused, either unintentionally or maliciously. For instance, an improper disk driver could erase the contents of a disk due to a bug in the driver routine.

To avoid these problems and to provide a standard way of accessing I/O devices, operating systems provide routines to conveniently access I/O devices. For example, Linux provides a

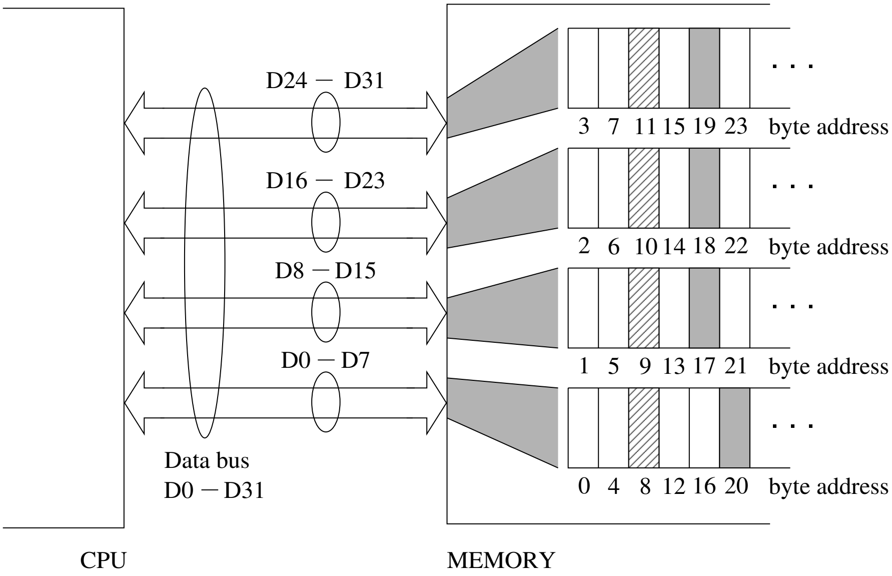

D24 − D31
D16 − D23
D8 − D15
D0 − D7
Data bus
D0 − D31
3 7 11 15 19 23 byte address
2 6 10 14 18 22 byte address
1 5 9 13 17 21 byte address
0 4 8 12 16 20 byte address
. . .
CPU
MEMORY

Figure 2.12 Byte-addressable memory interface to the 32-bit data bus.

set of system calls to access system I/O devices. In Windows, access to I/O devices can be obtained from two layers of system software: the basic I/O system (BIOS), and the operating system. BIOS is ROM resident and is a collection of routines that control the basic I/O devices. Both provide access to routines that control the I/O devices though a mechanism called *interrupts*. Interrupts are discussed in detail in Chapter 14.

2.7 Performance: Effect of Data Alignment

Execution time of a program is influenced by several factors—some of which are under the control of the programmer. Other factors that influence the running time of a program include the clock rate of the system, efficiency of the compiler if the program is written in a high-level language, presence of a cache memory, and so on.

Here we look at the influence of data alignment on the performance of the bubble sort example discussed in Chapter 5 (see Example 5.5 on page 142). One of the factors influencing the sort time is the time required to access the array.

Suppose we want to read a 32-bit variable from the memory. Assume that the data bus is 32 bits wide. If the address of this variable is a multiple of four, the 32-bit data are stored in a single row of memory. Thus the processor can get the data in one read cycle. If this is not true, then the 32-bit data item is spread over two rows. Thus the processor reads two 32-bits of data and assembles the required 32-bit data. This scenario is clearly demonstrated in Figure 2.12.

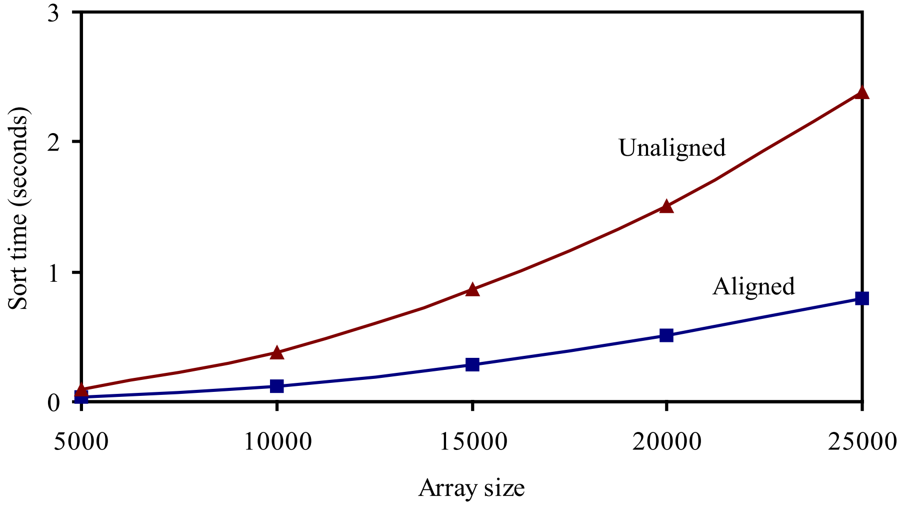


Figure 2.13 Impact of data alignment on the performance of the bubble sort algorithm.

In Figure 2.12, 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.

Figure 2.13 shows the impact of data alignment on the sort time of the bubble sort. These results were obtained on a 2.4-GHz Pentium 4 processor system. The unaligned sort time is approximately three times more than the aligned sort time.

Except for the performance penalty, data alignment is totally transparent to the application. However, to avoid this performance penalty, data should be aligned.

- *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.
- *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 processors such as the Pentium that have a 64-bit-wide data bus. On processors (e.g., 80486) that have 32-bit-wide data bus, a 64-bit data item is read in two bus cycles and alignment at four-byte boundaries is sufficient.

The Intel Pentium family of processors allows aligned and unaligned data items. Of course, unaligned data cause performance degradation. An alignment constraint of this type is referred to as the *soft alignment* constraint. Because of the performance penalty associated with unaligned data, some processors do not allow unaligned data. This alignment constraint is referred to as the *hard alignment* constraint.

2.8 Summary

Programmers should have some basic knowledge about the processor and the system architecture in order to effectively program in the assembly language. This chapter has presented the basics of computer organization.

We started with a high-level view of the system. At this level, a computer system can be thought of as consisting of three main components: a processor, a memory unit, and I/O devices. The remainder of the chapter briefly described these three components.

We also considered the impact of data alignment on the execution time of application programs. By using the bubble sort example discussed in Chapter 5, we demonstrated the influence of data alignment on the sort time.

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
- Conditional branch
- End of procedure
- Isolated I/O
- Load/store architecture
- Memory-mapped I/O
- Number of addresses
- Parameter passing
- PC-relative
- Procedure call
- Processor registers
- Return address
- Stack machines
- Unconditional branch
- Wait cycles

2.9 Exercises

2–1 Describe the execution cycle. What is the default mode of execution?

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

2–3 If a system uses a 1.5-GHz clock, what is the clock period?

2–4 If a processor has 64 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.

2–5 What are the characteristics of the load/store architecture that sets it apart from the other architectures discussed in this chapter?

2–6 We stated that the RISC processors use the load/store architecture. What are its advantages over the CISC architectures as exemplified by the Pentium processor?

2–7 Give some of the reasons why instruction execution rate is higher in RISC processors than in CISC processors.

2–8 From the processor point of view, what are the differences between branches and procedures?

2–9 Explain the differences between branches that use absolute address and PC-relative address.

2–10 What are the differences between ROM and RAM?

2–11 Compare and contrast DRAM and SRAM.

2–12 Why do DRAMs need to be refreshed?

2–13 What is the difference between volatile and nonvolatile memories?

2–14 Can you think of a reason why ROMs tend to be nonvolatile and RAMs volatile?

2–15 Consider the Pentium processor with an 800-MHz front-side bus. What is the bandwidth of this bus?

2–16 We stated that DDR SDRAMs and RDRAMs compete for the high-end systems that require higher bandwidth to support 533-MHz FSB. Suppose we use a four-channel RDRAM memory subsystem. What is the clock frequency that this subsystem should operate in order to meet the bandwidth requirement of the last question?

2–17 Discuss why I/O controllers are used to interface I/O devices to the system.

2–18 For each address below, state whether a 32-bit value stored at that address is aligned or not (all numbers are in hex):

(a) 12345678   (c) 9128ADCC
(b) ABCD755A   (d) 38B0F050

2–19 Repeat the above exercise for 64-bit values.

PART II

Pentium Assembly Language

This part is dedicated to Pentium assembly language programming. It should be noted that the Intel 32-bit instruction set architecture is referred to as IA-32 architecture. However, for concreteness, we refer to the IA-32 instruction set as the Pentium instruction set in this book. Bear in mind that there are other implementations of this instruction set beside the Pentium processor. Therefore, our reference to Pentium in this context should be treated as an alias for IA-32; it certainly does not refer only to the Pentium implementation of this instruction set.

This part consists of nine chapters—Chapters 3 through 11. It provides the basics of the assembly language. Before reading this part, you should be familiar with the material presented in Appendices B and C. These appendices give details on how you can assemble and debug the assembly language programs.

This part begins with a description of the Pentium processor organization (Chapter 3). In particular, this chapter gives sufficient details on the 16-bit and 32-bit Intel processors so that you can effectively program in the assembly language. Chapter 4 gives an overview of the assembly language. After covering these two chapters, you should be able to write simple standalone assembly language programs.

To emphasize the importance of modular programming, we introduce procedures early on in Chapter 5. The other chapters in this part expand on the overview given in Chapter 4. Chapter 6 presents the addressing modes supported by the Intel 16-bit and 32-bit processors. This chapter also contains a detailed discussion on the motivation for providing the various addressing modes. Addressing modes are one of the differentiating characteristics of CISC processors.

Chapter 7 discusses the arithmetic instructions and the use of the flags register. Chapters 8 and 9 present conditional and bit manipulation instructions. A feature of these two chapters is that they relate how high-level language statements can be implemented using the instructions discussed in these two chapters. Chapter 10 discusses the string processing instructions in detail. ASCII and BCD arithmetic instructions are presented in Chapter 11.

Chapter 3

The Pentium Processor

Objectives

- To describe the basic organization of the Pentium processor
- To introduce the Pentium protected-mode memory architecture
- To discuss the real-mode memory organization

We discussed processor design space in the last chapter. Now we look at the 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. Section 3.2 presents the internal register details of the Pentium processor. 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 protected- and real-mode memory architectures. Protected-mode architecture is the native mode for the Pentium processor. The real mode is provided to mimic the 16-bit 8086 memory architecture. 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.

3.1 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 microprocessors. 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 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. The 80286 is backward compatible in that it can run the 8086-based software.

Intel introduced its first 32-bit processor—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 of 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-saving mode 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 has also been added.

The Pentium Pro processor has a three-way superscalar architecture. That is, it can execute three instructions per clock cycle. 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. 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.

Table 3.1 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	1978	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
Pentium 4	2000	1500	42 M	32	64	64 GB

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 added more comprehensive power management features including Sleep and Deep Sleep modes to conserve power during idle times.

The Pentium III processor introduced streaming SIMD extensions (SSE), cache prefetch instructions, and memory fences and the single-instruction multiple-data (SIMD) architecture for concurrent execution of multiple floating-point operations. Pentium 4 enhanced these features further. Table 3.1 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 the RISC orientation for their Itanium processors. The Itanium also incorporates several advanced architectural features to provide improved performance for the high-end server market.

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.

3.2 The Pentium Registers

The Pentium has 10 32-bit and 6 16-bit registers. These registers are grouped into general, control, and segment registers. The general registers are further divided into data, pointer, and index registers.

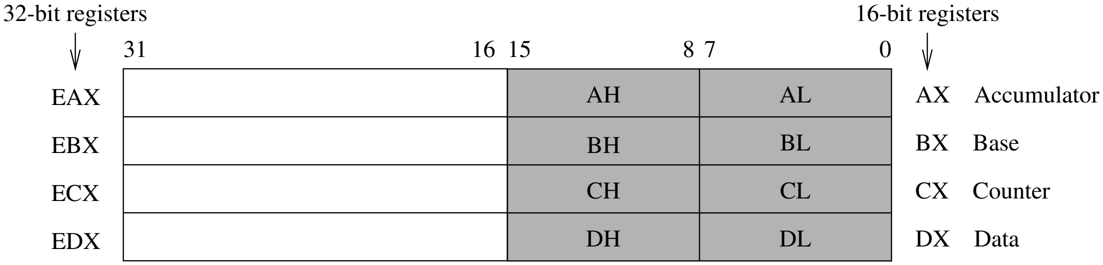


Figure 3.1 Data registers of the Pentium processor (16-bit registers are shown shaded).

3.2.1 Data Registers

There are four 32-bit data registers that can be used for arithmetic, logical, and other operations (see Figure 3.1). 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).

As shown in Figure 3.1, 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.

3.2.2 Pointer and Index Registers

Figure 3.2 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 10). 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 5.

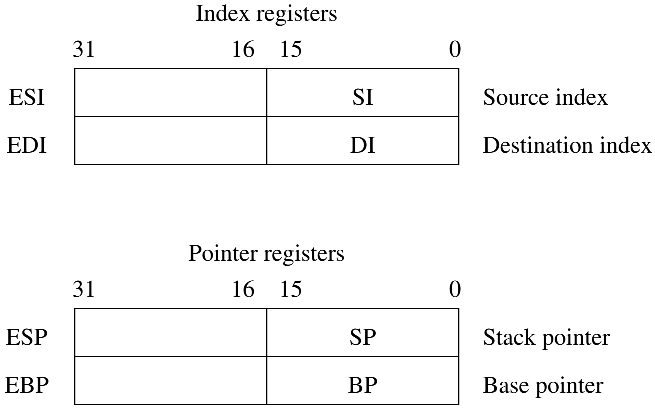


Figure 3.2 Index and pointer registers of the Pentium processor.

3.2.3 Control Registers

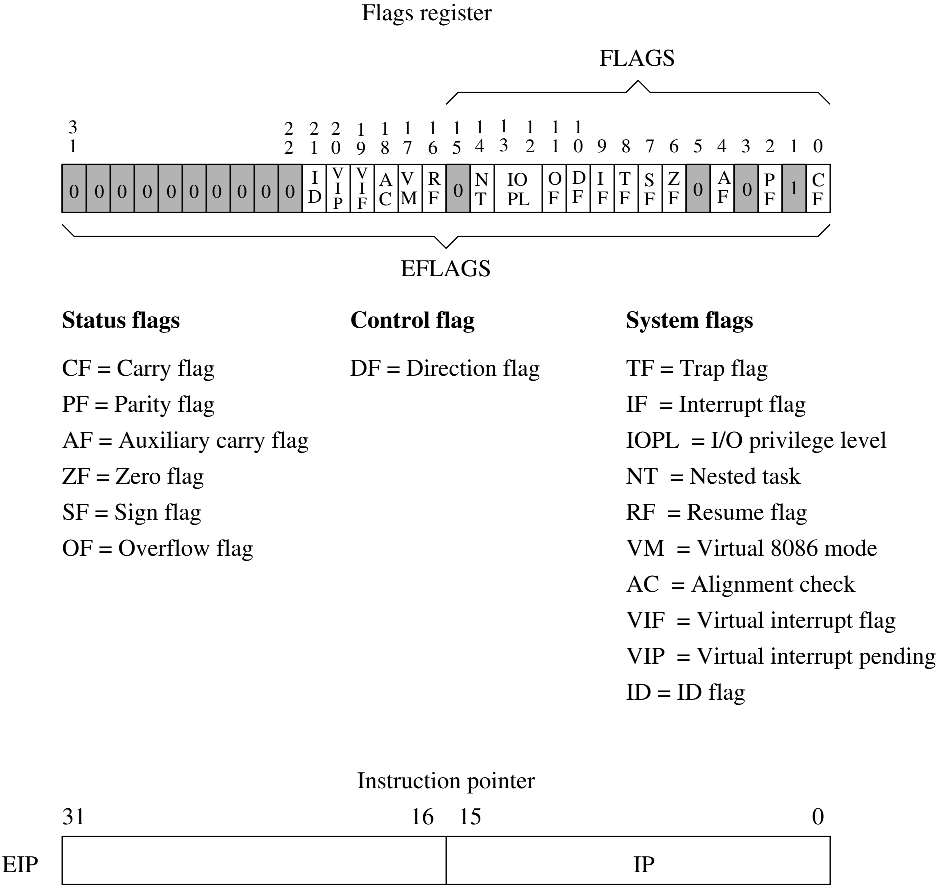
This group of registers consists of two 32-bit registers: the instruction pointer register and the flags register. The processor uses the instruction pointer register to keep track of the location of the next instruction to be executed. Instruction pointer register is sometimes called the program counter register (see our discussion in the last chapter). 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 3.3 and 3.4 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).

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 3.3. Bits of this register can be set (1) or cleared (0). The Pentium provides instructions to set and clear some of the 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) would be set (i.e., ZF = 1). Chapter 7 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 10, which discusses the string instructions supported by the Pentium.



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flag

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

Figure 3.3 Flags and instruction pointer registers of the Pentium processor.

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 14, 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

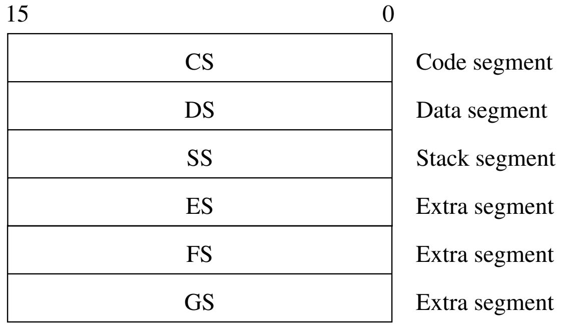


Figure 3.4 The six segment registers of the Pentium processor.

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. We discuss data alignment and its impact on application performance in Section 2.7 on page 41.

3.2.4 Segment Registers

The six 16-bit segment registers of the Pentium are shown in Figure 3.4. 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.

A 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 the program's instructions are stored in the main memory, and the data segment (DS) register points to the data part of the program. The stack segment (SS) register points to the program's stack segment (further discussed in Chapter 5).

The last three segment registers—ES, FS, and GS—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 segment registers to point to the two data segments.

3.3 Protected-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

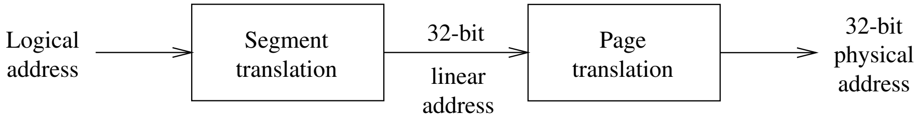


Figure 3.5 Logical to physical address translation process in the protected mode.

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. We discuss the real-mode memory architecture in the next section and devote the rest of this section to describing the protected-mode segmented memory architecture.

In the 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 3.5. 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.

The protected-mode segment translation process is shown in Figure 3.6. In this mode, contents of the segment register are taken as an index into a segment descriptor table to get a descriptor. 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.

3.3.1 Segment Registers

Every segment register has a “visible” part and an “invisible” part, as shown in Figure 3.7. 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 E. The invisible part of the segment registers is automatically loaded by the processor from a descriptor table (described next).

As shown in Figure 3.6, 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 3.8, is 8 bytes long, the processor multiplies the index by 8 and adds the result to the base address of the selected descriptor table.
- *Table Indicator (TI):* This bit indicates whether the local or global descriptor table should be used.

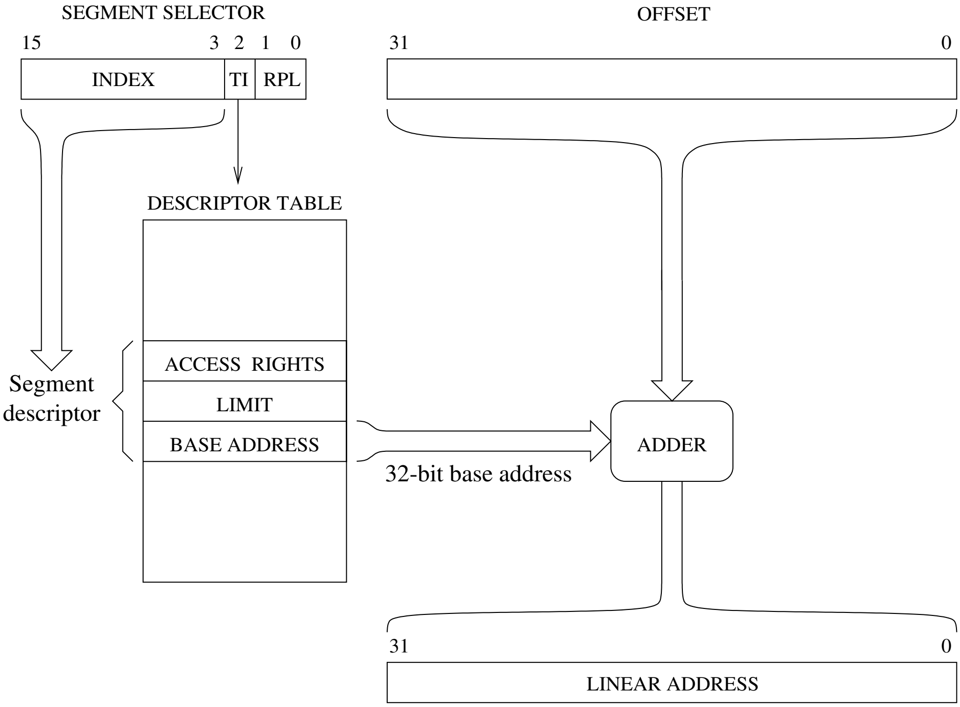


Figure 3.6 Protected-mode address translation.

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. Operating systems don't have to use all the four levels. For example, Linux uses level 0 for the kernel and level 3 for the user programs. It does not use levels 1 and 2.

3.3.2 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 3.8. 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 3.6).

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 3.7 Visible and invisible parts of segment registers.

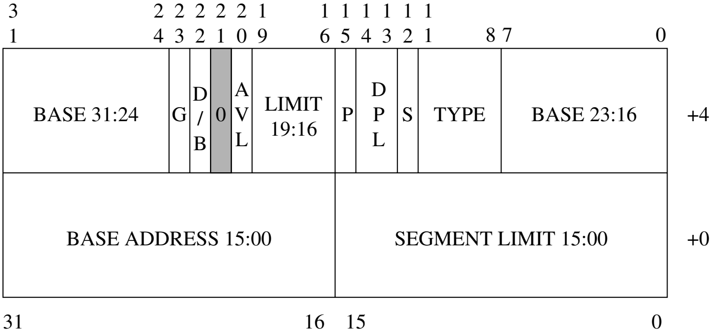


Figure 3.8 A segment descriptor.

- *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. Recall that numbers expressed in the hexadecimal number system are indicated by suffix H (see Appendix A).

Typically, this bit is cleared for the real-mode operation and set for the protected-mode operation. Section 3.5 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, it is treated 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, the type can identify it as a read-only, read-write, and so on. For a code segment, the 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.

3.3.3 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors shown in Figure 3.8. 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 14. Both LDT and GDT can contain up to $2^{13} = 8192$ 8-bit descriptors. As shown in Figure 3.6, 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. The LDTR and GDTR registers are used for this purpose. These registers 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 to set up the segment descriptor tables.

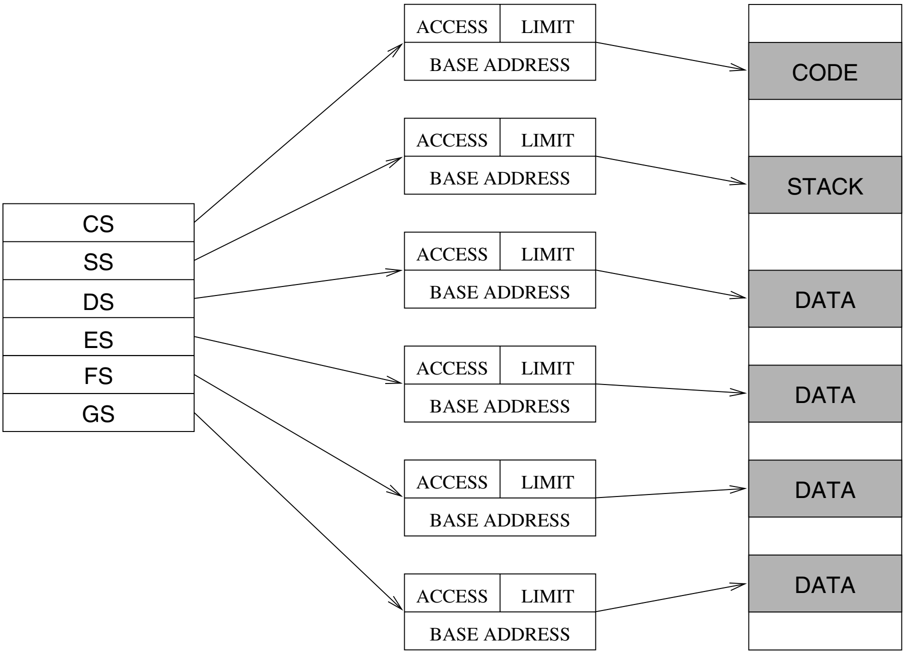


Figure 3.9 Segments in a multisegment model.

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 the GDT.

3.3.4 Segmentation Models

The Pentium segments can span the entire memory 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 and Linux.

Another model that uses the capabilities of segmentation to the full extent is the *multisegment model*. Figure 3.9 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

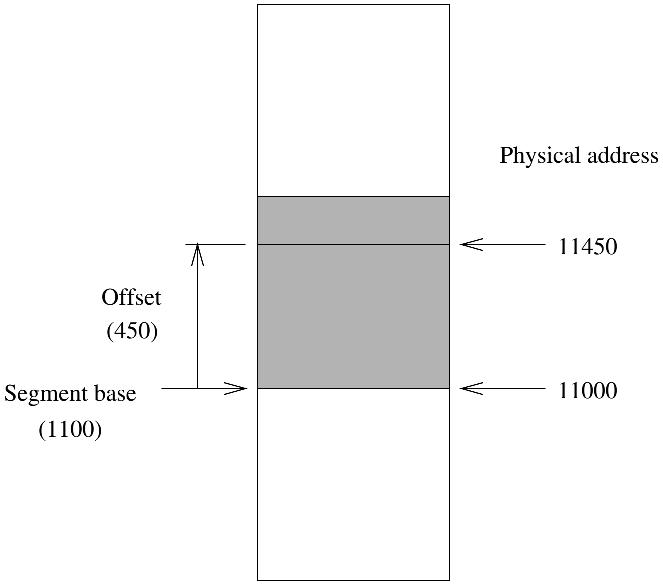


Figure 3.10 Relationship between logical and physical addresses of memory (all numbers are in hex).

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 3.7). The Pentium generates a general-protection exception if an attempt is made to access memory beyond the segment limit.

3.4 Real-Mode Memory Architecture

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.

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 3.10.

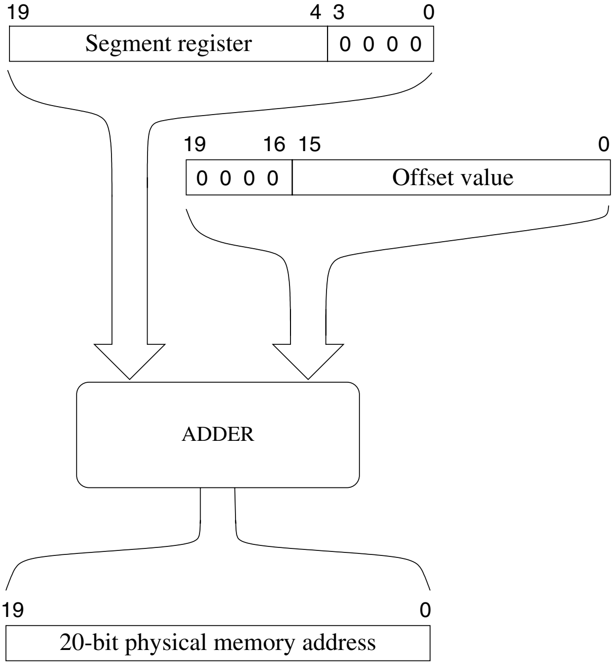


Figure 3.11 Physical address generation in the 8086.

Notice from Figure 3.10 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 3.10. The latter value to identify a memory location is referred to as the *physical memory address*.

Programmers have to be concerned with the logical addresses only. However, when the processor 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 3.11, 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

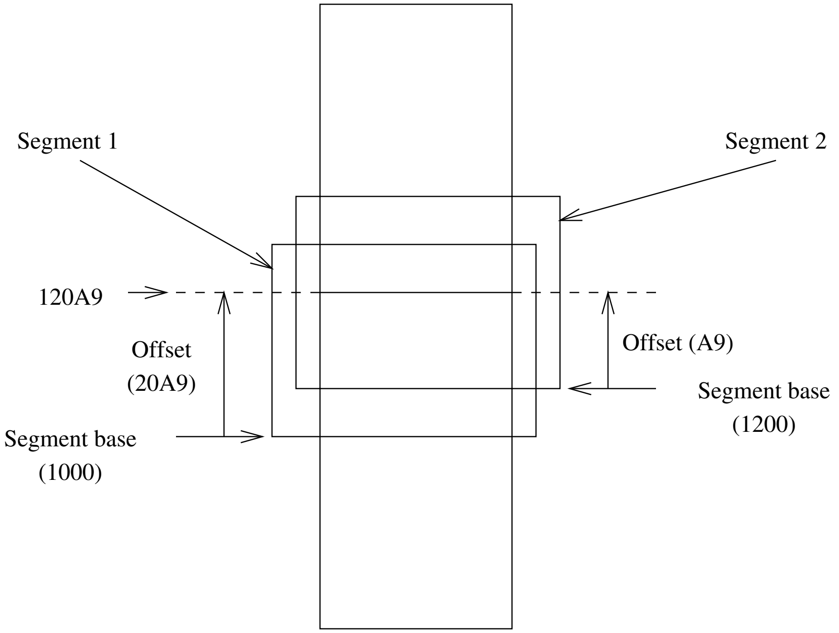


Figure 3.12 Two logical addresses map to the same physical address (all numbers are in hex).

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:

```
  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 3.12, 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 KB in size. In the real mode, the Pentium sets the size of each segment to exactly 64 KB. 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.

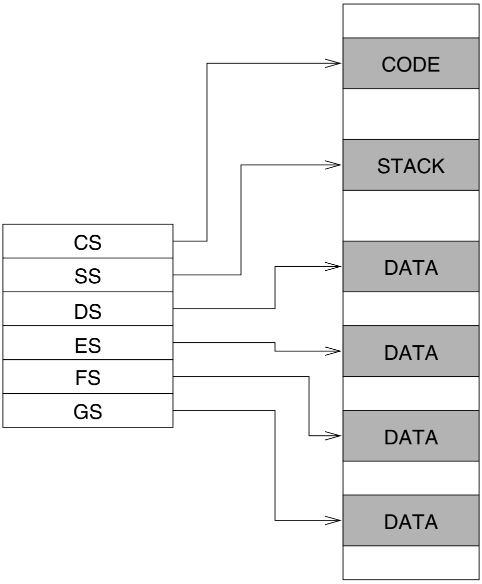


Figure 3.13 The six segments of the memory system.

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 3.13. 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 3.14.

3.5 Mixed-Mode Operation

Our previous discussion of protected and real 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 6.

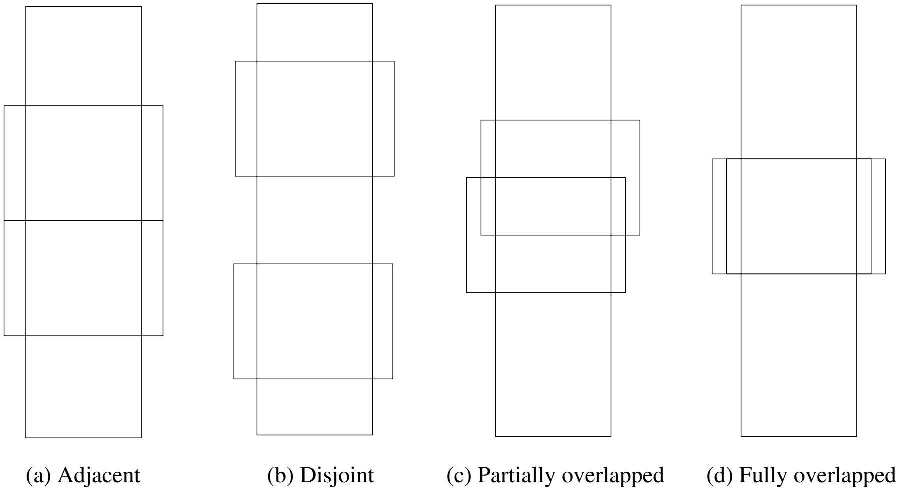


Figure 3.14 Various ways of placing segments in the memory.

3.6 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 5.

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 6.

3.7 Initial State

When the system is turned on, a built-in self-test (BIST) could be done to check the processor health. The result of this test is returned in the EAX register. EAX contains a zero if the processor has passed the test; otherwise, a nonzero value is returned.

The EIP is initialized to 0000FFF0H and code segment register is set to F000H. This segment descriptor contains the following information:

Base address = `FFFF0000H`
Segment limit = `FFFFH`

Since we know the values of the EIP and base address, we can find the address of the first instruction:

Base address:	`FFFF0000`
Offset (in EIP):	`FFF0`
First address:	`FFFFFFF0`

The first instruction executed must be located at 16 bytes below the highest address. This is where the system's EPROM/flash memory should be located to provide the initialization code.

All the other segment registers are cleared to use the segment selector 0. This segment selector contains the following information:

Base address = `00000000H`
Segment limit = `FFFFH`

As you can see from the base address values, all segments start at address 0. Note also that the segment size of all segments is limited to 64 KB as in the real mode. In fact, Pentium starts in the real mode. It maintains a control register (CR0) to facilitate mode switching. The least significant bit of this register can be used to switch to the protected mode. For example, we can switch to the protected mode using the following code:

```
; enter protected mode
mov     EAX,CR0     ; EAX = CR0
or      EAX,1       ; EAX = EAX OR 1
mov     CR0,EAX     ; CR0 = EAX
```

We discuss these instructions in the next chapter. Briefly, the first instruction copies the CR0 register value into the EAX register. The second instruction sets the least significant bit to 1, and the last instruction copies this modified value back to the CR0 register. We have to use this indirect method, as modifying the contents of CR0 is not allowed by Pentium.

Note that this code should not be used to switch the mode without setting up the appropriate tables such as global descriptor table, interrupt descriptor table, and so on.

After initialization, the global, local, and interrupt descriptor table registers, GDTR, LDTR and IDTR, are initialized as follows:

Base address = 00000000H
Segment limit = FFFFH

All the eight registers discussed in Sections 3.2.1 and 3.2.2 are initialized to zero with the exception of EAX and EDX. When BIST is selected, the EAX contains the result of the test. If no BIST is done, EAX is also initialized to zero. The EDX register is used to identify the processor (386, 486, Pentium).

3.8 Summary

We described the architecture of the Pentium processor. The Pentium can address up to 4 GB of memory. It provides protected- and real-mode memory architectures. 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.

In the real mode, the Pentium supports 16-bit addresses and the memory architecture of the 8086 processor. 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 pointer
- Linear address
- Logical address
- Mixed-mode operation
- Pentium control registers
- Pentium data registers
- Pentium flags register
- Pentium index registers
- Pentium pointer registers
- Pentium registers
- Physical address
- Protected-mode architecture
- Real-mode architecture
- Segment descriptor
- Segment descriptor tables
- Segment registers
- Segmented memory organization

3.9 Exercises

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

3–2 What are the three address spaces supported by the Pentium processor in the protected mode?

3–3 What is a segment? Why does the Pentium support segmented memory architecture?

3–4 Why is segment size limited to 64 KB in the real mode?

3–5 In the real mode, segments cannot be placed anywhere in memory. Explain the reason for this restriction.

3–6 In the real mode, can a segment begin at the following physical addresses?

(a) 1235AH	(b) 53535H
(c) 21700H	(d) ABCD0H

3–7 What is the maximum size of a segment in the protected mode?

3–8 We stated that the Pentium can access up to six segments at a time. What is the hardware reason for this limitation?

3–9 In the protected mode, segment size granularity can be either 1 byte or 4 KB. Explain the hardware reason for this restriction.

3–10 What is the purpose of the TI field in the segment descriptor?

3–11 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.

3–12 Describe the logical to physical address translation process in the real mode.

3–13 Describe the logical to linear address translation process in the protected mode.

3–14 Discuss the differences between the segmentation architectures supported in the real and protected modes.

3–15 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.

3–16 Convert the following logical addresses to physical addresses. All numbers are in hexadecimal. Assume the real mode.

(a) 1A2B:019A	(b) 3911:200
(c) 2591:10B5	(d) 1100:ABCD

Chapter 4

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 data transfer instructions of Pentium
- To provide an overview of the Pentium instruction set
- To examine how constants and macros are defined in the assembly language
- To demonstrate the performance benefits of the translation instruction

The objective of this chapter is to review the basics of the Pentium assembly language. Assembly language statements can either instruct the processor to perform a task, or direct the assembler during the assembly process. The latter statements are called assembler directives. Section 4.1 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 4.2. The instructions of the processor consist of an operation code to indicate the type of operation to be performed, and the specification of the data required (also called the addressing mode) by the operation. Section 4.3 describes some basic addressing modes supported by the Pentium.

The instruction set of the Pentium can be divided into several groups of instructions. Section 4.4 discusses the instructions that transfer data, including `mov`*,* `xchg`*, and* `xlat` *instructions. Section 4.5 provides an overview of some of the Pentium instructions belonging to the other groups. Later chapters discuss these instructions in more detail.*

Section 4.6 describes the assembler directives to define constants—numeric as well as string constants. NASM allows definition of macros with parameters. Macros provide a sophisticated text substitution mechanism, which is introduced in Section 4.7. Several examples are provided in Section 4.8. The performance advantage of the translation instruction `xlat` *is demonstrated in Section 4.9. The chapter concludes with a summary.*

4.1 Assembly Language Statements

Assembly language programs are created out of three different classes of statements. Statements in the first class tell the processor what to do. These statements 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. As stated in Chapter 1, 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 *pseudo-ops*. Assembler directives are nonexecutable and do not generate any machine language instructions.

The last class of statements, called *macros*, are 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 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. Macros are discussed in Section 4.7.

Assembly language statements are entered one per line in the source file. 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 common practice to align the fields to aid readability of assembly language programs. The assembler does not care about spaces between the fields.

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 to be done on the data stored in memory at a location identified by `result`. The following assembler directive defines a 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. Certain reserved words that have special meaning to the assembler are not allowed as labels. These include mnemonics such as `inc` and `EQU`.

The fields in a statement must be separated by at least one space or tab character. More spaces and tabs can be used at the programmer's discretion, but the assembler ignores them.

It is a 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 assembly language statement is written on two lines as

```
repeat:
     inc     result    ;increment result by 1
```

4.2 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;   /* allocates 1 byte  */
int        value;      /* allocates 4 bytes */
float      total;      /* allocates 4 bytes */
double     temp;       /* allocates 8 bytes */
```

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 use four bytes of storage. However, the bit pattern stored in them would be interpreted differently. For instance, the bit pattern (8FF08DB9H)

```
1000 1111 1111 0000 1000 1101 1011 1001
```

stored in the four bytes allocated for `value_1` is interpreted as representing $+2.4149 \times 10^9$, while the same bit pattern stored in `value_2` would be interpreted as -1.88006×10^9.

In the 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 (as in a C variable declaration) is attached to the contents of these bytes. It is entirely up to the program to interpret the bit pattern stored in the space reserved for data.

The general format of the storage allocation statement for initialized data 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.

```
sorted    DB    'y'
```

This statement allocates a single byte of storage and initializes it to `y`. Our assembly language program can refer to this character location by its name `sorted`. We 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. The following data definition statement allocates two bytes of contiguous storage and initializes it to 25159.

```
value    DW    25159
```

The decimal value 25159 is automatically converted to its 16-bit binary equivalent (6247H). Since Pentium uses little-endian byte ordering (see Chapter 2), 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
```

Range of Numeric Operands

The numeric operand of a define directive can take both signed and unsigned numbers. The valid range depends on the number of bytes allocated as shown in the following table:

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)

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
```

Uninitialized Data

To reserve space for uninitialized data, we use `RESB`, `RESW`, and so on. Each reserve directive takes a single operand that specifies the number of units of space (bytes, words, ...) to be reserved. There is a reserve directive for each define directive.

```
RESB    Reserve a Byte
RESW    Reserve a Word
RESD    Reserve a Doubleword
RESQ    Reserve a Quadword
REST    Reserve Ten Bytes
```

Here are some examples:

```
response   RESB    1
buffer     RESW    100
total      RESD    1
```

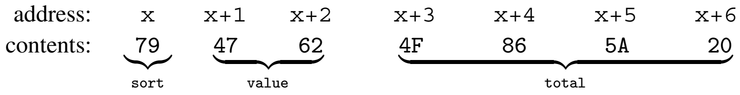
The first statement reserves a byte while the second reserves space for an array of 100 words. The last statement reserves space for a doubleword.

Multiple Definitions

Assembly language programs typically contain several data definition statements. For example, look at the following assembly language program fragment:

```
sort      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 for these variables. The memory layout for these 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, a `marks` array of size 8 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
```

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 your 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.

Multiple Initializations

In the previous example, if the class size is 90, it is inconvenient to define the array as described. The TIMES directive allows multiple initializations to the same value. Using TIMES, the marks array can be defined as

```
marks     TIMES  8  DW  0
```

The TIMES directive is useful in defining arrays and tables.

Symbol Table

When we allocate storage space using a data definition directive, we usually associate a symbolic name to refer to it. The assembler, during the assembly process, assigns an offset value for each symbolic name. For example, consider the following data definition statements:

```
.DATA
value     DW   0
sum       DD   0
marks     TIMES  10  DW  0
message   DB   'The grade is:',0
char1     DB   ?
```

As noted before, the assembler assigns contiguous memory space for the variables. The assembler also uses the same ordering of variables that is present in the source code. Then, finding the offset value of a variable is a simple matter of counting the number of bytes allocated to all the variables preceding it. For example, the offset value of marks is 6 because value and sum are allocated 2 and 4 bytes, respectively. The symbol table for the data segment is shown below:

Name	Offset
value	0
sum	2
marks	6
message	26
char1	40

4.3 Where Are the Operands?

Assembly language programs can be thought of as consisting of two logical parts: *data* and *code*. Most assembly language instructions require operands. There are several ways to specify the location of the operands. These are called the *addressing modes*. This section is a brief overview of some of the addressing modes required to do basic assembly language programming. A complete discussion is given in Chapter 6.

An operand required by an instruction may be in any one of the following locations:

- in a register internal to the processor;
- in the instruction itself;
- in main memory (usually in the data segment);
- at an I/O port (discussed in Chapter 15).

Specification of an operand that is in a register is called *register addressing mode*, while *immediate addressing mode* refers to specifying an operand that is part of the instruction. Several addressing modes are available to specify the location of an operand residing in memory. The motivation for providing these addressing modes comes from the perceived need to efficiently support high-level language constructs (see Chapter 6 for details).

4.3.1 Register Addressing Mode

In this addressing mode, processor's internal registers contain the data to be manipulated by the instruction. For example, the instruction

```
mov     EAX,EBX
```

requires two operands and both are in the processor registers. The syntax of the `mov` instruction is

```
mov     destination,source
```

The `mov` instruction copies contents of `source` to `destination`. The contents of `source` are not destroyed. Thus,

```
mov     EAX,EBX
```

copies the contents of the EBX register into the EAX register. Note that the original contents of EAX are lost. In this example, the `mov` instruction is operating on 32-bit data. However, it can also work on 16- and 8-bit data, as shown below:

```
mov     BX,CX
mov     AL,CL
```

Register-addressing mode is the most efficient way of specifying data because the data are within the processor and, therefore, no memory access is required.

4.3.2 Immediate Addressing Mode

In this addressing mode, data are specified as part of the instruction itself. As a result, even though the data are in memory, it is located in the code segment, not in the data segment. This addressing mode is typically used in instructions that require at least two data items to manipulate. In this case, this mode can only specify the source operand. In addition, the immediate data are always a constant, given either directly or via the EQU directive (discussed in Section 4.6). Thus, instructions typically use another addressing mode to specify the destination operand. In the following 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.

The remainder of the addressing modes we discuss here deal with operands that are located in the data segment. These are called the *memory addressing modes*. We discuss two memory-addressing modes here: *direct* and *indirect*. The remaining memory-addressing modes are discussed in Chapter 6.

4.3.3 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 two previous addressing modes.

Recall that to locate a data item in the data segment, we need two components: the segment start address and an offset value within the segment. The start address of the segment is typically found in the DS register. Thus, various memory-addressing modes differ in the way the offset value of the data is specified. The offset value is often called the *effective address*.

In the direct addressing mode, the offset value is specified directly as part of the instruction. In an assembly language program, this value is usually indicated by the variable name of the data item. The assembler will translate this name into its associated offset value during the assembly process. To facilitate this translation, the assembler maintains a symbol table. As discussed before, the symbol table stores the offset values of all variables in the assembly language program.

This addressing mode is the simplest of all the memory-addressing modes. A restriction associated with the memory-addressing modes is that these can be used to specify only one operand. The examples that follow assume the following data definition statements in the program.

```
response  DB     'Y'         ; allocates a byte, initializes to Y
table1    TIMES 20 DD 0      ; allocates 80 bytes, initializes to 0
name1     DB     'Jim Ray'   ; 7 bytes are initialized to Jim Ray
```

Here are some examples of the mov instruction:

```
mov    AL,[response]    ; copies Y into AL register
mov    [response],'N'   ; N is written into response
mov    [name1],'K'      ; write K as the first character of name1
mov    [table1],56      ; 56 is written in the first element
```

This last statement is equivalent to `table1[0] = 56` in C.

In NASM, we write

```
mov    EBX,table1
```

to copy the *address* of `table1` into the EBX register. If we want the *value*, we should use [ ] as in the previous examples. For example, the statement

```
mov    EBX,[table1]
```

copies the first element of `table1` into EBX. This notation is different from the TASM/ MASM notation.

4.3.4 Indirect Addressing Mode

The direct addressing mode can be used in a straightforward way but is limited to accessing simple variables. For example, it is not useful in accessing the second element of `table1` as in the following C statement:

```
table1[1] = 99
```

The indirect addressing mode remedies this deficiency. In this addressing mode, the offset or effective address of the data is in one of the general registers. For this reason, this addressing mode is sometimes referred to as 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 like `table1` containing several elements, the starting address of the data structure can be loaded into, say, the EBX register and then EBX acts as a pointer to an element in `table1`. By manipulating the contents of the EBX register, we can access different elements of `table1`.

The following code assigns 100 to the first element and 99 to the second element of `table1`. Note that EBX is incremented by 4 because each element of `table1` requires four bytes.

```
mov     EBX,table1     ; copy address of table1 to EBX
mov     [EBX],100      ; table1[0] = 100
add     EBX,4          ; EBX = EBX + 4
mov     [EBX],99       ; table1[1] = 99
```

Chapter 6 discusses other memory-addressing modes that can perform this task more efficiently.

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
```

Thus,

```
lea     EBX,[table1]
```

can be used in place of the

```
mov     EBX,table1
```

instruction. The difference is that `lea` computes the offset values at run time, whereas the `mov` version 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     EBX,[array+ESI]
```

to load EBX with the address of an element of `array` whose index is in the ESI register. However, we cannot write

```
mov     EBX,[array+ESI]      ; illegal
```

as the contents of ESI are known at assembly time.

4.4 Data Transfer Instructions

We now discuss some of the data transfer instructions supported by Pentium. Specifically, we describe the `mov`, `xchg`, and `xlat` instructions. Other data transfer instructions such as `movsx` and `movzx` are discussed in Chapter 7.

4.4.1 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
mov     register,immediate
mov     memory,immediate
mov     register,memory
mov     memory,register
```

There is no move instruction to transfer data from memory to memory, as the Pentium processor does not allow it. However, as we will see in Chapter 10, memory-to-memory data transfer is possible using the string instructions.

Here are some example `mov` statements:

```
mov     [response],BH
mov     EDX,[table1]
mov     [name1+4],'K'
```

4.4.2 Ambiguous Moves

Moving an immediate value into memory sometimes causes ambiguity as to the type of operand. For example, in the statements

```
mov    EBX,table1
mov    ESI,name1
mov    [EBX],100
mov    [ESI],100
```

it is not clear whether a word (2 bytes) or a byte equivalent of 100 is to be written in the memory. We can clarify this ambiguity by using a type specifier. For example, we can use WORD type specifier to identify a word operation and BYTE for a byte operation. Using the type specifiers, we can write

```
mov    WORD [EBX],100
mov    BYTE [ESI],100
```

We can also write these statements as

```
mov    [EBX],WORD 100
mov    [ESI],BYTE 100
```

Some of the type specifiers available are given below:

Type specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

4.4.3 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. Some examples are

```
xchg    EAX,EDX
xchg    [response],CL
xchg    [total],DX
```

As in the mov instruction, both operands cannot be located in memory. Note that this restriction is applicable to most instructions. 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`. This instruction is especially useful in sorting applications. It is also useful to swap the two bytes of 16-bit data to perform conversions between little-endian and big-endian forms. The following example

```
xchg    AL,AH
```

converts the value in AX to the other endian form. Pentium provides the `bswap` instruction to perform such conversions on 32-bit data. The format is

```
bswap  register
```

This instruction works only on the data located in a 32-bit register.

4.4.4 The XLAT Instruction

The `xlat` (translate) instruction can be used to perform character translation. The format of this instruction is shown below:

```
xlatb
```

To use this instruction, the EBX register must to 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 AL to EBX 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 4.8.

4.5 Overview of Assembly Language Instructions

This section briefly reviews some of the remaining assembly language instructions. The discussion presented here would provide sufficient exposure to the assembly language so that you can write meaningful assembly language programs.

4.5.1 Simple Arithmetic Instructions

The Pentium provides several instructions to perform simple arithmetic operations. In this section, we describe five instructions to perform addition and subtraction. We defer a full discussion until Chapter 7.

The INC and DEC Instructions

These instructions can be used to either increment or decrement the operands by one. The `inc` (INCrement) instruction adds one to its operand and the `dec` (DECrement) instruction subtracts one from its operand. Both instructions require 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    EBX        ; increment 32-bit register
dec    DL         ; decrement 8-bit register
```

Let us assume that EBX and DL have 1057H and 5AH, respectively. After executing the above two instructions, EBX and DL would have 1058H and 59H, respectively. If the initial values of EBX and DL are FFFFH and 00H, after executing the two statements the contents of EBX and DL are changed to 0000H and FFH, respectively.

As another example, consider the following program:

```
.DATA
count     DW    0
value     DB    25

.CODE
      inc     [count]        ;unambiguous
      dec     [value]        ;unambiguous
      mov     EBX,count
      inc     [EBX]          ;ambiguous
      mov     ESI,value
      dec     [ESI]          ;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    [EBX]
dec    [ESI]
```

are ambiguous because EBX and ESI merely point to an object in memory but the actual object type (whether a WORD or BYTE) is not clear. We have to use a type specifier to clarify this ambiguity, as shown below:

```
inc    WORD [EBX]
dec    BYTE [ESI]
```

Table 4.1 Some Examples of the `add` Instruction

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`

The ADD Instruction

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
```

Some examples of `add` instruction are givn in Table 4.1. In general,

```
inc     EAX
```

is preferred to

```
add     EAX,1
```

as the `inc` version improves readability and requires less memory space to store the instruction. However, both instructions execute at the same speed.

The SUB and CMP 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
```

Table 4.2 gives examples of the `sub` instruction.

The `cmp` (CoMPare) instruction is used to compare two operands (equal, not equal, and so on). The `cmp` instruction performs the same operation as the `sub` instruction except that

Table 4.2 Some Examples of the `sub` Instruction

	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 result of subtraction is not saved. Thus, `cmp` does not disturb the source and destination operands. While both `sub` and `cmp` instructions take the same number of clocks in most cases, `cmp` requires one less if the destination is memory. This is because the `cmp` instruction does not write the result in memory, whereas the `sub` instruction does.

The `cmp` instruction is typically used in conjunction with a conditional jump instruction for decision making. This is the topic of the next section.

4.5.2 Conditional Execution

The Pentium instruction set has several branching and looping instructions to construct programs that require conditional execution. In this section, we discuss a subset of these instructions. A detailed discussion is in Chapter 8.

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. The following example

```
        mov     EAX,1
inc_again:
        inc     EAX
        jmp     inc_again
        mov     EBX,EAX
           . . .
```

results in an infinite loop incrementing EAX repeatedly. The instruction

```
mov    EBX,EAX
```

and all the instructions following it are never executed!

From this example, the `jmp` instruction appears to be useless. Later, we show some examples that illustrate the use of this instruction.

Conditional Jump

In conditional jump instructions, program execution is transferred to the target instruction only if 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 or logic operation. For example, the following 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 previous `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. While 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 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, at the time of the `je` instruction execution, the processor checks the ZF and program execution jumps to the target instruction if and only if ZF = 1. To cause the jump, Pentium loads the EIP register with the target instruction address. Recall that the EIP register always points to the next instruction to be executed. Therefore, when the input character 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 the conditional jump instructions:

je	jump if equal
jg	jump if greater
jl	jump if less
jge	jump if greater than 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

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)

Example 4.1 *Conditional jump examples.*
Consider the following code.

```
go_back:
      inc     AL
        . . .
        . . .
      cmp     AL,BL
      statement_1
      mov     BL,77H
```

Table 4.3 shows the actions taken depending on `statement_1`. □

These conditional jump instructions assume that the operands compared were treated 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 8, these six conditional jump instructions are sufficient for writing simple assembly language programs.

When you use these conditional jump instructions, sometimes your assembler complains that the destination of the jump is “out of range.” If you find yourself in this situation, you can use the trick described in Section 8.3.4 on page 249.

Table 4.3 Conditional Jump Examples

`statement_1`	AL	BL	Action taken
`je   go_back`	56H	56H	Program control transferred to `inc   AL`
`jg   go_back`	56H	55H	Program control transferred to `inc   AL`

`jg   go_back`
`jl   go_back`
	56H	56H	No jump; executes `mov   BL,77H`

`jle  go_back`
`jge  go_back`
	56H	56H	Program control transferred to `inc   AL`

`jne  go_back`
`jg   go_back`
`jge  go_back`
	27H	26H	Program control transferred to `inc   AL`

4.5.3 Iteration Instruction

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 if CL is not 0
            . . .
            . . .
```

The Pentium instruction set, however, includes a group of `loop` instructions to support iteration. Here we describe the basic `loop` instruction. The syntax of this instruction is

```
loop    target
```

where `target` is a label that identifies the target instruction as in the jump instructions.

This instruction assumes that the ECX register contains the loop count. As part of executing the `loop` instruction, it decrements the ECX register and jumps to the `target` instruction if ECX $\neq 0$. Using this instruction, we can write the previous example as

```
        mov     ECX,50
repeat1:
        <loop body>
        loop    repeat1
            . . .
            . . .
```

4.5.4 Logical Instructions

The Pentium instruction set provides several logical instructions including `and`, `or`, `xor`, and `not`. The syntax of these instructions is

```
and    destination,source
or     destination,source
xor    destination,source
not    destination
```

The first three are binary operators and perform bitwise `and`, `or`, and `xor` logical operations, respectively. The `not` is a unary operator that performs bitwise complement operation. Truth tables for the logical operations are shown in Table 4.4. Some examples that explain the operation of these logical instructions are shown in Table 4.5. In this table, all numbers are expressed in binary.

Logical instructions set some of the flags and therefore can be used in conditional jump instructions to implement high-level language decision structures in the assembly language. Until we fully discuss the flags in Chapter 7, the following usage should be sufficient to write and understand the assembly language programs.

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
```

would make the result 00H and is stored in the AL register. At the same time, the logical operation 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
```

Table 4.4 Truth Tables for the Logical Operations

`and` Operation

Input bits		Output bit
Source b_i	Destination b_i	Destination b_i
0	0	0
0	1	0
1	0	0
1	1	1

`or` Operation

Input bits		Output bit
Source b_i	Destination b_i	Destination b_i
0	0	0
0	1	1
1	0	1
1	1	1

`xor` Operation

Input bits		Output bit
Source b_i	Destination b_i	Destination b_i
0	0	0
0	1	1
1	0	1
1	1	0

can replace the

```
je    bit_is_zero
```

instruction. The conditional jump `je` is an alias for `jz`.

A problem with using the `and` instruction for testing, as used in the previous 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, the Pentium provides a `test` instruction. The syntax is

Table 4.5 Logical Instruction Examples

		`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`

```
test    destination,source
```

The `test` instruction performs logical bitwise **and** operation like the `and` instruction except that the source and destination operands are not modified. However, `test` sets the flags just like the `and` instruction. Therefore, we can use

```
test    AL,01H
```

instead of

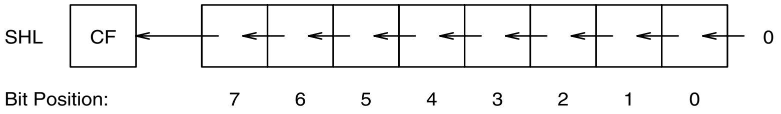
```
and    AL,01H
```

in the last example. Like the `cmp` instruction, `test` takes one clock less to execute than `and` if the destination operand is in memory.

4.5.5 Shift Instructions

The Pentium instruction set includes several shift instructions. We discuss the following two instructions here: `shl` (SHift Left) and `shr` (SHift Right).

The `shl` instruction can be used to left-shift a destination operand. Each shift to the left by one bit position causes the leftmost bit to move to the carry flag (CF). The vacated rightmost bit is filled with a zero. The bit that was in CF is lost as a result of this operation.



The `shr` instruction works similarly but shifts bits to the right as shown below:

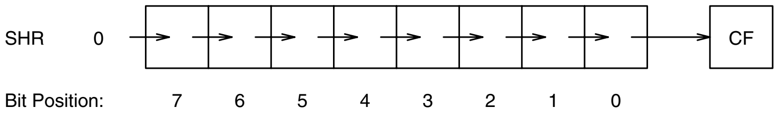


Table 4.6 Shift Instruction Examples

	Before shift	After shift	
Instruction	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

The general formats of these instructions are

```
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 specifies the shift count directly. The shift `count` 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 either the `shl` or `shr` 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), or 15 (for a 16-bit operand), or 31 (for a 32-bit operand). As indicated, Pentium does not allow the specification of shift count to be greater than 31. If a greater value is specified, Pentium takes only the least significant 5 bits of the number as the shift count. Table 4.6 shows some examples of the `shl` and `shr` instructions.

The following code shows another way of testing the least significant bit of the data in the AL register.

```
            . . .
      shr    AL,1
      jnc    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>
```

If the value in the AL register has a 1 in the least significant bit position, this bit will be in the carry flag after the `shr` instruction has been executed. We can then use a conditional jump instruction that tests the carry flag. Recall that `jc` (jump if carry) would cause the jump if CF = 1 and `jnc` (jump if no carry) causes jump only if CF = 0.

4.5.6 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, or through the carry flag. We will briefly discuss these two types of instructions next.

Rotate Without Carry

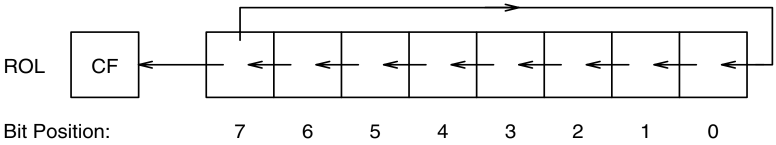
There are two instructions in this group:

`rol` (ROtate Left)
`ror` (ROtate Right)

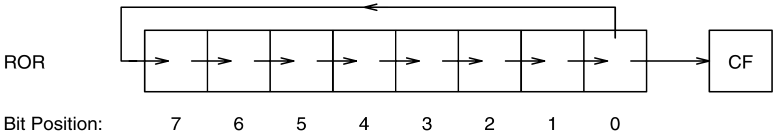
The format of these instructions is similar to the shift instructions and is given below:

```
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:



For both of these instructions, the CF catches the last bit rotated out of `destination`. The examples in Table 4.7 illustrate the rotate operation.

Rotate Through Carry

The instructions

Table 4.7 Rotate Examples

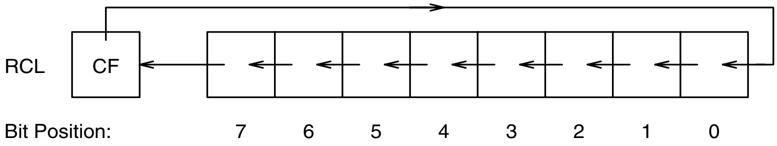
	Before execution	After execution	
Instruction	AL or AX	AL or AX	CF
`rol AL,1`	1010 1110	0101 1101	1
`ror AL,1`	1010 1110	0101 0111	0

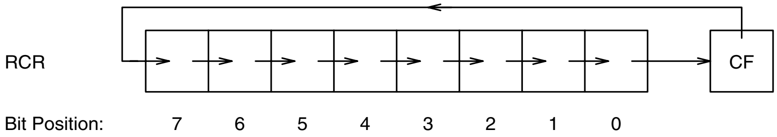
`mov CL,3`
`rol AL,CL`
	0110 1101	0110 1011	1

`mov CL,5`
`ror AX,CL`
	1011 1101 0101 1001	1100 1101 1110 1010	1

`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 in Table 4.8.

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. As an example, suppose 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 into the carry flag. The `rcr` instruction copies this carry flag value into the most significant bit of EAX. Chapter 9 introduces two doubleshift instructions to facilitate shifting of 64-bit numbers.

Table 4.8 Rotate Through Carry Examples

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

4.6 Defining Constants

NASM provides several directives to define constants. In this section, we discuss three directives—`EQU`, `%assign`, and `%define`.

4.6.1 The EQU Directive

The syntax of the EQU directive is

```
name     EQU     expression
```

which assigns the result of the `expression` to `name`. 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. Then, we can write

```
             . . .
mov     ECX,NUM_OF_STUDENTS
             . . .
cmp     EAX,NUM_OF_STUDENTS
             . . .
```

to move 90 into the ECX register and to compare EAX with 90, respectively. Defining constants this way has two advantages:

1. Such definitions increase program readability. This can be seen by comparing the statement

   ```
   mov     ECX,NUM_OF_STUDENTS
   ```

   with

```
        mov     ECX,90
```

The first statement clearly indicates that we are moving the class size into the ECX 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 just need to change the value in the EQU statement. If we didn't use the EQU directive, we 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.

The symbols that have been assigned a value cannot be reassigned another value in a given source module. If such redefinitions are required, you should use %assign directive, which is discussed next.

4.6.2 The %assign Directive

This directive can be used to define numeric constants like the EQU directive. However, %assign allows redefinition. For example, we define

```
%assign   i   j+1
```

and later in the code we can redefine it as

```
%assign   i   j+2
```

Like the EQU directive, it is evaluated once when %assign is processed.

The %assign is case-sensitive. That is, i and I are treated as different. We can use %iassign for case-insensitive definition.

Both EQU and %assign directives can be used to define numeric constants. The next directive removes this restriction.

4.6.3 The %define Directive

This directive is like the #define in C. It can be used to define numeric as well as string constants. For example,

```
%define   X1   [EBP+4]
```

replaces X1 by [EBP+4]. Like the last directive, it allows redefinition. For example, we can redefine X1 as

```
%define   X1   [EBP+20]
```

The %define directive is case-sensitive. If you want the case-insensitive version, you should use %idefine.

4.7 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 NASM, macros can be defined with %macro and %endmacro directives. The macro text begins with the %macro directive and ends with the %endmacro directive. The macro definition syntax is

```
%macro    macro_name   para_count
          <macro body>
%endmacro
```

The para_count specifies the number parameters used in the macro. 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.

Example 4.2 *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.

```
%macro  multEAX_by_16
        sal    EAX,4
%endmacro
```

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 multEAX_by_16, as in the following example:

```
      . . .
mov    EAX,27
multEAX_by_16
      . . .
```

When the assembler encounters the macro name multEAX_by_16, it is replaced (i.e., text-substituted) by the macro body. Thus, after the macro expansion, the assembler finds the code

```
        . . .
mov     EAX,27
sal     EAX,4
        . . .
```

□

Macros with Parameters

Just as with procedures, using parameters with macros helps us in writing more flexible and useful macros. The previous macro always multiplies EAX 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

```
%macro    mult_by_16  1
          sal    %1,4
%endmacro
```

This macro takes one parameter, which can be any operand that is valid in the `sal` instruction. Within the macro body, we refer to the parameters by their number as in `%1`. 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 defined it. These are referred to as *macroinstructions*.

Example 4.3 *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, `mxchg` to exchange doublewords of data in memory.

```
%macro   mxchg  2
         xchg   EAX,%1
         xchg   EAX,%2
         xchg   EAX,%1
%endmacro
```

For example, when this macro is invoked as

```
mxchg    value1,value2
```

it exchanges the memory words `value1` and `value2` while leaving EAX unaltered. □

To end this section, we give a couple of examples from the `io.mac` file.

Example 4.4 `PutInt` *macro definition from* `io.mac` *file.*
This macro is used to display a 16-bit integer, which is given as the argument to the macro, by calling the `proc_PutInt` procedure. The macro definition is shown below:

```
%macro  PutInt  1
        push    AX
        mov     AX,%1
        call    proc_PutInt
        pop     AX
%endmacro
```

The `PutInt` procedure expects the integer to be in AX. Thus, in the macro body, we move the input integer to AX before calling the procedure. Note that by using the `push` and `pop`, we preserve the AX register. □

Example 4.5 `GetStr` *macro definition from* `io.mac` *file.*
This macro takes one or two parameters: a pointer to a buffer and an optional buffer length. The input string is read into the buffer. If the buffer length is given, it will read a string that is one less than the buffer length (one byte is reserved for the NULL character). If the buffer length is not specified, a default value of 81 is assumed. This macro calls `proc_GetStr` procedure to read the string. This procedure expects the buffer pointer in EDI and buffer length in ESI register. The macro definition is given below:

```
%macro  GetStr  1-2 81
        push    ESI
        push    EDI
        mov     EDI,%1
        mov     ESI,%2
        call    proc_GetStr
        pop     EDI
        pop     ESI
%endmacro
```

This macro is different from the previous one in that the number of parameters can be between 1 and 2. This condition is indicated by specifying the range of parameters (1–2 in our example). A further complication is that, if the second parameter is not specified, we have to use the default value (81 in our example). As shown in our example, we include this default value in the macro definition. Note that this default value is used only if the buffer length is not specified. □

Our coverage of macros is a small sample of what is available in NASM. You should refer to the latest version of the NASM manual for complete details on macros.

4.8 Illustrative Examples

This section presents several examples that 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, the number 5 is stored as

```
00000101B
```

On the other hand, the 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 A.

Another prerequisite: Before looking at the examples of this section, you should read the material presented in Appendix B. This appendix gives details on the structure of the standalone assembly language program. In addition, it also explains how to assemble and link the assembly language programs to generate the executable file. You would also benefit from reading Appendix C, which gives details on debugging the assembly language programs.

Example 4.6 *ASCII to binary conversion.*

The goal of this example is to illustrate how the logical `test` instruction can be used to test a bit. The program reads a key from the keyboard and displays its ASCII code in binary. It then queries the user as to whether he or she wants to quit. Depending on the response, the program either requests another character input from the keyboard, 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). The `mask` is initialized to 80H (=10000000B), which tests only the value of the most significant bit of the ASCII value. If this bit is 0, the code

```
test    AL,mask
```

sets the ZF (assuming that the ASCII value is in the AL register). In this case, a 0 is displayed by directing the program flow using the `jz` instruction (line 29). Otherwise, a 1 is displayed.

The mask is then divided by 2, 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 {ECX 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 4.1, follows the pseudo-code in a straightforward way. Note that Pentium provides an instruction to perform integer division. However, to divide a number by 2, shr is much faster than the divide instruction. More details about the division instructions are given in Chapter 7.

Program 4.1 Conversion of ASCII to binary representation

```
;Binary equivalent of characters   BINCHAR.ASM
;
```

```
;        Objective: To print the binary equivalent of
;                   ASCII character code.
;            Input: Requests a character from keyboard.
;           Output: Prints the ASCII code of the
;                   input character in binary.
%include "io.mac"

.DATA
char_prompt    db  "Please input a character: ",0
out_msg1       db  "The ASCII code of ’",0
out_msg2       db  "’ in binary is ",0
query_msg      db  "Do you want to quit (Y/N): ",0

.CODE
        .STARTUP
read_char:
        PutStr  char_prompt  ; request a char. input
        GetCh   AL           ; read input character

        PutStr  out_msg1
        PutCh   AL
        PutStr  out_msg2
        mov     AH,80H       ; mask byte = 80H
        mov     ECX,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
        cmp     AL,’Y’       ; if response is not ’Y’
        jne     read_char    ; read another character
done:                        ; otherwise, terminate program
        .EXIT
```

Example 4.7 *ASCII to hexadecimal conversion using character manipulation.*
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.

$$\text{L} \xrightarrow{\text{ASCII}} \text{01001100B} \xrightarrow{\substack{\text{mask off}\\ \text{upper half}}} \text{00001100B} \xrightarrow{\substack{\text{convert}\\ \text{to hex}}} \text{C}$$

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:

$$\text{L} \xrightarrow{\text{ASCII}} \text{01001100B} \xrightarrow{\substack{\text{mask off}\\ \text{lower half}}} \text{01000000B} \xrightarrow{\substack{\text{shift right}\\ \text{4 positions}}} \text{00000100B} \xrightarrow{\substack{\text{convert}\\ \text{to hex}}} 4$$

Notice that shifting right by four bit positions is equivalent to performing integer division by 16. The pseudocode of the program shown in Program 4.2 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 or not. The conversion using character manipulation can be summarized as follows:

```
if (number ≤ 9)
then
    write (number + '0')
then
    write (number + 'A' − 10)
end if
```

If the number is between 0 and 9, we add the ASCII value for character 0 to convert the number to its character equivalent. 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 4.2 by

```
add    AL,'0'
```

on line 31. 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 for character A
```

In Program 4.2, this is done by

```
add    AL,'A'-10
```

on line 34.

Program 4.2 Conversion to hexadecimal by character manipulation

```
 1:  ;Hex equivalent of characters   HEX1CHAR.ASM
 2:  ;
 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:  %include "io.mac"
 9:
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
```

```

.CODE
        .STARTUP
read_char:
        PutStr  char_prompt  ; request a char. input
        GetCh   AL           ; read input character

        PutStr  out_msg1
        PutCh   AL
        PutStr  out_msg2
        mov     AH,AL        ; save input character in AH
        shr     AL,4         ; move upper 4 bits to lower half
        mov     CX,2         ; loop count - 2 hex digits to print
print_digit:
        cmp     AL,9         ; if greater than 9
        jg      A_to_F       ; convert to A through F digits
        add     AL,'0'       ; otherwise, convert to 0 through 9
        jmp     skip
A_to_F:
        add     AL,'A'-10    ; subtract 10 and add 'A'
                             ;  to convert to A through F
skip:
        PutCh   AL           ; write the first hex digit
        mov     AL,AH        ; restore input character in AL
        and     AL,0FH       ; mask off the upper half-byte
        loop    print_digit
        nwln
        PutStr  query_msg    ; query user whether to terminate
        GetCh   AL           ; read response

        cmp     AL,'Y'       ; if response is not 'Y'
        jne     read_char    ; read another character
done:                        ; otherwise, terminate program
        .EXIT
```

Example 4.8 *ASCII to hexadecimal conversion 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 a number between 0 and 15 to its equivalent hex digit. The program is shown in Program 4.3. 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 number as an index into the table. For example, index value of 10 points to A, which is the equivalent hex digit. In order to use the xlat instruction, the EBX register should point to the base of the hex_table and AL should have the number. The rest of the program is straightforward to follow.

Program 4.3 Conversion to hexadecimal by using the xlat instruction

```
;Hex equivalent of characters   HEX2CHAR.ASM
;
;        Objective: To print the hex equivalent of
;                   ASCII character code. Demonstrates
;                   the use of xlat instruction.
;            Input: Requests a character from keyboard.
;           Output: Prints the ASCII code of the
;                   input character in hex.
%include "io.mac"

.DATA
char_prompt    db  "Please input a character: ",0
out_msg1       db  "The ASCII code of ’",0
out_msg2       db  "’ in hex is ",0
query_msg      db  "Do you want to quit (Y/N): ",0
; translation table: 4-bit binary to hex
hex_table      db  "0123456789ABCDEF"

.CODE
        .STARTUP
read_char:
        PutStr  char_prompt  ; request a char. input
        GetCh   AL           ; read input character

        PutStr  out_msg1
        PutCh   AL
        PutStr  out_msg2
        mov     AH,AL        ; save input character in AH
        mov     EBX,hex_table ; EBX = translation table
        shr     AL,4         ; move upper 4 bits to lower half
        xlatb                ; replace AL with hex digit
        PutCh   AL           ; write the first hex digit
        mov     AL,AH        ; restore input character to AL
        and     AL,0FH       ; mask off upper 4 bits
        xlatb
        PutCh   AL           ; write the second hex digit
        nwln
```

```
38:         PutStr  query_msg    ; query user whether to terminate
39:         GetCh   AL           ; read response
40:
41:         cmp     AL,'Y'       ; if response is not 'Y'
42:         jne     read_char    ; read another character
43:  done:                       ; otherwise, terminate program
44:         .EXIT
```

Example 4.9 *Conversion of lowercase letters to uppercase.*
This program 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 4.4 is shown below:

```
main()
    display prompt message
    read input string
    index := 0
    char := string[index]
    while (char $\neq$ NULL)
        if ((char $\geq$ 'a') AND (char $\leq$ '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 4.4 that the compound **if** condition requires two cmp instructions (lines 27 and 29). Also, the program uses the EBX register in indirect addressing mode and always holds the pointer value of the character to be processed. In Chapter 6 we will see a better way of accessing the 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 4.4 Conversion to uppercase by character manipulation

```
;Uppercase conversion of characters   TOUPPER.ASM
;
;        Objective: To convert lowercase letters to
;                   corresponding uppercase letters.
;            Input: Requests a character string from keyboard.
;           Output: Prints the input string in uppercase.
%include "io.mac"

.DATA
name_prompt    db  "Please type your name: ",0
out_msg        db  "Your name in capitals is: ",0

.UDATA
in_name        resb  31

.CODE
        .STARTUP
        PutStr  name_prompt  ; request character string
        GetStr  in_name,31   ; read input character string

        PutStr  out_msg
        mov     EBX,in_name  ; EBX = pointer to in_name
process_char:
        mov     AL,[EBX]     ; move the char. to AL
        cmp     AL,0         ; if it is the NULL character
        je      done         ;  conversion done
        cmp     AL,'a'         ; if (char < 'a')
        jl      not_lower_case ; not a lowercase letter
        cmp     AL,'z'         ; if (char > 'z')
        jg      not_lower_case ; not a lowercase letter
lower_case:
        add     AL,'A'-'a'   ; convert to uppercase
not_lower_case:
        PutCh   AL           ; write the character
        inc     EBX          ; EBX points to the next char.
        jmp     process_char ; go back to process next char.
done:
 nwln
        .EXIT
```

Example 4.10 *Sum of the individual digits of a number.*
This last example shows how decimal digits can be converted from their character representations to the binary equivalent. 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 digit characters, it is straightforward to get their numerical value (see our discussion in Appendix A). All we have to do is to mask off the upper half of the byte, as is done in Program 4.5 (line 28) by

```
and    AL,0FH
```

Alternatively, we could also subtract the character code for 0

```
sub    AL,'0'
```

instead of masking the upper half byte. For the sake of brevity, we leave writing the pseudocode of Program 4.5 as an exercise.

Program 4.5 Sum of 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:  %include  "io.mac"
 9:
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:
14:  .UDATA
15:  number         resb  11
16:
17:  .CODE
18:          .STARTUP
19:          PutStr  number_prompt  ; request an input number
20:          GetStr  number,11     ; read input number as a string
21:
22:          mov     EBX,number    ; EBX = address of number
23:          sub     DX,DX         ; DX = 0 -- DL keeps the sum
24:  repeat_add:
25:          mov     AL,[EBX]      ; move the digit to AL
```

```
        cmp     AL,0        ; if it is the NULL character
        je      done        ;  sum is done
        and     AL,0FH      ; mask off the upper 4 bits
        add     DL,AL       ; add the digit to sum
        inc     EBX         ; update EBX to point to next digit
        jmp     repeat_add
done:
        PutStr  out_msg
        PutInt  DX          ; write sum
        nwln
        .EXIT
```

4.9 Performance: When to Use XLAT Instruction

The xlat instruction is convenient to perform character conversions. Proper use of xlat would produce an efficient assembly language program. In this section, we demonstrate by means of two examples when xlat is beneficial from the performance point of view.

In general, xlat is not really useful if, for example, there is a straightforward method or a "formula" for the required conversion. This is true for conversions that exhibit a regular structure. An example of this type of conversion is the case conversion between uppercase and lowercase letters in ASCII. As you know, the ASCII encoding makes this conversion rather simple. Experiment 1 takes a look at this type of example.

The use of the xlat instruction, however, produces efficient code if the conversion does not have a regular structure. Conversion from EBCDIC to ASCII is one example that can benefit from using the xlat instruction. Conversion to hex is another example, as shown in Examples 4.7 and 4.8. This example is used in Experiment 2 to show the performance benefit that can be obtained from using the xlat instruction for the conversion.

4.9.1 Experiment 1

In this experiment, we show how using the xlat instruction for case conversion of letters deteriorates the performance. We have transformed the code of Example 4.9 to a procedure that can be called from a C main program that keeps track of the time (see Chapter 1 for details about the C main program). All interaction with the display is suppressed for these experiments. This case conversion procedure is called several times to convert a string of lowercase letters. The string length is fixed at 1000 characters.

We used two versions of the case conversion procedure. The first version does not use the xlat instruction for case conversion. Instead, it uses the statement

```
add    AL,'A'-'a'
```

as shown in Program 4.4.

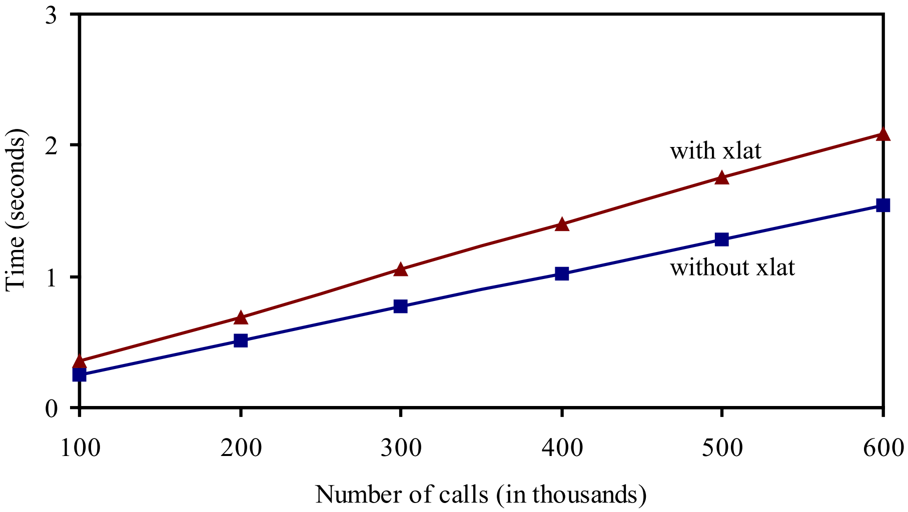


Figure 4.1 Performance of the case conversion program.

The other version uses the `xlat` instruction for case conversion. In order to do so, we have to set up the following conversion table in the data section:

```
upper_table    DB    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Furthermore, after initializing EBX to `upper_table`, the following code

```
sub    AL,'a'
xlat
```

replaces the code

```
add    AL,'A'-'a'
```

You can clearly see the disadvantage of the `xlat` version of the code. First of all, it requires additional space to store the translation table `upper_table`. More important than this is the fact that the `xlat` version requires additional time. Note that the `add` and `sub` instructions take the same amount of time to execute. Therefore, the `xlat` version requires additional time to execute `xlat`, which generates a memory read to get the byte from `upper_table` located in the data segment.

The performance superiority of the first version (i.e., the version that does not use the `xlat` instruction) is clearly shown in Figure 4.1. These results were obtained on a 2.4-GHz Pentium 4 system. In this plot, the x-axis gives the number of times the case conversion procedure is called to convert a lowercase string of 1,000 characters. The data show that using the `xlat` instruction deteriorates the performance by about 35 percent! For the reasons discussed before, this is clearly a bad example to use the `xlat` instruction.

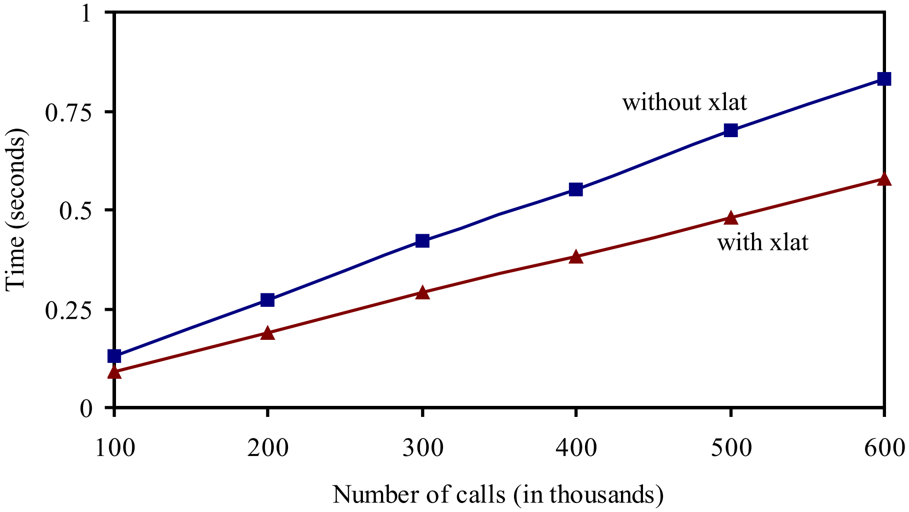


Figure 4.2 Performance of the hex conversion program.

4.9.2 Experiment 2

In this experiment, we use the hex conversion examples of Section 4.8 to show the benefits of the `xlat` instruction. As shown in Example 4.7, without using the `xlat`, we have to test the input number to see if it falls in the range of 0–9 or 10–15. However, such testing and hence the associated overhead can be avoided by using a translation table along with `xlat`.

The two programs of Examples 4.7 and 4.8 have been converted to C callable procedures as in the last experiment. Each procedure receives a string and converts the characters in the input string to their hex equivalents. However, the hex code is not displayed. The input test string in this experiment consists of lowercase and uppercase letters, digits, and special symbols for a total of 100 characters.

The results, obtained on a 2.4-GHz Pentium 4 system, are shown in Figure 4.2. The data presented in this figure clearly demonstrate the benefit of using the `xlat` in this example. The procedure that does not use the `xlat` instruction is about 45% slower!

The moral of the story is that judicious use of assembly language instructions is necessary in order to reap the benefits of the assembly language.

4.10 Summary

In this chapter, we presented basics of the Pentium assembly language programming. We discussed three types of assembly language statements:

1. Executable statements that instruct the CPU as to what to do;

2. Assembler directives that facilitate the assembly process;
3. Macros that facilitate modular program design.

Assembler directives to allocate storage space for data variables and to define numeric and string constants were discussed in detail. Macros provide a sophisticated text substitution mechanism. Although not used as frequently as procedures, macros are useful in certain situations.

An overview of the Pentium instruction set was also presented. While we discussed in detail the data transfer instructions, there was only a brief review of the remaining instructions of the Pentium instruction set. A detailed discussion of these instructions is provided in later chapters.

We also demonstrated the performance advantage of the `xlat` instruction under certain conditions. The results show that judicious use of the `xlat` instruction provides significant performance advantages for character conversions. On the other hand, there are instances wherein the `xlat` instruction deteriorates performance.

4.11 Exercises

4–1 Why doesn't the CPU execute assembler directives?

4–2 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  TIMES  100  DW  -1
(b) value  DW  -2300
(c) count  DW  40000
(d) msg1   DB  'Finder''s fee is:',0
(e) msg3   DB  'Sorry! Invalid input.',0DH,0AH,0
```

4–3 What is an addressing mode? Why does Pentium provide several addressing modes?

4–4 We discussed four different addressing modes in this chapter. Which addressing mode is the most efficient? Explain why.

4–5 Can we use the immediate addressing mode in the `inc` instruction? Justify your answer.

4–6 Discuss the pros and cons of using the `lea` instruction as opposed to using the `mov` instruction to get the effective address.

4–7 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
```

4–8 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    ECX,EDX            (l) sub    DX,CX
```

4–9 In 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)                          (d)
    mov    CX,15BAH              mov    CX,5
    mov    DX,8244H              not    CX
    and    DX,CX                 mov    DX,10
    jz     jump1                 cmp    CX,DX
    mov    BX,1                  jg     jump1
    jmp    skip1                 mov    BX,1
jump1:                           jmp    skip1
    mov    AX,10             jump1:
```

```
skip1:                             mov    AX,10
         . . .                 skip1:
                                         . . .
```

4–10 Describe in one sentence what the following code is accomplishing in terms of number manipulation:

```
(a)                                (b)
     not    AX                          not    AX
     add    AX,1                        inc    AX
(c)                                (d)
     sub    AH,AH                       sub    AH,AH
     sub    DH,DH                       sub    DH,DH
     mov    DL,AL                       mov    DL,AL
     add    DX,DX                       mov    CL,3
     add    DX,DX                       shl    DX,CL
     add    DX,AX                       shl    AX,1
     add    DX,DX                       add    DX,AX
```

4–11 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)                                (b)
     mov    DX,AX                       mov    DX,AX
     not    AX                          not    AX
     or     AX,DX                       and    AX,DX
```

4.12 Programming Exercises

4–P1 Modify the program of Example 4.6 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; otherwise, repeats the query.

4–P2 Modify the program of Example 4.6 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.

4–P3 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
```

4–P4 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 or not 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 he or 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.

4–P5 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. A typical interaction of your program is (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.

4–P6 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!)

4–P7 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 `GetInt` routine. As with the other programming exercises, you should query the user for program termination.

4–P8 Modify the above program to display the octal equivalent instead of the hexadecimal equivalent of the input number.

4–P9 Write a complete assembly language program to perform logical-address to physical-address translation (see Chapter 3 for details on the translation process). Your program should take a logical address as its input and display the corresponding physical address. The input consists of two parts: segment value and offset value. Both are given as hexadecimal numbers. You can assume the real mode.

Chapter 5

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 5.1 introduces the stack concept, and the next section discusses how the stack is implemented in the Pentium processor. Stack operations—push and pop—are discussed in Section 5.3. Section 5.4 discusses some typical uses of the stack.

After a brief introduction to procedures in Section 5.5, the Pentium instructions for procedure invocation and return are discussed in Section 5.6. Parameter passing mechanisms are discussed in detail in Section 5.7. The stack plays an important role in parameter passing. Using the stack it is relatively straightforward to pass variable number of arguments to a procedure. We discuss this topic in Section 5.8. The issue of local variable storage in procedures is discussed in Section 5.9.

Although short assembly language programs can be stored in a single file, real application programs are broken into several files called modules. The issues involved in writing and assembling multiple source program modules are discussed in Section 5.10. Section 5.11 presents the performance overheads associated with procedures. The last section provides a summary of the chapter.

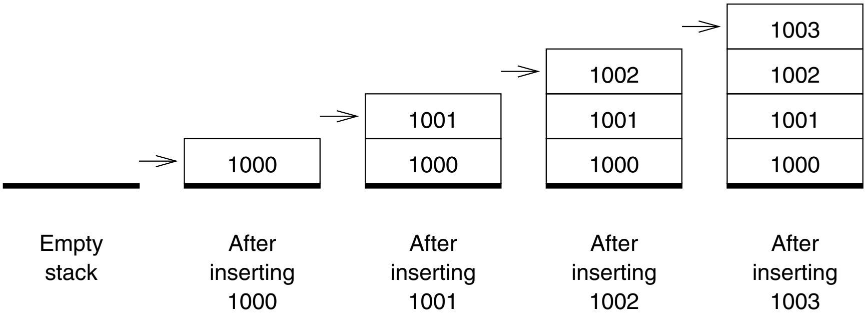


Figure 5.1 An example showing stack growth: Numbers 1000 through 1003 are inserted in ascending order. The arrow points to the top-of-stack.

5.1 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 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 the 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 5.1. 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 5.2).

By 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.

5.2 Pentium Implementation of the Stack

The memory space reserved in the stack segment is used to implement the stack. The registers SS and ESP are used to implement the Pentium stack. The top-of-stack, which points to the last item inserted into the stack, is indicated by SS:ESP, with the SS register pointing to the

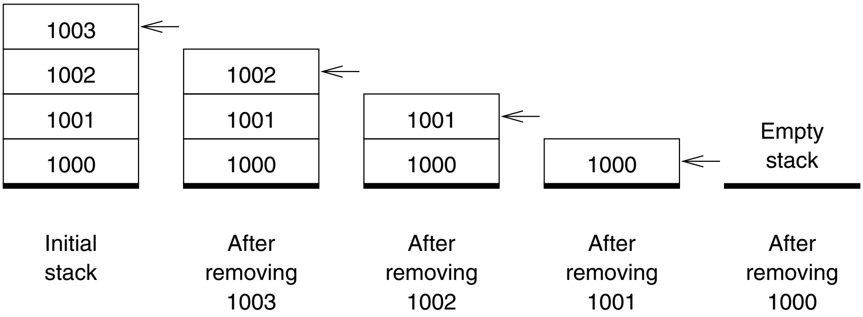


Figure 5.2 Deletion of data items from the stack: The arrow points to the top-of-stack.

beginning of the stack segment, and the ESP 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 (TOS) always points to the last data item placed on the stack. TOS always points to the lower byte of the last word inserted into the stack.

Figure 5.3a shows an empty stack with 256 bytes of memory for stack operations. When the stack is initialized, TOS points to the byte just outside the reserved stack area. It is an error to read from an empty stack, as this causes *stack underflow*.

When a word is pushed onto the stack, ESP is first decremented by two, and then the word is stored at SS:ESP. 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 ESP is decremented by two to point to the last data item, as shown in Figure 5.3b. The stack shown in Figure 5.3c 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., ESP = 0). 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.

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 5.4a, we get 7FBD329AH from the stack and ESP is updated, as shown in Figure 5.4b. Notice that the four memory locations retain their values. However, since TOS is updated, these four locations can be used to store the next data value pushed onto the stack, as shown in Figure 5.4c.

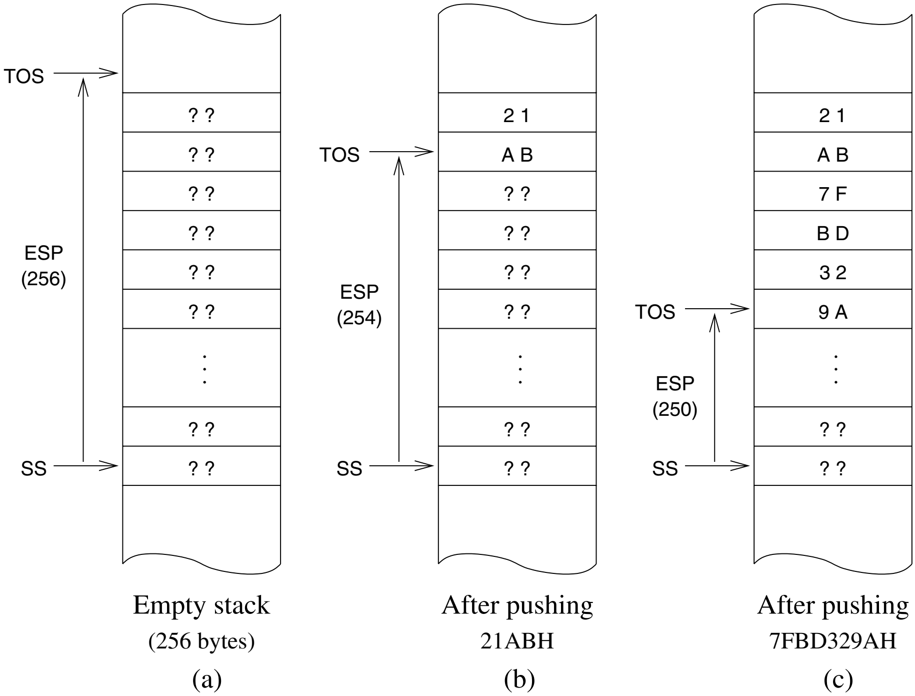


Figure 5.3 Stack implementation in the Pentium: SS:ESP points to the top-of-stack.

5.3 Stack Operations

The Pentium instruction set has several instructions to support stack operations. These include the basic instructions like `push` and `pop` that operate on single registers. In addition, separate instructions are provided to push and pop the flags register as well as the complete register set.

5.3.1 Basic Instructions

The Pentium allows the push and pop 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 can be an immediate operand of size 8, 16, or 32 bits. Table 5.1 summarizes the two stack operations.

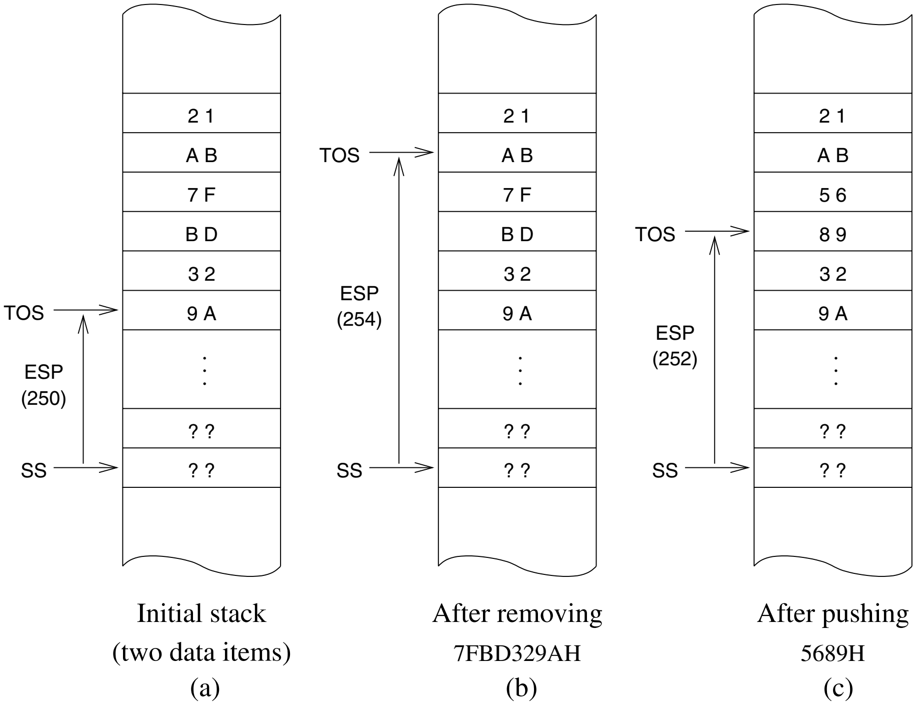


Figure 5.4 An example showing stack insert and delete operations.

On an empty stack shown in Figure 5.3a the statements

```
push   21ABH
push   7FBD329AH
```

would result in the stack shown in Figure 5.4a. Executing the statement

```
pop    EBX
```

on this stack would result in the stack shown in Figure 5.4b with the register EBX receiving 7FBD329AH.

5.3.2 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 5.1 Stack Operations on 16- and 32-Bit Data

`push source16`	ESP = ESP − 2 SS:ESP = `source16`	ESP 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 two bytes.
`push source32`	ESP = ESP − 4 SS:ESP = `source32`	ESP 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 four bytes.
`pop dest16`	`dest16` = SS:ESP ESP = ESP + 2	The data item located at TOS is copied to `dest16`. Then ESP is incremented by 2 to update TOS. The stack shrinks by two bytes.
`pop dest32`	`dest32` = SS:ESP ESP = ESP + 4	The data item located at TOS is copied to `dest32`. Then ESP is incremented by 4 to update TOS. The stack shrinks by four 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:

`pushfd`	(push 32-bit flags)
`popfd`	(pop 32-bit flags)

These instructions do not need any operands. For operating on the 16-bit flags register (FLAGS), we can use `pushfw` and `popfw` instructions. If we use `pushf` the default operand size selects either `pushfd` or `pushfw`. In our programs, since our default is 32-bit operands, `pushf` is used as an alias for `pushfd`. However, we use `pushfd` to make the operand size explicit. Similarly, `popf` can be used as an alias for either `popfd` or `popfw`.

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 `pushad` saves the 32-bit general-purpose registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. These registers are pushed in the order specified. The last register pushed is the EDI register. The `popad` restores these registers except that it will not copy the ESP value (i.e., the ESP value is not loaded into the ESP register as part of the `popad` instruction). The corresponding instructions for the 16-bit registers are `pushaw` and popaw. These instructions are useful in procedure calls, as we show in Section 5.7.4. Like pushf and popf, we can use pusha and popa as aliases.

5.4 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.

5.4.1 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 and restore these registers as shown below:

```
        . . .
;save EAX and EBX registers on the stack
     push   EAX
     push   EBX
     ;EAX and EBX registers can now be used
     mov    EAX,value1
     mov    EBX,value2
     mov    value1,EBX
     mov    value2,EAX
;restore EAX and EBX 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 10, 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 5.7.

It should be clear from these examples that the stack grows and shrinks during the course of 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.

5.4.2 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 appears in Section 5.6.

5.4.3 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 5.7.

5.5 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, there are two types of subprograms: *procedures* and *functions*. 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 are 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 receives parameters a, b, and c via 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);
```

We visit this example in Chapter 18, which discusses floating-point operations.

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.

5.6 Pentium Instructions for Procedures

The Pentium provides call and ret (return) instructions to write procedures in the 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.

5.6.1 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:
                                         . . .
00000002    E816000000             call   sum
00000007    89C3                   mov    EBX,EAX
                                         . . .
                               ; end of main procedure
;*****************************************************
                               sum:
0000001D    55                     push   EBP
                                         . . .
                               ; end of sum procedure
;*****************************************************
                               avg:
                                         . . .
00000028    E8F0FFFFFF             call   sum
0000002D    89D8                   mov    EAX,EBX
                                         . . .
                               ; end of avg procedure
;*****************************************************
```

After the `call` instruction in `main` has been fetched, the EIP register points to the next instruction to be executed (i.e., EIP = 00000007H). This is the instruction that should be executed after completing the execution of `sum` procedure. The processor makes a note of this by pushing the contents of the EIP register onto the stack.

Now, to transfer control to the first instruction of the `sum` procedure, the EIP register would have to be loaded with the offset value of the

```
push   EBP
```

instruction in `sum`. To do this, the processor adds the 32-bit relative displacement found in the `call` instruction to the contents of the EIP register. Proceeding with our example, the machine language encoding of the `call` instruction, which requires five bytes, is E816000000H. The first byte E8H is the opcode for the `call` and the next four bytes give the (signed) relative displacement in bytes. In this example, it is the difference between 0000001DH (offset of the `push EBP` instruction in `sum`) and 00000007H (offset of the instruction `mov EBX,EAX` in `main`). Therefore, the displacement should be 0000001DH − 00000007H = 00000016H.

This is the displacement value encoded in the `call` instruction. Note that this displacement value in this instruction is shown in the little-endian order, which is equal to 00000016H. Adding this difference to the contents of the EIP register leaves the EIP register pointing to the first instruction of `sum`.

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 `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 0000001DH − 0000002DH = FFFFFFF0H. Since negative numbers are expressed in 2's complement notation, FFFFFFF0H corresponds to −10H (i.e., −16D), which is the displacement value in bytes.

The following is a summary of the actions taken during a near procedure call:

```
ESP = ESP − 4                      ; push return address onto the stack
SS:ESP = EIP
EIP = EIP + relative displacement  ; update EIP to point to the procedure
```

The relative displacement is a signed 32-bit number to accommodate both forward and backward procedure calls.

5.6.2 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 EBX,EAX` 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

```
EIP = SS:ESP    ; pop return address at TOS into IP
ESP = ESP + 4   ; update TOS by adding 4 to ESP
```

An optional integer may be included in the `ret` instruction, as in

```
ret  8
```

The details on this optional number are covered in Section 5.7.2, which discusses the stack-based parameter passing mechanism.

5.7 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.

5.7.1 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 5.1 *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 5.1, requests two integers from the user and displays the sum on the screen.

Program 5.1 Parameter passing by call-by-value using registers

```
;Parameter passing via registers            PROCEX1.ASM
;
;        Objective: To show parameter passing via registers.
;            Input: Requests two integers from the user.
;           Output: Outputs the sum of the input integers.
%include "io.mac"
.DATA
prompt_msg1  db   "Please input the first number: ",0
prompt_msg2  db   "Please input the second number: ",0
sum_msg      db   "The sum is ",0

.CODE
      .STARTUP
      PutStr  prompt_msg1    ; request first number
      GetInt  CX             ; CX = first number

      PutStr  prompt_msg2    ; request second number
      GetInt  DX             ; DX = second number

      call    sum            ; returns sum in AX
      PutStr  sum_msg        ; display sum
      PutInt  AX
      nwln
```

```
done:
      .EXIT

;------------------------------------------------------------
;Procedure sum receives two integers in CX and DX.
;The sum of the two integers is returned in AX.
;------------------------------------------------------------
sum:
      mov     AX,CX           ; sum = first number
      add     AX,DX           ; sum = sum + second number
      ret
```

Example 5.2 *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 str_len function. 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] $\neq$ 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 EBX and returns the string length in the AX register. The program listing is given in Program 5.2. The main procedure executes

```
mov    EBX,string
```

to place the address of string in EBX (line 22) before invoking the procedure on line 23. Note that even though the procedure modifies the EBX register during its execution, it restores the original value of EBX by saving its value initially on the stack (line 35) and restoring it (line 44) before returning to the main procedure.

Program 5.2 Parameter passing by call-by-reference using registers

```
;Parameter passing via registers      PROCEX2.ASM
;
;        Objective: To show parameter passing via registers.
;            Input: Requests a character string from the user.
;           Output: Outputs the length of the input string.

%include "io.mac"
BUF_LEN     EQU  41          ; string buffer length

.DATA
prompt_msg  db  "Please input a string: ",0
length_msg  db  "The string length is ",0

.UDATA
string      resb  BUF_LEN    ;input string < BUF_LEN chars.

.CODE
      .STARTUP
      PutStr  prompt_msg     ; request string input
      GetStr  string,BUF_LEN ; read string from keyboard

      mov     EBX,string     ; EBX = string address
      call    str_len        ; returns string length in AX
      PutStr  length_msg     ; display string length
      PutInt  AX
      nwln
done:
      .EXIT

;-----------------------------------------------------------
;Procedure str_len receives a pointer to a string in EBX.
;String length is returned in AX.
;-----------------------------------------------------------
str_len:
      push    EBX
      sub     AX,AX          ; string length = 0
repeat:
      cmp     byte [EBX],0   ; compare with NULL char.
      je      str_len_done   ; if NULL we are done
      inc     AX             ; else, increment string length
      inc     EBX            ; point EBX to the next char.
      jmp     repeat         ;  and repeat the process
```

```
str_len_done:
      pop     EBX
      ret
```

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 is a limited number of general-purpose registers available.
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.

5.7.2 Stack Method

In this method, all parameters 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 5.1. This can be done by

```
push   number1
push   number2
call   sum
```

After executing the call instruction, which automatically pushes the EIP contents onto the stack, the stack state is shown in Figure 5.5.

Reading the two arguments—`number1` and `number2`—is tricky. Since the parameter values are buried inside the stack, first we have to pop the EIP value to read the required two parameters. This, for example, can be done by

```
pop    EAX
pop    EBX
pop    ECX
```

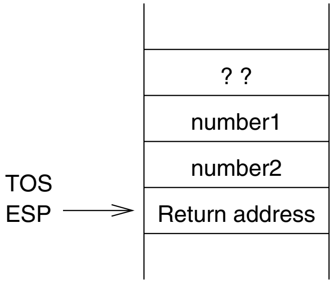


Figure 5.5 Stack state after the sum procedure call: Return address is the EIP value pushed onto the stack as part of executing the call instruction.

in the sum procedure. Since we have removed the return address (EIP) from the stack, we will have to restore it by

```
push   EAX
```

so that TOS points 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 off the stack as needed. Since the stack is a sequence of memory locations, ESP + 4 points to number2, and ESP + 6 to number1. For instance,

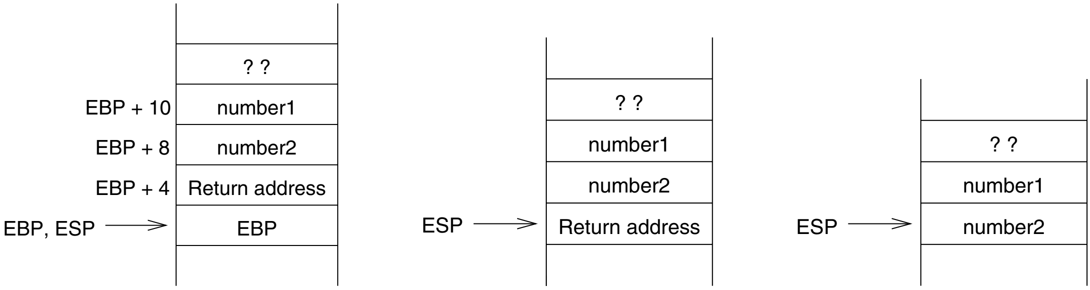
```
mov    EBX,[ESP+4]
```

can be used to read number2, but this causes a problem. The stack pointer 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 EBP register instead of ESP to specify an offset into the stack segment. For example, we can copy the value of number2 into the AX register by

```
mov    EBP,ESP
mov    AX,[EBP+4]
```

This is the usual way of accessing the parameters from the stack. Since every procedure uses the EBP register to access the parameters, the EBP register should be preserved. Therefore, we should save the contents of the EBP register before executing the



(a) Stack after saving EBP (b) Stack after `pop EBP` (c) Stack after `ret`

Figure 5.6 Changes in stack state during a procedure execution.

```
mov     EBP,ESP
```

statement. We, of course, use the stack for this. Note that

```
push    EBP
mov     EBP,ESP
```

causes the parameter displacement to increase by four bytes, as shown in Figure 5.6a.

The information stored in the stack—parameters, return address, and the old EBP value—is collectively called the *stack frame*. As we show on page 151, the stack frame also consists of local variables if the procedure uses them. The EBP value is referred to as the *frame pointer* (FP). Once the EBP value is known, we can access all items in the stack frame.

Before returning from the procedure, we should use

```
pop    EBP
```

to restore the original value of EBP. The resulting stack state is shown in Figure 5.6b.

The `ret` statement discussed in Section 5.6.2 causes the return address to be placed in the EIP register, and the stack state after `ret` is shown in Figure 5.6c.

We have a problem here—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 ESP by four after the call statement, as shown below:

```
push     number1
push     number2
call     sum
add      ESP,4
```

For example, C compilers use 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:
        . . .
      add    ESP,4
      ret
```

because when `ret` is executed, ESP 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:

EIP = SS:ESP
ESP = ESP + 4 + `optional-value`

The `optional-value` should be a number (i.e., 16-bit immediate value). 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 discussed the following ways of discarding the unwanted parameters on the stack:

1. clean-up done by the calling procedure, or
2. clean-up 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 Section 5.8.

5.7.3 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    ECX,count
```

```
repeat:
        call    compute
            . . .
        loop    repeat
            . . .
```

The code invokes the compute procedure count times. The ECX register maintains the number of remaining iterations. Recall that, as part of executing the loop instruction, the ECX register is decremented by 1 and, if not 0, starts another iteration.

Now suppose that the compute procedure uses the ECX register during its execution. Then, when compute returns control to the calling program, ECX would have changed, and the program logic would be incorrect. To preserve the contents of the ECX register, it should be saved. Of course, we use the stack for this purpose.

5.7.4 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, gcc uses the EAX register to return integer results.

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 pushad instruction (see page 122). But such an action results in unnecessary overhead, as pushad 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 are 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 the modular program design principles.

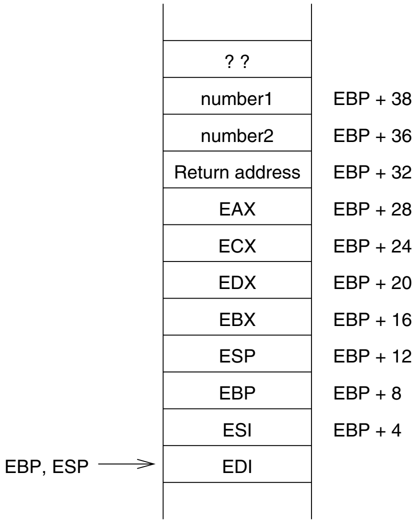


Figure 5.7 Stack state after `pusha`.

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, the EAX register is often used to return integer results. 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 five registers. If we 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     EBP,ESP
```

causes the stack state, shown in Figure 5.7, to be different from that shown in Figure 5.6a on page 134. You can see that the offset of `number1` and `number2` increases substantially.

5.7.5 ENTER and LEAVE Instructions

The Pentium instruction set has two instructions to facilitate allocation and release of stack frames. The `enter` instruction can be used to allocate a stack frame on entering a procedure. The format is

```
enter    bytes,level
```

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 5.8 on page 153. 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 the statement

```
enter    XX,0
```

is equivalent to

```
push    EBP
mov     EBP,ESP
sub     ESP,XX
```

The `leave` instruction releases the stack frame allocated by the `enter` instruction. It does not take any operands. The `leave` instruction effectively performs the following:

```
mov     ESP,EBP
pop     EBP
```

We place the `leave` instruction just before the `ret` instruction as shown in the following template for procedures:

```
proc-name:
    enter   XX,0
        . . .
    procedure body
        . . .
    leave
    ret    YY
```

As we show on page 154, 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.

5.7.6 Illustrative Examples

In this section, we use three examples to illustrate the use of the stack for parameter passing.

Example 5.3 *Parameter passing by call-by-value using the stack.*
This is the stack counterpart of Example 5.1, which passes two integers to the procedure `sum`. The procedure returns the sum of these two integers in the AX register, as in Example 5.1. The program listing is given in Program 5.3.

The program requests two integers from the user. It reads the two numbers into the CX and DX registers using `GetInt` (lines 16 and 19). 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 21 and 22). The state of the stack after the control is transferred to `sum` is shown in Figure 5.5 on page 133.

As discussed in Section 5.7.2, the EBP register is used to access the two parameters from the stack. Therefore, we have to save EBP itself on the stack. We do this by using the `enter` instruction (line 35), which changes the stack state to that in Figure 5.6a on page 134.

The original value of EBP is restored at the end of the procedure using the `leave` instruction (line 38). Accessing the two numbers follows the explanation given in Section 5.7.2. Note that the first number is at EBP + 10, and the second one at EBP + 8. As in Example 5.1, 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` instruction clears `number1` and `number2` from the stack.

Program 5.3 Parameter passing by call-by-value using the stack

```
 1:  ;Parameter passing via the stack      PROCEX3.ASM
 2:  ;
 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:  %include "io.mac"
 7:
 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:        .STARTUP
15:        PutStr  prompt_msg1    ; request first number
16:        GetInt  CX             ; CX = first number
17:
18:        PutStr  prompt_msg2    ; request second number
19:        GetInt  DX             ; DX = second number
20:
21:        push    CX             ; place first number on stack
22:        push    DX             ; place second number on stack
23:        call    sum            ; returns sum in AX
24:        PutStr  sum_msg        ; display sum
25:        PutInt  AX
```

```
      nwln
done:
      .EXIT

;-----------------------------------------------------------
;Procedure sum receives two integers via the stack.
; The sum of the two integers is returned in AX.
;-----------------------------------------------------------
sum:
      enter   0,0             ; save EBP
      mov     AX,[EBP+10]     ; sum = first number
      add     AX,[EBP+8]      ; sum = sum + second number
      leave                   ; restore EBP
      ret     4               ; return and clear parameters
```

Example 5.4 *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 5.4, 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 23 to 26). The swap procedure, after saving the EBP register as in the last example, can access the pointers of the two characters at EBP + 8 and EBP + 12. Since the procedure uses the EBX register, we save it on the stack as well. Note that, once the EBP is pushed onto the stack and the ESP value is copied to EBP, the two parameters (i.e., the two character pointers in this example) are available at EBP + 8 and EBP + 12, irrespective of the other stack push operations in the procedure. This is important from the program maintenance point of view.

Program 5.4 Parameter passing by call-by-reference using the stack

```
 1:  ;Parameter passing via the stack      PROCSWAP.ASM
 2:  ;
 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:  %include "io.mac"
```

```

.DATA
prompt_msg  db    "Please input a string: ",0
output_msg  db    "The swapped string is: ",0

.UDATA
string      resb  BUF_LEN      ;input string < BUF_LEN chars.

.CODE
      .STARTUP
      PutStr  prompt_msg     ; request string input
      GetStr  string,BUF_LEN ; read string from the user

      mov     EAX,string     ; EAX = string[0] pointer
      push    EAX
      inc     EAX            ; EAX = string[1] pointer
      push    EAX
      call    swap           ; swaps the first two characters
      PutStr  output_msg     ; display the swapped string
      PutStr  string
      nwln
done:
      .EXIT

;-----------------------------------------------------------
;Procedure swap receives two pointers (via the stack) to
; characters of a string. It exchanges these two characters.
;-----------------------------------------------------------
.CODE
swap:
      enter   0,0
      push    EBX            ; save EBX - procedure uses EBX
      ; swap begins here. Because of xchg, AL is preserved.
      mov     EBX,[EBP+12]   ; EBX = first character pointer
      xchg    AL,[EBX]
      mov     EBX,[EBP+8]    ; EBX = second character pointer
      xchg    AL,[EBX]
      mov     EBX,[EBP+12]   ; EBX = first character pointer
      xchg    AL,[EBX]
      ; swap ends here
      pop     EBX            ; restore registers
      leave
      ret     8              ; return and clear parameters
```

```
       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 5.8 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 5.9 Behavior of the bubble sort algorithm.

Example 5.5 *Bubble sort procedure.*

There are several algorithms to sort an array of numbers. The algorithm we use here is called the *bubble sort* algorithm. We assume that the array is to be sorted in ascending order. The bubble sort algorithm consists of several passes through the array. Each pass scans the array, performing the following actions:

- Compare adjacent pairs of data elements;
- If they are out of order, swap them.

The algorithm terminates if, during a pass, no data elements are swapped. Even if a single swap is done during a pass, it will initiate another pass to scan the array.

Figure 5.8 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 5.9 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 this.

```
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 $i$th and $(i + 1)$th elements of the array
                status := UNSORTED
            end if
        end for
    end while
end bubble_sort
```

Figure 5.10 Pseudocode for the bubble sort algorithm.

The number of passes required to sort an array depends on how unsorted the initial array is. If the array is 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 the maximum 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 5.10.

The bubble sort program (Program 5.5) 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 25 to 32) reads the input integers. Since the ECX register is initialized to MAX_SIZE, which is set to 20 in this program, the read_loop iterates a maximum of 20 times. Typing a zero can also terminate the loop. The zero input condition is detected and the loop is terminated by the statements on lines 27 and 28.

The bubble_sort procedure receives the array size and a pointer to the array. These two parameters are pushed onto the stack (lines 34 and 35) before calling the bubble_sort procedure. The print_loop (lines 41 to 45) displays the sorted array.

In the bubble-sort procedure, the ECX register is used to keep track of the number of comparisons while EDX maintains the status information. The ESI register points to the ith element of the input array.

The while loop condition is tested by lines 91 to 93. The for loop body corresponds to lines 80 to 89 and 97 to 100. The rest of the code follows the pseudocode. Note that the array pointer is available in the stack at EBP + 36 and its size at EBP + 40, as we use pushad to save all registers.

Program 5.5 Bubble sort program to sort integers in ascending order

```
;Bubble sort procedure                      BBLSORT.ASM
;     Objective: To implement the bubble sort algorithm.
;         Input: A set of nonzero integers to be sorted.
;                Input is terminated by entering zero.
;        Output: Outputs the numbers in ascending order.

%define    CRLF  0DH,0AH
MAX_SIZE   EQU   20
%include "io.mac"
.DATA
prompt_msg  db  "Enter nonzero integers to be sorted.",CRLF
            db  "Enter zero to terminate the input.",0
output_msg  db  "Input numbers in ascending order:",0

.UDATA
array       resd  MAX_SIZE   ; input array for integers

.CODE
      .STARTUP
      PutStr  prompt_msg     ; request input numbers
      nwln
      mov     EBX,array      ; EBX = array pointer
      mov     ECX,MAX_SIZE   ; ECX = array size
      sub     EDX,EDX        ; number count = 0
read_loop:
      GetLInt EAX            ; read input number
      cmp     EAX,0          ; if the number is zero
      je      stop_reading   ; no more numbers to read
      mov     [EBX],EAX      ; copy the number into array
      add     EBX,4          ; EBX points to the next element
      inc     EDX            ; increment number count
      loop    read_loop      ; reads a max. of MAX_SIZE numbers
stop_reading:
      push    EDX            ; push array size onto stack
      push    array          ; place array pointer on stack
      call    bubble_sort
      PutStr  output_msg     ; display sorted input numbers
      nwln
      mov     EBX,array
      mov     ECX,EDX        ; ECX = number count
print_loop:
      PutLInt  [EBX]
      nwln
      add     EBX,4
      loop    print_loop
done:
      .EXIT
;-----------------------------------------------------------
;This procedure receives a pointer to an array of integers
; and the size of the array via the stack. It sorts the
; array in ascending order using the bubble sort algorithm.
;-----------------------------------------------------------
SORTED    EQU   0
UNSORTED  EQU   1
bubble_sort:
      pushad
      mov     EBP,ESP

      ; ECX serves the same purpose as the end_index variable
      ; in the C procedure. ECX keeps the number of comparisons
      ; to be done in each pass. Note that ECX is decremented
      ; by 1 after each pass.
      mov     ECX, [EBP+40]  ; load array size into ECX

next_pass:
      dec     ECX           ; if # of comparisons is zero
      jz      sort_done     ; then we are done
      mov     EDI,ECX       ; else start another pass

      ;DL is used to keep SORTED/UNSORTED status
      mov     DL,SORTED     ; set status to SORTED

      mov     ESI,[EBP+36]  ; load array address into ESI
      ; ESI points to element i and ESI+4 to the next element
pass:
      ; This loop represents one pass of the algorithm.
      ; Each iteration compares elements at [ESI] and [ESI+4]
      ; and swaps them if ([ESI]) < ([ESI+4]).

      mov     EAX,[ESI]
      mov     EBX,[ESI+4]
      cmp     EAX,EBX
      jg      swap

increment:
      ; Increment ESI by 4 to point to the next element
```

```
        add     ESI,4
        dec     EDI
        jnz     pass

        cmp     EDX,SORTED      ; if status remains SORTED
        je      sort_done       ; then sorting is done
        jmp     next_pass       ; else initiate another pass

swap:
        ; swap elements at [ESI] and [ESI+4]
        mov     [ESI+4],EAX     ; copy [ESI] in EAX to [ESI+4]
        mov     [ESI],EBX       ; copy [ESI+4] in EBX to [ESI]
        mov     EDX,UNSORTED    ; set status to UNSORTED
        jmp     increment

sort_done:
        popad
        ret     8
```

5.8 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 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, 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 5.6 *Passing 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. The procedure finds the sum of the integers and returns this value in the EAX register.

The `main` procedure in Program 5.6 requests input from the user. Only nonzero values are accepted as valid input (entering a zero terminates the input). The `read_number` loop (lines 24 to 30) reads input numbers using `GetLInt` and pushes them onto the stack. The ECX register keeps a count of the number of input values, which is passed as the last parameter

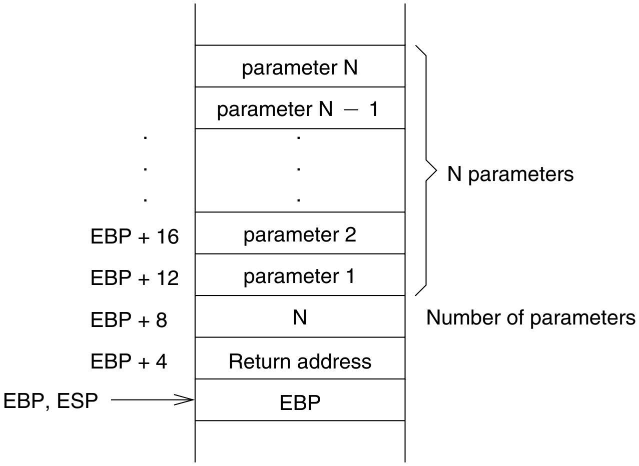


Figure 5.11 State of the stack after executing the `enter` statement.

(line 32) before calling the `variable_sum` procedure. The state of the stack at line 53, after executing the `enter` instruction, is shown in Figure 5.11.

The `variable_sum` procedure first reads the number of parameters passed onto it from the stack at EBP + 8 into the ECX register. The `add_loop` (lines 60 to 63) successively reads each integer from the stack and computes their sum in the EAX. Note that on line 61 we use a segment override prefix. If we write

```
add     EAX,[EBX]
```

the contents of the EBX 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 in EBX 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.

Notes

1. If you are running this program on a Linux system, you don't need the segment override prefix. The reason is that Linux and UNIX systems do not use the physical segmentation provided by the Pentium. Instead, these systems treat the memory as a single physical segment, which is partitioned into various logical segments. Figure 5.12 shows the memory layout for Linux. The bottom two segments are used for the code and data. For example, the code segment (`.text`) is placed at the bottom, which is a read-only segment. The next segment stores the data part (`.data` and `.bss`). The stack segment is placed below the kernel space.

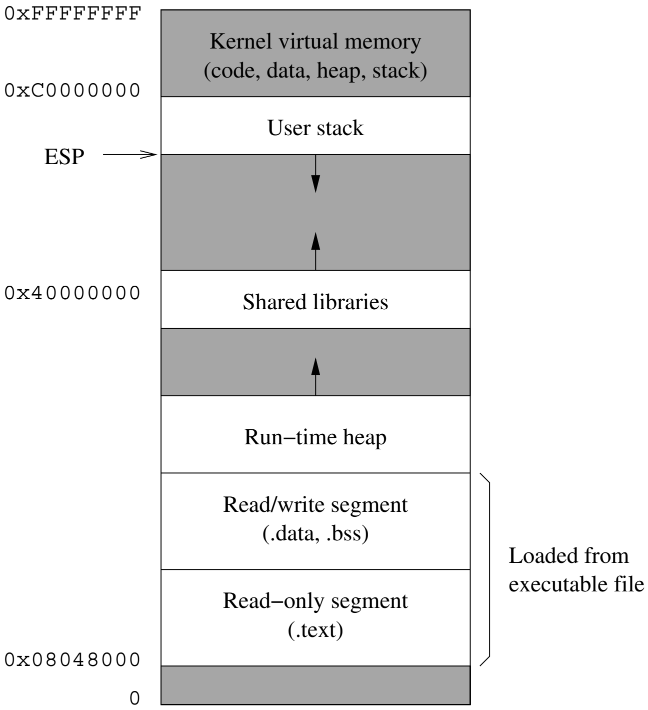


Figure 5.12 Memory layout of a Linux process.

2. In this example, we deliberately used the EBX to illustrate the use of segment override prefixes. We could have used the EBP itself to access the parameters. For example, the code

```
          add    EBP,12
          sub    EAX,EAX
    add_loop:
          add    EAX,[EBP]
          add    EBP,4
          loop   add_loop
```

   can replace the code at lines 58 to 63. A disadvantage of this version is that, since we have modified the EBP, 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 EBP. We defer this discussion to the next chapter, which discusses some additional addressing modes of the Pentium.

3. 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 param-

eter space. This is done in `main` by lines 35 to 38. The ECX is first incremented to include the count parameter (line 35). The byte count of the parameter space is computed on lines 36 and 37. These lines effectively multiply ECX by four. This value is added to the ESP register to clear the parameter space (line 38).

Program 5.6 Program to illustrate passing a variable number of parameters

```
 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   0DH,0AH    ; carriage return and line feed
10:
11:  %include "io.mac"
12:
13:  .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:  .CODE
20:        .STARTUP
21:        PutStr  prompt_msg     ; request input numbers
22:        nwln
23:        sub     ECX,ECX        ; ECX keeps number count
24:  read_number:
25:        GetLInt EAX            ; read input number
26:        cmp     EAX,0          ; if the number is zero
27:        je      stop_reading   ; no more nuumbers to read
28:        push    EAX            ; place the number on stack
29:        inc     ECX            ; increment number count
30:        jmp     read_number
31:  stop_reading:
32:        push    ECX            ; place number count on stack
33:        call    variable_sum   ; returns sum in EAX
34:        ; clear parameter space on the stack
35:        inc     ECX            ; increment ECX to include count
36:        add     ECX,ECX        ; ECX = ECX * 4 (space in bytes)
37:        add     ECX,ECX
```

```
      add     ESP,ECX        ; update ESP to clear parameter
                             ; space on the stack
      PutStr  sum_msg        ; display the sum
      PutLInt EAX
      nwln
done:
      .EXIT

;------------------------------------------------------------
;This procedure receives variable number of integers via the
; stack. The last parameter pushed on the stack should be
; the number of integers to be added. Sum is returned in EAX.
;------------------------------------------------------------
variable_sum:
      enter   0,0
      push    EBX            ; save EBX and ECX
      push    ECX

      mov     ECX,[EBP+8]    ; ECX = # of integers to be added
      mov     EBX,EBP
      add     EBX,12         ; EBX = pointer to first number
      sub     EAX,EAX        ; sum = 0
add_loop:
      add     EAX,[SS:EBX]   ; sum = sum + next number
      add     EBX,4          ; EBX points to the next integer
      loop    add_loop       ; repeat count in ECX

      pop     ECX            ; restore registers
      pop     EBX
      leave
      ret                    ; parameter space cleared by main
```

5.9 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;
         . . .
         . . .
}
```

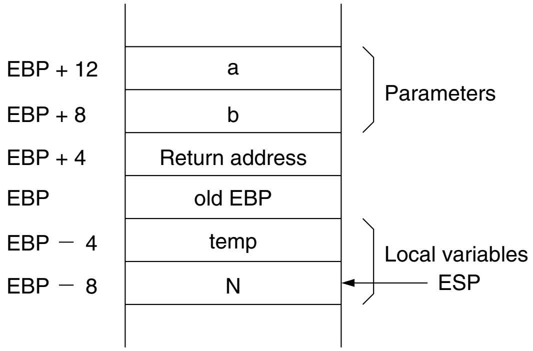


Figure 5.13 Stack frame with space for local variables.

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 (e.g., procedures that call themselves, either directly or indirectly).

For these reasons, space for local variables is reserved on the stack. Figure 5.13 shows the contents of the stack frame for the C function. In high-level languages, it is also referred to as the *activation record* because each procedure activation requires all this information. The EBP 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 EBP + 12 and EBP + 8, respectively. Local variables `temp` and `N` can be accessed at EBP − 4 and EBP − 8, respectively.

To aid program readability, we can use the `%define` directive to name the stack locations. Then we can write

```
mov     EBX,a
mov     temp,EAX
```

instead of

```
mov     EBX,[EBP+12]
mov     [EBP-4],EAX
```

after establishing `temp` and `a` labels by using the `%define` directive as shown below:

```
%define   a     dword [EBP+12]
%define   temp  dword [EBP-4]
```

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 5.7 *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 EAX and EBX. The higher of the two Fibonacci numbers is kept in the EBX. The `fib_loop` successively computes the Fibonacci number until it is greater than or equal to the input number. Then the Fibonacci number in EAX is returned to the `main` procedure.

Program 5.7 Fibonacci number computation with local variables mapped to registers

```
;Fibonacci numbers (register version)    PROCFIB1.ASM
;
;        Objective: To compute Fibonacci number using registers
;                   for local variables.
;            Input: Requests a positive integer from the user.
;           Output: Outputs the largest Fibonacci number that
;                   is less than or equal to the input number.

%include "io.mac"

.DATA
prompt_msg   db  "Please input a positive number (>1): ",0
output_msg1  db  "The largest Fibonacci number less than "
             db  "or equal to ",0
output_msg2  db  " is ",0
```

```

.CODE
      .STARTUP
      PutStr   prompt_msg     ; request input number
      GetLInt  EDX            ; EDX = input number
      call     fibonacci
      PutStr   output_msg1    ; display Fibonacci number
      PutLInt  EDX
      PutStr   output_msg2
      PutLInt  EAX
      nwln
done:
      .EXIT

;-----------------------------------------------------------
;Procedure fibonacci receives an integer in EDX and computes
; the largest Fibonacci number that is less than or equal to
; the input number. The Fibonacci number is returned in EAX.
;-----------------------------------------------------------
fibonacci:
      push    EBX
      ; EAX maintains the smaller of the last two Fibonacci
      ;  numbers computed; EBX maintains the larger one.
      mov     EAX,1          ; initialize EAX and EBX to
      mov     EBX,EAX        ;  first two Fibonacci numbers
fib_loop:
      add     EAX,EBX        ; compute next Fibonacci number
      xchg    EAX,EBX        ; maintain the required order
      cmp     EBX,EDX        ; compare with input number in EDX
      jle     fib_loop       ; if not greater, find next number
      ; EAX contains the required Fibonacci number
      pop     EBX
      ret
```

Example 5.8 *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).

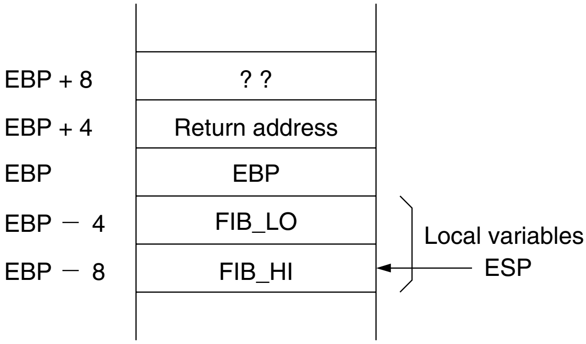
The code

```
push   EBP
mov    EBP,ESP
sub    ESP,8
```

saves the EBP value and copies the ESP value into the EBP as usual. It also decrements the ESP by 8, thus creating eight bytes of storage space for the two local variables `fib_lo` and `fib_hi`. This three-instruction sequence can be replaced by

```
enter  8,0
```

instruction. As mentioned before, the first operand specifies the number of bytes reserved for local variables. At this point, the stack allocation looks as follows:



The two local variables can be accessed at BP − 4 and BP − 8. The two EQU statements on lines 34 and 35 conveniently establish labels for these two locations. We can clear the local variable space and restore the EBP value by

```
mov    ESP,EBP
pop    EBP
```

instructions. The `leave` instruction performs exactly this. Thus, the `leave` instruction on line 53 automatically clears the local variable space. The rest of the code follows the logic of Example 5.7.

Program 5.8 Fibonacci number computation with local variables mapped to the stack

```
 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:  %include "io.mac"
 9:
10:  .DATA
11:  prompt_msg   db  "Please input a positive number (>1): ",0
```

```
output_msg1  db  "The largest Fibonacci number less than "
             db  "or equal to ",0
output_msg2  db  " is ",0

.CODE
      .STARTUP
      PutStr   prompt_msg     ; request input number
      GetLInt  EDX            ; EDX = input number
      call     fibonacci
      PutStr   output_msg1    ; print Fibonacci number
      PutLInt  EDX
      PutStr   output_msg2
      PutLInt  EAX
      nwln
done:
      .EXIT

;-----------------------------------------------------------
;Procedure fibonacci receives an integer in EDX and computes
; the largest Fibonacci number that is less than the input
; number. The Fibonacci number is returned in EAX.
;-----------------------------------------------------------
%define FIB_LO  dword [EBP-4]
%define FIB_HI  dword [EBP-8]
fibonacci:
      enter   8,0            ; space for two local variables
      push    EBX
      ; FIB_LO maintains the smaller of the last two Fibonacci
      ;  numbers computed; FIB_HI maintains the larger one.
      mov     FIB_LO,1       ; initialize FIB_LO and FIB_HI to
      mov     FIB_HI,1       ;  first two Fibonacci numbers
fib_loop:
      mov     EAX,FIB_HI     ; compute next Fibonacci number
      mov     EBX,FIB_LO
      add     EBX,EAX
      mov     FIB_LO,EAX
      mov     FIB_HI,EBX
      cmp     EBX,EDX        ; compare with input number in EDX
      jle     fib_loop       ; if not greater, find next number
      ; EAX contains the required Fibonacci number
      pop     EBX
      leave                  ; clears local variable space
      ret
```

5.10 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. NASM provides two directives—GLOBAL and EXTERN—to facilitate separate assembly of source modules. These two directives are discussed in the following sections. A simple example follows this discussion.

GLOBAL Directive

The GLOBAL directive makes the associated label(s) available to other modules of the program. The format is

```
global    label1, label2, ...
```

Almost any label can be made public. These include procedure names, memory variables, and equated labels, as shown in the following example:

```
global  error_msg, total, sample
              . . .
.DATA
error_msg   db   'Out of range!',0
total       dw   0
            . . .
.CODE
            . . .
sample:
            . . .
            ret
```

Microsoft and Borland assemblers use PUBLIC directive for this purpose.

EXTERN Directive

The `EXTERN` 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 object file that the linker will fill in later. The format is

```
extern    label1, label2, ...
```

where `label1` and `label2` are labels that are made public by a `GLOBAL` directive in some other module.

Example 5.9 *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 5.9. Notice that on line 18 we declare `string_length` as an externally defined procedure by using the `extern` directive.

Program 5.9 The `main` procedure defined in `module1.asm` calls the `sum` procedure defined in `module2.asm`

```
 1:  ;Multimodule program for string length   MODULE1.ASM
 2:  ;
 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:
 7:  BUF_SIZE  EQU  41   ; string buffer size
 8:  %include "io.mac"
 9:
10:  .DATA
11:  prompt_msg   db   "Please input a string: ",0
12:  length_msg   db   "String length is: ",0
13:
14:  .UDATA
15:  string1      resb   BUF_SIZE
16:
17:  .CODE
18:  extern   string_length
19:        .STARTUP
20:        PutStr  prompt_msg        ; request a string
```

```
      GetStr  string1,BUF_SIZE  ; read string input

      mov     EBX,string1    ; EBX = string pointer
      call    string_length  ; returns string length in AX
      PutStr  length_msg     ; display string length
      PutInt  AX
      nwln
done:
      .EXIT
```

Program 5.10 This module defines the `sum` procedure called by `main`

```
 1:  ;String length procedure         MODULE2.ASM
 2:  ;
 3:  ;        Objective: To write a procedure to compute string
 4:  ;                   length of a NULL-terminated string.
 5:  ;            Input: String pointer in EBX register.
 6:  ;           Output: Returns string length in AX.
 7:  %include "io.mac"
 8:
 9:  .CODE
10:  global string_length
11:  string_length:
12:        ; all registers except AX are preserved
13:        push    ESI            ; save ESI
14:        mov     ESI,EBX        ; ESI = string pointer
15:  repeat:
16:        cmp     byte [ESI],0   ; is it NULL?
17:        je      done           ; if so, done
18:        inc     ESI            ; else, move to next character
19:        jmp     repeat         ;       and repeat
20:  done:
21:        sub     ESI,EBX        ; compute string length
22:        mov     AX,SI          ; return string length in AX
23:        pop     ESI            ; restore ESI
24:        ret
```

Program 5.10 gives the `module2.asm` program listing. This module consists of a single procedure. By using the `global` directive, we make this procedure public (line 10) so that other modules can access it. The `string_length` procedure receives a pointer to a NULL-terminated string in EBX and returns the length of the string in AX. The procedure preserves all registers except for AX.

We can assemble each source code module separately producing the corresponding object files. We can then link the object files together to produce a single executable file. For example, using the NASM assembler, the following sequence of commands produces the executable file:

```
nasm  -f elf module1                          ← Produces module1.o
nasm  -f elf module2                          ← Produces module2.o
ld -s -o module module1.o module2.o io.o      ← Produces module
```

The above sequence assumes that you have io.o in your current directory.

5.11 Performance: Procedure Overheads

As we have seen in this chapter, procedures facilitate modular programming. However, there is a price to pay in terms of procedure invocation and return overheads. Parameter passing contributes additional overhead when procedures are used. In addition, allocation of storage for local variables can also significantly affect the performance. In this section, we quantify these overheads using the bubble sort and Fibonacci examples discussed before.

5.11.1 Procedure Overheads

This section reports the procedure call/return overhead on the performance of the assembly language bubble sort procedure discussed in this chapter. In the bubble sort procedure given in Program 5.5, swapping is done on lines 95–99. For convenience, we reproduce this code below:

```
swap:
      ; swap elements at [ESI] and [ESI+4]
      mov     [ESI+4],EAX    ; copy [ESI] in EAX to [ESI+4]
      mov     [ESI],EBX      ; copy [ESI+4] in EBX to [ESI]
      mov     EDX,UNSORTED   ; set status to UNSORTED
```

where ESI and ESI+4 point to the elements of the array to be interchanged.

To study the impact of procedure call and return overheads, we replaced the above code by a call to the following swap procedure:

```
swap_proc:
      mov     [ESI+4],EAX    ; copy [ESI] in EAX to [ESI+4]
      mov     [ESI],EBX      ; copy [ESI+4] in EBX to [ESI]
      mov     EDX,UNSORTED   ; set status to UNSORTED
      ret
```

Figure 5.14 shows the performance impact of the call/return overhead on the bubble sort procedure when run on a 2.4-GHz Pentium 4 system. Since the body of the procedure consists

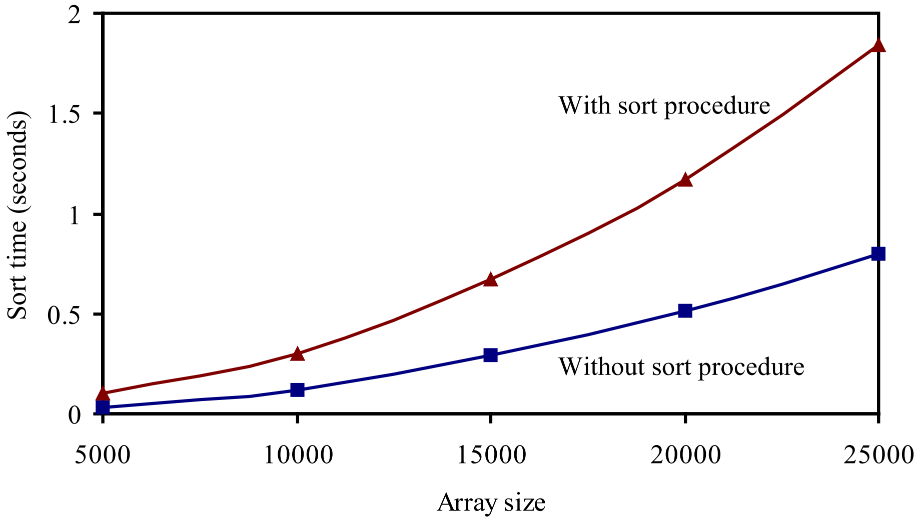


Figure 5.14 Performance impact of procedure call and return overheads on the bubble sort.

of the same three assembly language statements that it is replacing, the difference between the two lines can be directly attributed to the call/return overhead. It can be seen from this data that the overhead is substantial. For this program, the slowdown is by a factor of 2.3. Note that this overhead increases further if we were to pass parameters via the stack.

5.11.2 Local Variable Overhead

In this section, we use the Fibonacci example to study the performance impact of keeping local variables in registers as opposed to storing them on the stack.

The Fibonacci procedures given in Programs 5.7 and 5.8 are used to measure the execution time to compute the largest Fibonacci number that is less than or equal to 25,000. The results are shown in Figure 5.15. The x-axis represents the number of calls to the procedure (varied from 1 to 9 million). The execution time increases by a factor of 3.4 when the local variable storage is moved from registers to the stack. Because of this performance impact, compilers attempt to keep the local variables in registers.

5.12 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:ESP. In the Pentium

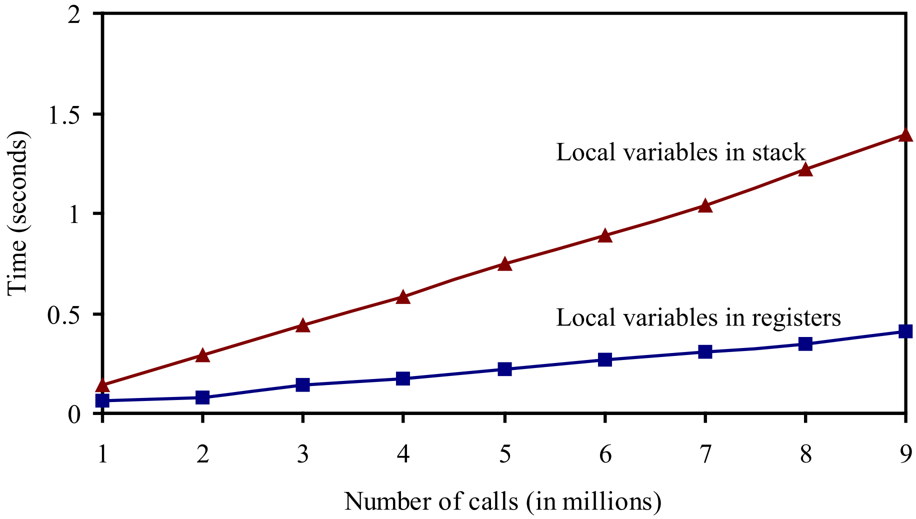


Figure 5.15 Local variable overhead: registers versus stack.

implementation, the stack grows toward lower memory addresses (i.e., grows downward). As discussed in Chapter 12, MIPS also implements the stack in a similar way.

The stack serves three main purposes: temporary storage of data, transfer of control during procedure calls, 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, passing 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
- EXTERN directive
- Frame pointer
- Local variables
- Parameter passing
- Parameter passing—register method
- Parameter passing—stack method
- GLOBAL directive
- Segment override
- Stack frame
- Stack operations
- Stack overflow
- Stack underflow
- Top-of-stack
- Variable number of parameters

5.13 Exercises

5–1 What are the defining characteristics of a stack?

5–2 Discuss the differences between a queue and a stack.

5–3 What is top-of-stack? How is it represented in the Pentium?

5–4 What is stack underflow? Which stack operation can potentially cause stack underflow?

5–5 What is stack overflow? Which stack operation can potentially cause stack overflow?

5–6 What are the main uses of the stack?

5–7 In Section 5.4.1 on page 123, we 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.

5–8 In the Pentium, can we invoke a procedure through the `call` instruction without the presence of a stack segment? Explain.

5–9 What are the two most common methods of parameter passing? Identify the circumstances under which each method is preferred.

5–10 What are the disadvantages of passing parameters via the stack?

5–11 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.

5–12 We have stated on page 134 that placing the code

```
push    EBP
mov     EBP,ESP
```

at the beginning of a procedure is good for program maintenance. Explain why.

5–13 In passing a variable number of parameters via the stack, why is it necessary to push the parameter count last?

5–14 Why are local variables of a procedure not mapped to the data segment?

5–15 How is storage space for local variables created in the stack?

5–16 A swap procedure can exchange two elements (pointed to by ESI and EDI) of an array using

```
xchg   EAX,[EDI]
xchg   EAX,[ESI]
xchg   EAX,[EDI]
```

The above code preserves the contents of the EAX 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 EAX using `push` and `pop` stack operations?

5–17 Verify that the following procedure is equivalent to the `string_length` procedure given in Section 5.10. Which procedure is better and why?

```
string_length1:
        push    EBX
        sub     AX,AX
repeat:
        cmp     [EBX],word 0
        je      done
        inc     AX
        inc     EBX
        jmp     repeat
done:
        pop     EBX
        ret
```

5.14 Programming Exercises

5–P1 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.

5–P2 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.

5–P3 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.

5–P4 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`.

5–P5 Extend the last exercise to handle 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`.

5–P6 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.

5–P7 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.

5–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 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.

5–P9 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.

5–P10 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`

5–P11 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.

5–P12 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 6

Addressing Modes

Objectives

- To discuss in detail various addressing modes supported by the Pentium processor
- To describe how arrays are implemented and manipulated in assembly language
- To show the performance impact of various addressing modes

In assembly language, specification of data required by instructions can be done in a variety of ways. In Chapter 4 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 6.2 describes these memory addressing modes in detail and Section 6.3 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 6.4. This section also gives some examples to illustrate the use of the addressing modes in processing one- and two-dimensional arrays. Section 6.5 shows how the performance is affected by the various addressing modes.

6.1 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 what we called the register addressing mode in Chapter 4. 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.

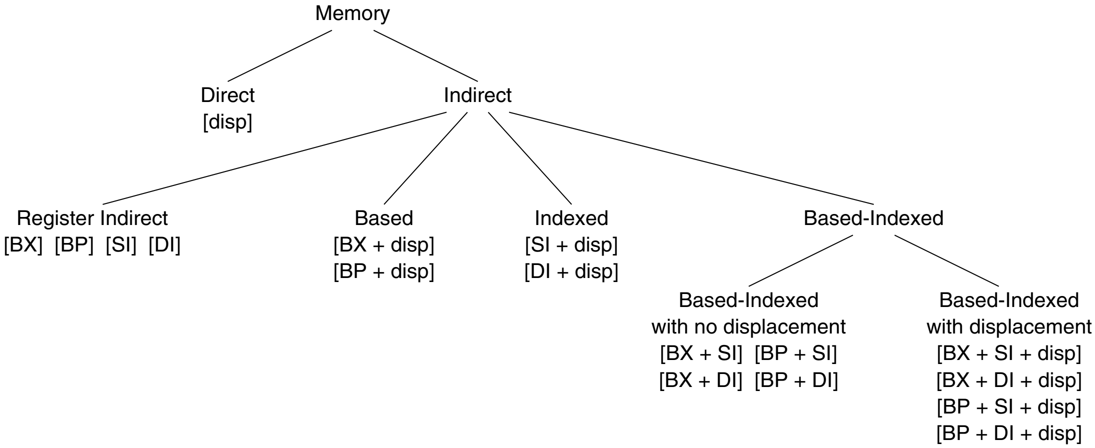


Figure 6.1 Memory addressing modes for 16-bit addresses.

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 4, 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 4. 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.

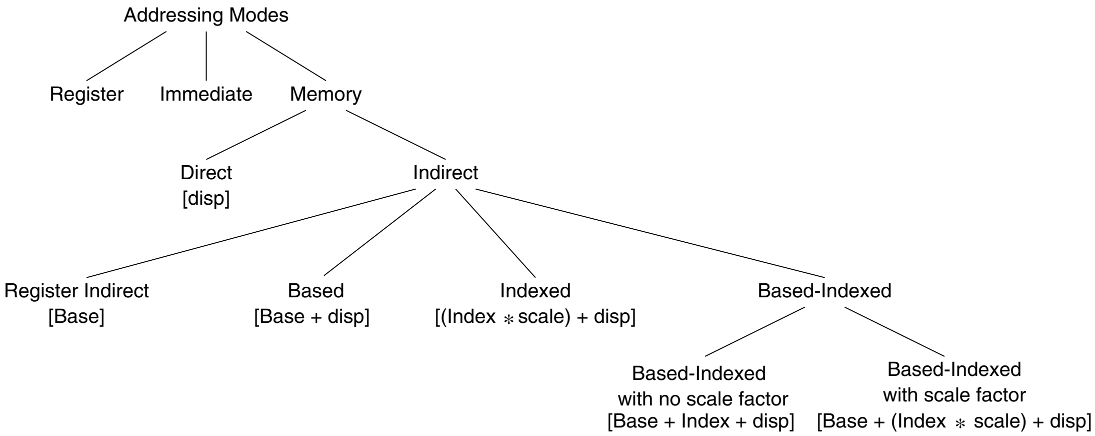


Figure 6.2 Addressing modes of the Pentium for 32-bit addresses.

6.2 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 6.1 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 6.2 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 6.1. How does the processor know whether to use 16- or 32-bit addressing? As discussed in Chapter 3, it uses the D bit in the CS segment descriptor to determine if the address is 16 or 32 bits long (see page 56). It is, however, possible to override these defaults. The Pentium provides two size override prefixes:

Table 6.1 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

66H	Operand size override prefix
67H	Address size override prefix

By using these prefixes, we can mix 16- and 32-bit data and addresses. Remember that our assembly language programs use 32-bit data and addresses. This, however, does not restrict us from using 16-bit data and addresses. For example, when we write

```
mov    EAX,123
```

the assembler generates the following machine language code:

```
B8 0000007B
```

However, when we use a 16-bit operand as in

```
mov    AX,123
```

the following code is generated by the assembler:

```
66 | B8 007B
```

The assembler automatically inserts the operand size override prefix (66H). Similarly, we can use the 16-bit addresses. For instance, consider the following example:

```
mov    EAX,[BX]
```

The assembler automatically inserts the address size override prefix (67H) as shown below:

```
67 | 8B 07
```

It is also possible to mix both override prefixes as demonstrated by the following example. The assembly language statement

```
mov    AX,[BX]
```

causes the assembler to insert both operand and address size override prefixes:

```
66 | 67 | 8B 07
```

6.2.1 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 EBX 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 6.3.

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.

6.2.2 Indexed Addressing

In this addressing mode, the effective address is computed as

$$(\text{Index} * \text{scale factor}) + \text{signed displacement}.$$

For 16-bit addresses, no scaling factor is allowed (see Table 6.1 on page 170). 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.

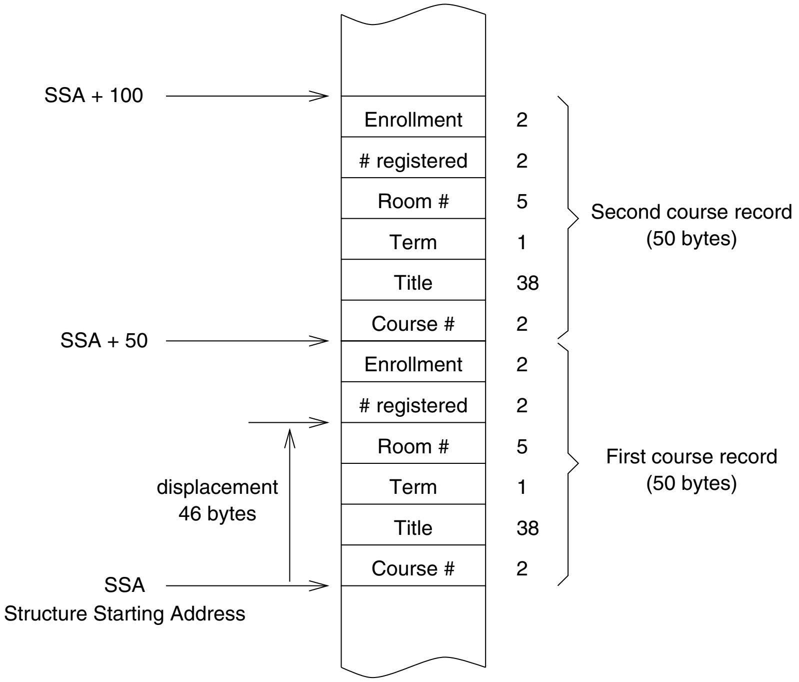


Figure 6.3 Course record layout in memory.

The following are valid instructions using the indexed addressing mode to specify one of the operands.

```
add     EAX,[EDI+20]
mov     EAX,[marks_table+ESI*4]
add     EAX,[table1+ESI]
```

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, ESI 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, ESI register should have 36 to refer to the tenth element. By using the scale factor, we avoid such byte counting.

6.2.3 Based-Indexed Addressing

Based-indexed addressing mode comes in two flavors: with or without the scale factor. These two addressing modes are discussed next.

Based-Indexed with No Scale Factor

In this addressing mode, the effective address is computed as

$$\text{Base} + \text{Index} + \text{signed displacement.}$$

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 useful 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 EBX points to `table1`, which consists of four-byte elements, we can use the code

```
mov     EAX,[EBX+ESI]
cmp     EAX,[EBX+ESI+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

$$\text{Base} + (\text{Index} * \text{scale factor}) + \text{signed displacement.}$$

This addressing mode provides an efficient indexing mechanism into a two-dimensional array when the element size is 2, 4, or 8 bytes.

6.3 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 6.1 *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 20 to 28). It accepts only nonnegative numbers. Entering a negative number terminates the input (lines 24 and 25).

The main procedure passes the array pointer and its size (lines 30 to 34) 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 26 and 41.

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 6.1, follows the pseudocode.

Program 6.1 Insertion sort

```
 1:  ;TITLE    Sorting an array by insertion sort    INS_SORT.ASM
 2:  ;
 3:  ;        Objective: To sort an integer array using insertion sort.
 4:  ;            Input: Requests numbers to fill array.
 5:  ;           Output: Displays sorted array.
 6:  %include "io.mac"
 7:
 8:  .DATA
 9:  MAX_SIZE       EQU 100
10:  input_prompt   db  "Please enter input array: "
11:                 db  "(negative number terminates input)",0
```

```
out_msg        db   "The sorted array is:",0

.UDATA
array          resd  MAX_SIZE

.CODE
       .STARTUP
       PutStr  input_prompt ; request input array
       mov     EBX,array
       mov     ECX,MAX_SIZE
array_loop:
       GetLInt EAX          ; read an array number
       cmp     EAX,0        ; negative number?
       jl      exit_loop    ; if so, stop reading numbers
       mov     [EBX],EAX    ; otherwise, copy into array
       add     EBX,4        ; increment array address
       loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
       mov     EDX,EBX      ; EDX keeps the actual array size
       sub     EDX,array    ; EDX = array size in bytes
       shr     EDX,2        ; divide by 4 to get array size
       push    EDX          ; push array size & array pointer
       push    array
       call    insertion_sort
       PutStr  out_msg      ; display sorted array
       nwln
       mov     ECX,EDX
       mov     EBX,array
display_loop:
       PutLInt  [EBX]
       nwln
       add     EBX,4
       loop    display_loop
done:
       .EXIT

;-----------------------------------------------------------
; This procedure receives a pointer to an array of integers
; and the array size via the stack. The array is sorted by
; using insertion sort. All registers are preserved.
;-----------------------------------------------------------
%define   SORT_ARRAY   EBX
insertion_sort:
       pushad                ; save registers
```

```
56:         mov     EBP,ESP
57:         mov     EBX,[EBP+36]   ; copy array pointer
58:         mov     ECX,[EBP+40]   ; copy array size
59:         mov     ESI,4          ; array left of ESI is sorted
60:  for_loop:
61:         ; variables of the algorithm are mapped as follows.
62:         ; EDX = temp, ESI = i, and EDI = j
63:         mov     EDX,[SORT_ARRAY+ESI] ; temp = array[i]
64:         mov     EDI,ESI        ; j = i-1
65:         sub     EDI,4
66:  while_loop:
67:         cmp     EDX,[SORT_ARRAY+EDI]  ; temp < array[j]
68:         jge     exit_while_loop
69:         ; array[j+1] = array[j]
70:         mov     EAX,[SORT_ARRAY+EDI]
71:         mov     [SORT_ARRAY+EDI+4],EAX
72:         sub     EDI,4          ; j = j-1
73:         cmp     EDI,0          ; j >= 0
74:         jge     while_loop
75:  exit_while_loop:
76:         ; array[j+1] = temp
77:         mov     [SORT_ARRAY+EDI+4],EDX
78:         add     ESI,4          ; i = i+1
79:         dec     ECX
80:         cmp     ECX,1          ; if ECX = 1, we are done
81:         jne     for_loop
82:  sort_done:
83:         popad                  ; restore registers
84:         ret     8
```

Since the sort procedure does not return any value to the main program in registers, we can use pushad (line 55) and popad (line 83) to save and restore registers. As pushad saves all eight registers on the stack, the offset is appropriately adjusted to access the array size and array pointer parameters (lines 57 and 58).

The while loop is implemented by lines 66 to 75, and the for loop is implemented by lines 60 to 81. Note that the array pointer is copied to the EBX (line 57), and line 53 assigns a convenient label to this. We have used the based-indexed addressing mode on lines 63, 67, and 70 without any displacement and on lines 71 and 77 with displacement. Based addressing is used on lines 57 and 58 to access parameters from the stack.

Example 6.2 *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 $\leq$ 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 6.2. The main procedure is similar to that in the last example. In the binary search procedure, 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 EBX. Register SI keeps the middle index value.

Program 6.2 Binary search

```
;Binary search of a sorted integer array   BIN_SRCH.ASM
;
;        Objective: To implement binary search of a sorted
;                   integer array.
;            Input: Requests numbers to fill array and a
;                   number to be searched for from user.
;           Output: Displays the position of the number in
;                   the array if found; otherwise, not found
;                   message.
%include "io.mac"

.DATA
MAX_SIZE       EQU 100
input_prompt   db  "Please enter input array (in sorted order): "
               db  "(negative number terminates input)",0
query_number   db  "Enter the number to be searched: ",0
out_msg        db  "The number is at position ",0
not_found_msg  db  "Number not in the array!",0
query_msg      db  "Do you want to quit (Y/N): ",0

.UDATA
array          resw  MAX_SIZE

.CODE
        .STARTUP
        PutStr  input_prompt ; request input array
        nwln
        sub     ESI,ESI      ; set index to zero
        mov     CX,MAX_SIZE
array_loop:
        GetInt  AX           ; read an array number

        cmp     AX,0             ; negative number?
        jl      exit_loop        ; if so, stop reading numbers
        mov     [array+ESI*2],AX ; otherwise, copy into array
        inc     ESI          ; increment array index
        loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
read_input:
        PutStr  query_number ; request number to be searched for
        GetInt  AX           ; read the number
        push    AX           ; push number, size & array pointer
        push    SI
        push    array
        call    binary_search
        ; binary_search returns in AX the position of the number
        ; in the array; if not found, it returns 0.
        cmp     AX,0         ; number found?
        je      not_found    ; if not, display number not found
        PutStr  out_msg      ; else, display number position
        PutInt  AX
        jmp     user_query
not_found:
        PutStr  not_found_msg
user_query:
        nwln
        PutStr  query_msg    ; query user whether to terminate
        GetCh   AL           ; read response
        cmp     AL,’Y’       ; if response is not ’Y’
        jne     read_input   ; repeat the loop
done:                        ; otherwise, terminate program
        .EXIT

;-----------------------------------------------------------
; This procedure receives a pointer to an array of integers,
; the array size, and a number to be searched via the stack.
; It returns in AX the position of the number in the array
; if found; otherwise, returns 0.
; All registers, except AX, are preserved.
;-----------------------------------------------------------
binary_search:
       enter   0,0
       push    EBX
       push    ESI
       push    CX
       push    DX
       mov     EBX,[EBP+8]   ; copy array pointer
       mov     CX,[EBP+12]   ; copy array size
       mov     DX,[EBP+14]   ; copy number to be searched
       xor     AX,AX         ; lower = 0
       dec     CX            ; upper = size-1
while_loop:
       cmp     AX,CX         ;lower > upper?
       ja      end_while
       sub     ESI,ESI
       mov     SI,AX         ; middle = (lower + upper)/2
       add     SI,CX
       shr     SI,1
       cmp     DX,[EBX+ESI*2]    ; number = array[middle]?
       je      search_done
       jg      upper_half
lower_half:
       dec     SI            ; middle = middle-1
       mov     CX,SI         ; upper = middle-1
       jmp     while_loop
```

```
 96:  upper_half:
 97:          inc     SI              ; middle = middle+1
 98:          mov     AX,SI           ; lower = middle+1
 99:          jmp     while_loop
100:  end_while:
101:          sub     AX,AX           ; number not found (clear AX)
102:          jmp     skip1
103:  search_done:
104:          inc     SI              ; position = index+1
105:          mov     AX,SI           ; return position
106:  skip1:
107:          pop     DX              ; restore registers
108:          pop     CX
109:          pop     ESI
110:          pop     EBX
111:          leave
112:          ret     8
```

Since the binary search procedure returns a value in the AX register, we cannot use the `pusha` instruction as in the last example. On line 89, we use a scale factor of two to convert the index value in SI to byte count. Also, a single comparison (line 89) 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 105).

6.4 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 6.4.2.

6.4.1 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 (4 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 $10 * 4 = 40$ 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    resd    10
```

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 6.4. 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 $5 * 4 = 20$ as the displacement value into the `test_marks` array. Section 6.4.3 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.

6.4.2 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.

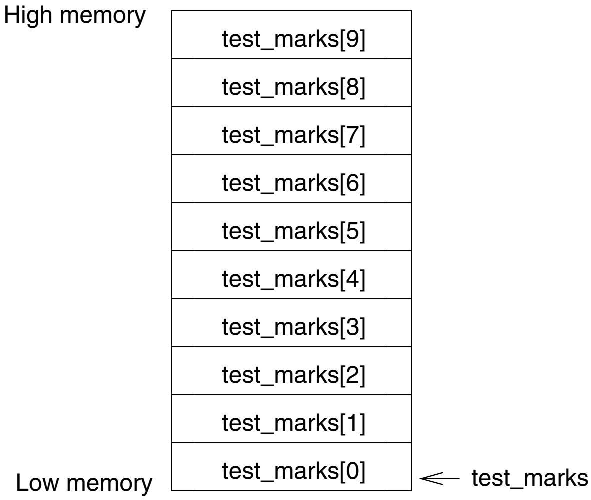


Figure 6.4 One-dimensional array storage representation.

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:

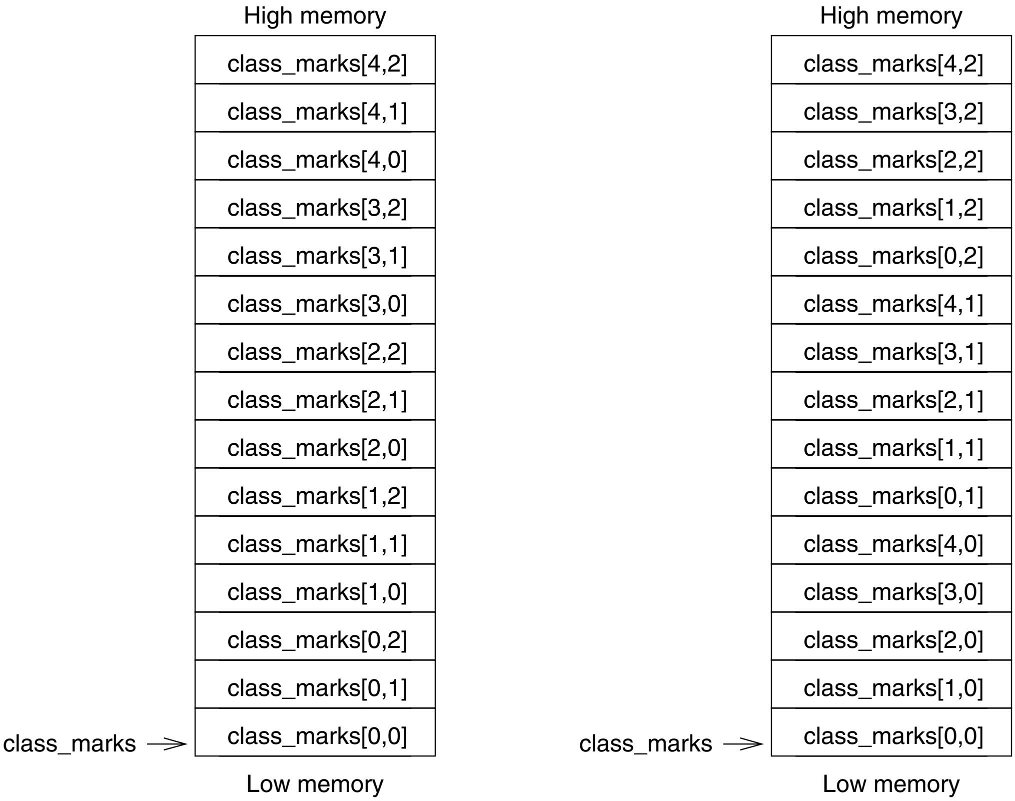
- Order the array elements row-by-row, starting with the first row,
- Order the array elements column-by-column, starting with the first column.

The first method, called the *row-major ordering*, is shown in Figure 6.5a. Row-major ordering is used in most high-level languages including C. The other method, called the *column-major ordering*, is shown in Figure 6.5b. 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 use a high-level language, we really do not have to bother about the storage representation. Access to arrays is provided by subscripts: one subscript for each dimension of the array. However, when using the 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    resd    5*3
```



(a) Row−major order

(b) Column−major order

Figure 6.5 Two-dimensional array storage representation.

This statement simply allocates the 60 bytes required to store the array. Now we need a formula to translate row and column subscripts to the corresponding displacement. In the 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 $(3 * 3 + 1) * 4 = 40$. The next section gives an example to illustrate how two-dimensional arrays are manipulated.

6.4.3 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 6.3 *Finding the sum of a one-dimensional array.*
This example shows how one-dimensional arrays can be manipulated. Program 6.3 finds the sum of the `test_marks` array and displays the result.

Program 6.3 Computing the sum of a one-dimensional array

```
;Sum of a long integer array             ARAY_SUM.ASM
;
;        Objective: To find sum of all elements of an array.
;            Input: None.
;           Output: Displays the sum.
%include "io.mac"

.DATA
test_marks     dd  90,50,70,94,81,40,67,55,60,73
NO_STUDENTS    EQU ($-test_marks)/4     ; number of students
sum_msg        db  'The sum of test marks is: ',0

.CODE
        .STARTUP
        mov     ECX,NO_STUDENTS   ; loop iteration count
        sub     EAX,EAX          ; sum = 0
        sub     ESI,ESI          ; array index = 0
add_loop:
        mov     EBX,[test_marks+ESI*4]
        PutLInt EBX
        nwln
        add     EAX,[test_marks+ESI*4]
        inc     ESI
        loop    add_loop

        PutStr  sum_msg
        PutLInt EAX
        nwln
        .EXIT
```

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 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 value by four gives the number of elements in the array. The indexed addressing mode with a scale factor of four is used on lines 19 and 22. 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 6.4 *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 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 6.4.

Program 6.4 Finding the sum of a column in a two-dimensional array

```
;Sum of a column in a 2-dimensional array  TEST_SUM.ASM
;
;        Objective: To demonstrate array index manipulation
;                   in a two-dimensional array of integers.
;            Input: None.
;           Output: Displays the sum.
%include "io.mac"

.DATA
NO_ROWS        EQU  5
NO_COLUMNS     EQU  3
NO_ROW_BYTES   EQU  NO_COLUMNS * 2  ; number of bytes per row
class_marks    dw   90,89,99
               dw   79,66,70
               dw   70,60,77
               dw   60,55,68
               dw   51,59,57

sum_msg        db   "The sum of the last test marks is: ",0

.CODE
        .STARTUP
        mov     ECX,NO_ROWS  ; loop iteration count
```

```
24:         sub     AX,AX        ; sum = 0
25:         ; ESI = index of class_marks[0,2]
26:         sub     EBX,EBX
27:         mov     ESI,NO_COLUMNS-1
28: sum_loop:
29:         add     AX,[class_marks+EBX+ESI*2]
30:         add     EBX,NO_ROW_BYTES
31:         loop    sum_loop
32:
33:         PutStr  sum_msg
34:         PutInt  AX
35:         nwln
36: done:
37:         .EXIT
```

To access individual test marks, we use based-indexed addressing with a displacement on line 29. 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 26), `NO_ROW_BYTES` is added in the loop body (line 30). The ESI register is used as the column index. This works for row-major ordering.

6.5 Performance: Usefulness of Addressing Modes

The objective of this section is to show the performance impact of the various memory addressing modes.

Advantage of Based-Indexed Mode

This experiment shows the performance advantage of based-indexed addressing mode over the register indirect addressing mode. In Program 6.1, we used based-indexed addressing with and without displacement. For example, the two statements on lines 70 and 71 are equivalent to

```
mov     EAX,[EBX+EDI]
mov     [EBX+EDI+4],EAX
```

By using a displacement value of 4, we could use the same two registers to access the next element in the array. We would not have this kind of flexibility if we use only the indirect addressing mode. To see the relative performance, we have rewritten the insertion sort procedure discussed in Section 6.3 with only direct and register indirect addressing modes. We, however, have not modified how the parameters are accessed from the stack.

Program 6.5 Insertion sort procedure using only the indirect addressing mode

```
;------------------------------------------------------------
; This procedure receives a pointer to an array of integers
; and the array size via the stack. The array is sorted by
; using insertion sort. All registers are preserved.
;------------------------------------------------------------
;%define   SORT_ARRAY   EBX
ins_sort:
       pushad                ; save registers
       mov     EBP,ESP
       mov     EBX,[EBP+36]  ; copy array pointer
       mov     ECX,[EBP+40]  ; copy array size
       mov     ESI,EBX       ; array left of ESI is sorted
       add     ESI,4
for_loop:
       ; variables of the algorithm are mapped as follows.
       ; EDX = temp, ESI = i, and EDI = j
       mov     EDX,[ESI]     ; temp = array[i]
       mov     EDI,ESI       ; j = i-1
       sub     EDI,4
while_loop:
       cmp     EDX,[EDI]     ; temp < array[j]
       jge     exit_while_loop
       ; array[j+1] = array[j]
       mov     EAX,[EDI]
       mov     EBX,EDI
       add     EBX,4
       mov     [EBX],EAX
       sub     EDI,4         ; j = j-1
       cmp     EDI,[EBP+36]  ; j >= 0
       jge     while_loop
exit_while_loop:
       ; array[j+1] = temp
       add     EDI,4
       mov     [EDI],EDX
       add     ESI,4         ; i = i+1
       dec     ECX
```

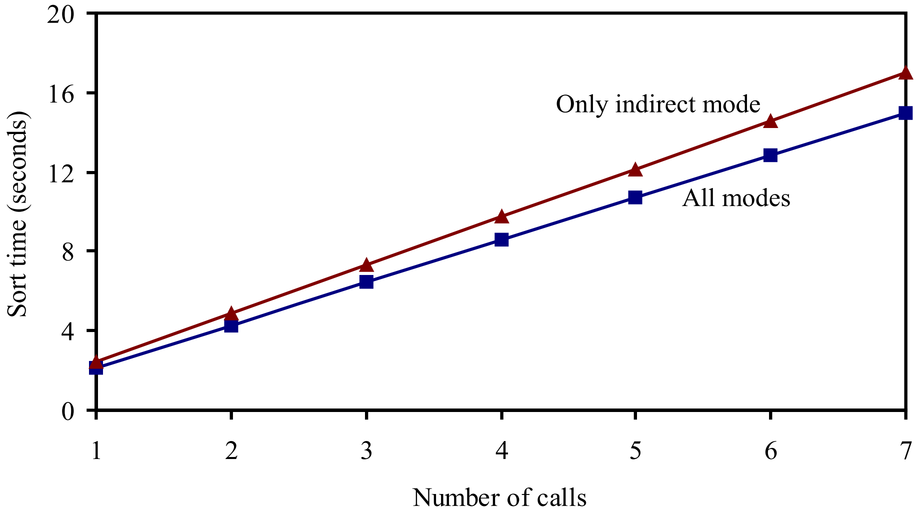


Figure 6.6 Performance impact of using only indirect addressing mode on the insertion sort.

```
        cmp     ECX,1           ; if ECX = 1, we are done
        jne     for_loop
sort_done:
        popad                   ; restore registers
        ret
```

The performance implications of these changes are shown in Figure 6.6. This plot gives the sort time to sort an array of 50,000 elements on a 2.4-GHz Pentium 4 system. The x-axis gives the number of times the sort procedure is called and the y-axis gives the corresponding sort time. The performance of the procedure given in Program 6.1 is represented by the "All modes" line while the other line represents the performance of the modified version given here. The data presented in this figure show that using only direct and register indirect addressing modes deteriorates performance of the insertion sort by about 13%! For example, when the array is sorted seven times, there is a 2-second difference in the sort time.

Impact of Scale Factor

The goal of this experiment is to show the impact of scale factor on the performance. To quantify this impact, we have written the insertion sort procedure using the based-indexed addressing mode that uses a scale factor (see the program listing below).

Program 6.6 Insertion sort procedure with the scale factor

```
;------------------------------------------------------------
; This procedure receives a pointer to an array of integers
; and the array size via the stack. The array is sorted by
; using insertion sort. All registers are preserved.
;------------------------------------------------------------
%define   SORT_ARRAY   EBX
ins_sort:
       pushad                ; save registers
       mov     EBP,ESP
       mov     EBX,[EBP+36]  ; copy array pointer
       mov     ECX,[EBP+40]  ; copy array size
       mov     ESI,1         ; array left of ESI is sorted
for_loop:
       ; variables of the algorithm are mapped as follows.
       ; DX = temp, ESI = i, and EDI = j
       mov     EDX,[SORT_ARRAY+ESI*4] ; temp = array[i]
       mov     EDI,ESI       ; j = i-1
       dec     EDI
while_loop:
       cmp     EDX,[SORT_ARRAY+EDI*4] ; temp < array[j]
       jge     exit_while_loop
       ; array[j+1] = array[j]
       mov     EAX,[SORT_ARRAY+EDI*4]
       mov     [SORT_ARRAY+EDI*4+4],EAX
       dec     EDI           ; j = j-1
       cmp     EDI,0         ; j >= 0
       jge     while_loop
exit_while_loop:
       ; array[j+1] = temp
       mov     [SORT_ARRAY+EDI*4+4],EDX
       inc     ESI           ; i = i+1
       dec     ECX
       cmp     ECX,1         ; if ECX = 1, we are done
       jne     for_loop
sort_done:
       popad                 ; restore registers
       ret
```

As shown in Program 6.6, the ESI and EDI register now hold the array subscript as opposed to byte count. While this improves program readability, we pay in terms of performance as shown in Figure 6.7. The sort times represent the time to sort a 50,000-element array on a

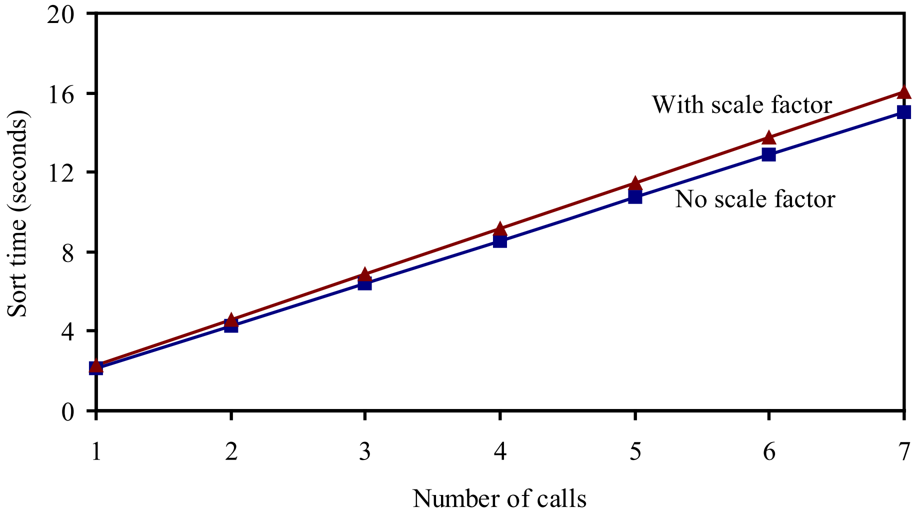


Figure 6.7 Performance impact of using scale factor on the insertion sort.

2.4-GHz Pentium 4 system. The x-axis gives the number of times the insertion sort procedure is called. As shown in this plot, the sort time increases by about 7% when we use the scale factor. This could make a significant difference in the execution times of some programs. For example, if this sort procedure is called seven times, the execution time increases by a second.

6.6 Summary

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 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.

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 size override prefix
- Based addressing mode
- Based-indexed addressing mode
- Binary search
- Column-major order
- Indexed addressing mode
- Insertion sort
- Location counter
- Multidimensional arrays
- One-dimensional arrays
- Operand size override prefix
- Row-major order

6.7 Exercises

6–1 Discuss the restrictions imposed by the immediate addressing mode.

6–2 Where (i.e., in which segment) are the data, specified by the immediate addressing mode, stored?

6–3 Describe all the 16-bit addressing modes that you can use to specify an operand that is located in memory.

6–4 Describe all the 32-bit addressing modes that you can use to specify an operand that is located in memory.

6–5 When is it necessary to use the segment override prefix?

6–6 When is it necessary to use the address size override prefix?

6–7 Is there a fundamental difference between the based and indexed addressing modes?

6–8 What additional flexibility does the based-indexed addressing mode have over based or indexed addressing modes?

6–9 Given the following declaration of table1

```
table1    resw    10
```

fill in the blanks in the following code:

```
mov    ESI, _______ ; ESI = displacement of 5th element
                    ; (i.e., table1[4] in C)
mov    AX,[table1+ESI]
cmp    AX, _______  ; compare 5th and 4th elements
```

6–10 What is the difference between row-major and column-major orders for storing multidimensional arrays in memory?

6–11 In manipulating multidimensional arrays in assembly language, why is it necessary to know their underlying storage representation?

6–12 How is the `class_marks` array in Program 6.4 stored in memory: row-major or column-major order? How would you change the `class_marks` declaration so that we can store it in the other order?

6–13 Assume that a two-dimensional array is stored in column-major order and its subscripts begin with 0. Derive a formula for the displacement (in bytes) of the element in row i and column j.

6–14 Suppose that 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].

6.8 Programming Exercises

6–P1 What modifications would you make to the insertion sort procedure discussed in Section 6.3 to sort the array in descending order? Make the necessary modifications to the program and test it for correctness.

6–P2 Modify Program 6.3 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.

6–P3 Modify Program 6.4 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 5 tests. Report error when exceeding these limits.

6–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**.

6–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 read these numbers.

6–P6 Modify the program of the last exercise to work on matrices stored in the column-major order.

6–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

$$\begin{bmatrix} 12 & 23 & 34 & 45 \\ 34 & 45 & 56 & 67 \\ 56 & 67 & 78 & 89 \\ 78 & 89 & 90 & 10 \end{bmatrix}.$$

6–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 in our example).

6–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

$$\begin{bmatrix} 34 & 56 & 78 & 12 \\ 45 & 67 & 89 & 23 \\ 56 & 78 & 90 & 34 \\ 67 & 89 & 10 & 45 \end{bmatrix}.$$

6–P10 Generalize the last exercise to cyclically permute by a user-specified number of elements.

6–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

6–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.

6–P13 If we are given a square matrix (i.e., a matrix with equal number of rows and 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.

Chapter 7

Arithmetic Flags and Instructions

Objectives

- To discuss how status flags are affected by arithmetic and logic instructions
- To present the multiplication and division instructions
- To introduce multiword arithmetic operations

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 helps us understand how some of the conditional jump instructions are executed. The next section deals with multiplication and division instructions. The instruction set includes multiplication and division instructions for both signed and unsigned integers. The following section presents some examples to illustrate the use of the instructions discussed in this chapter. The instruction set supports arithmetic operations on 32-bit values. For applications that use more than 32 bits, we need to perform multiword arithmetic operations. These operations are discussed in Section 7.4. The chapter concludes with a summary.

7.1 Status Flags

Six flags in the flags register, described in Chapter 3, are used to monitor the outcome of 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. 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     EAX,55H  ; ZF is still 0
sub     EAX,55H  ; result is zero
                 ; Thus, ZF is set (ZF = 1)
push    EBX      ; ZF remains 1
mov     EBX,EAX  ; ZF remains 1
pop     EDX      ; ZF remains 1
mov     ECX,0    ; ZF remains 1
inc     ECX      ; 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.

7.1.1 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 is 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     EAX,1
dec     EAX
```

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; it performs subtraction 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     EAX,EBX
```

Counting to a Preset Value: Another important use of the zero flag is shown below:

```
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 shown below:

```
      sub     EAX,EAX      ; EAX = 0 (EAX stores sum)
      mov     EDX,M
outer_loop:
      mov     ECX,N
inner_loop:
      inc     EAX
      loop    inner_loop
      dec     EDX
      jnz     outer_loop
exit_loops:
      mov     sum,EAX
```

In this example code, both M and N are assumed to be greater than or equal to 1. The inner loop count is placed in the ECX 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 EDX register to one and using the code

```
inc    EDX
cmp    EDX,M
jle    outer_loop
```

in place of the `dec`/`jnz` instruction combination.

7.1.2 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 following 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 for each data size 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    EAX,12AEH    ;EAX = 4782D
sub    EAX,12AFH    ;EAX = 4782D - 4783D
```

Executing the above code sets 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    EAX,0
dec    EAX
```

does not set the carry flag as we might expect because `inc` and `dec` instructions do not affect the carry flag.

Related Instructions

The following conditional jumps instructions test the carry flag:

`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 225.

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 4), 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.

7.1.3 The Overflow Flag

The overflow flag 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 ranges of the numbers are 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
```

sets 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 contains 80H, the correct result if the two 8-bit operands are treated as unsigned numbers. But it is 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.

```
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

The following conditional jumps instructions test the overflow flag:

`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 later chapters.

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.

7.1.4 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    EAX,15
add    EAX,97
```

clears 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 is zero).

The result produced by

```
mov    EAX,15
sub    EAX,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 EAX 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

The following conditional jumps test the sign flag:

`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    ECX,M
for_loop:
         . . .
      <loop body>
         . . .
      dec    ECX
      jns    for_loop
```

If we do not use the `jns` instruction, we have to use

```
      cmp    ECX,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 8.3 on page 244).

7.1.5 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:

$$\begin{array}{rl}
 & \quad\;\;\texttt{1} \leftarrow \text{carry generated from lower to upper nibble} \\
\texttt{43D} & = \texttt{00101011B} \\
\underline{\texttt{94D}} & = \underline{\texttt{01011110B}} \\
\texttt{137D} & = \texttt{10001001B}
\end{array}$$

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 four 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 Chapter 11.

7.1.6 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 since 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

The following conditional jumps instructions test the parity flag:

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 assumed to be the leftmost bit:

```
A = 01000001
```

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
```

7.1.7 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 C for information on debuggers).

Example 7.1 *Add/subtract example.*

Table 7.1 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.

Table 7.1 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

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). Since the result −129 is smaller than −128, 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 7.2 *A compare example.*

This example shows how the status flags are affected by the compare instruction discussed in Chapter 4 on page 82. Table 7.2 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.

Table 7.2 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

7.2 Arithmetic Instructions

For the sake of completeness, we list the complete set of arithmetic instructions below:

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 already looked at the addition and subtraction instructions in Chapter 4. Here we discuss the remaining instructions. There are a few other arithmetic instructions that operate on decimal and BCD numbers. Details on these instructions can be found in Chapter 11.

7.2.1 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:

$$\begin{array}{ccccc} 11111111 & \times & 11111111 & = & 11111110\,11111111. \\ \text{(255D)} & & \text{(255D)} & & \text{(65025D)} \end{array}$$

   Similarly, multiplication of two 16-bit numbers requires 32 bits to store the result, and two 32-bit numbers requires 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 represents a signed number? In this case, FFH represents -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.

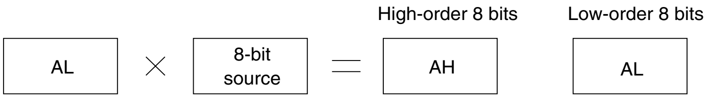
Thus, the instruction set provides two multiplication instructions: one for unsigned numbers and the other for signed numbers. We first discuss the unsigned multiplication instruction, which has the format

```
mul     source
```

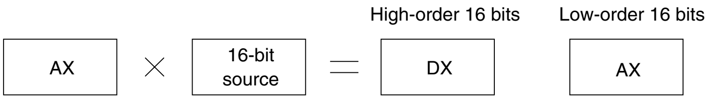
The source operand can be in a general-purpose register or in memory. Immediate operand specification is not allowed. Thus,

```
mul     10          ; invalid
```

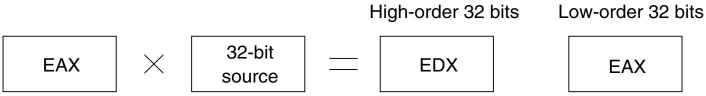
is an invalid instruction. The mul instruction works on 8-, 16-, and 32-bit unsigned numbers. But, where is the second operand? The instruction assumes that it is in the accumulator register. If the source operand is a byte, it is multiplied by the contents of the AL register. The 16-bit result is placed 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
```

clears both the carry and overflow flags as the result is 250, which can be stored in the AL register; in this case, the AH register contains `00000000`. On the other hand, executing

```
mov    AL,10
mov    DL,26
mul    DL
```

sets 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.

[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*.

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

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).

7.2.2 Division Instructions

The division operation is even more complicated than multiplication for two reasons:

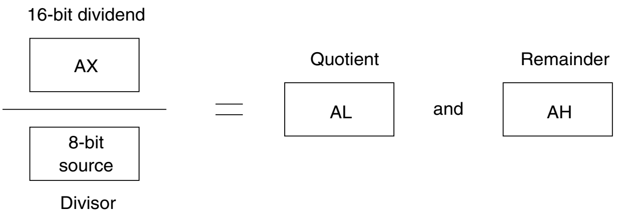
1. Division generates two result components: a quotient and a remainder.
2. In multiplication, by using double-length registers, overflow never occurs. In division, divide overflow is a real possibility. The processor generates a special software interrupt when a divide overflow occurs.

As with the multiplication, two versions are provided to work on unsigned and signed numbers.

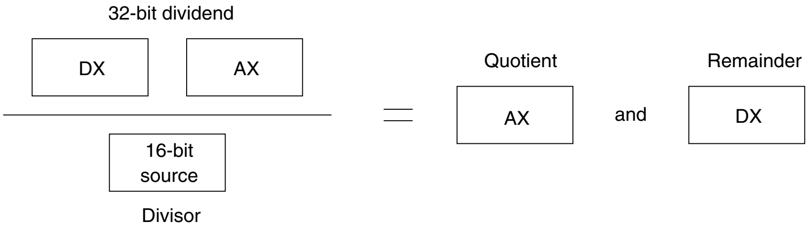
```
div     source  (unsigned)
idiv    source  (signed)
```

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 is updated. We first consider the unsigned version.

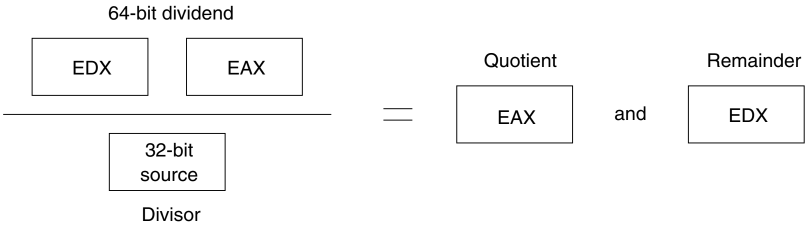
If the source operand is a byte, the dividend is assumed to be in the AX register and 16 bits long. After division, the quotient is returned in the AL register and the remainder in the AH register as shown below:



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 7.3 *Eight-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 7.4 *Sixteen-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`, several instructions are provided:

`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 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 instruction set also includes 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 7.5 *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 7.6 *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
```

performs this division and leaves −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.

7.3 Illustrative 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 the AL register;

PutInt8 Displays a signed 8-bit integer that is in the AL register.

The following two subsections describe these procedures in detail.

7.3.1 PutInt8 Procedure

Our objective here is to write a procedure that displays the signed 8-bit integer in AL. 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 7.1 The `PutInt8` procedure to display an 8-bit signed number (in `getput.asm` file)

```
;-----------------------------------------------------------
;PutInt8 procedure displays a signed 8-bit integer that is
;in AL register. All registers are preserved.
;-----------------------------------------------------------
PutInt8:
      enter   3,0          ; reserves 3 bytes of buffer space
      push    AX
      push    BX
      push    ESI
      test    AL,80H       ; negative number?
      jz      positive
negative:
      PutCh   '-'          ; sign for negative numbers
      neg     AL           ; convert to magnitude
positive:
```

```
16:        mov     BL,10          ; divisor  = 10
17:        sub     ESI,ESI        ; ESI = 0 (ESI 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     [EBP+ESI-3],AH ; copy into the buffer
24:        inc     ESI
25:        cmp     AL,0           ; quotient = zero?
26:        jne     repeat1        ; if so, display the number
27:  display_digit:
28:        dec     ESI
29:        mov     AL,[EBP+ESI-3]; display digit pointed by ESI
30:        PutCh   AL
31:        jnz     display_digit ; if ESI<0, done displaying
32:  display_done:
33:        pop     ESI             ; restore registers
34:        pop     BX
35:        pop     AX
36:        leave                  ; clears local buffer space
37:        ret
```

The `PutInt8` procedure, shown in Program 7.1, 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 on lines 10 and 11

  ```
  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 is in AX when the divisor is an 8-bit number.
- Conversion to ASCII character representation is done on line 22 using

```
        add     AH,'0'
```

- ESI is used as the index into the buffer, which starts at [EBP − 3]. Thus, [EBP + ESI − 3] points to the current byte in the buffer (lines 23 and 29).
- The repeat while condition (index = 0) is tested by

```
        jnz     display_digit
```

  on line 31.

7.3.2 GetInt8 Procedure

The `GetInt8` procedure reads a signed integer and returns the number in AL. 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 shown below:

```
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 the sign}
        end if
    end GetInt8
```

Program 7.2 The `GetInt8` procedure to read a signed 8-bit integer (in `getput.asm` file)

```
;-----------------------------------------------------------
;GetInt8 procedure reads an integer from the keyboard and
;stores its equivalent binary in AL register. If the number
;is within -128 and +127 (both inclusive), CF is cleared;
;otherwise, CF is set to indicate out-of-range error.
;No error check is done to see if the input consists of
;digits only. All registers are preserved except for AX.
;-----------------------------------------------------------
GetInt8:
      push    BX             ; save registers
      push    ECX
      push    DX
      push    ESI
      sub     DX,DX          ; DX = 0
      sub     BX,BX          ; BX = 0
      GetStr  number,5       ; get input number
      mov     ESI,number
get_next_char:
      mov     DL,[ESI]       ; read input from buffer
      cmp     DL,'-'         ; is it negative sign?
      je      sign           ; if so, save the sign
      cmp     DL,'+'         ; is it positive sign?
      jne     digit          ; if not, process the digit
sign:
      mov     BH,DL          ; BH keeps sign of input number
      inc    ESI
      jmp    get_next_char
digit:
      sub    AX,AX          ; AX = 0
      mov    BL,10          ; BL holds the multiplier
      sub    DL,’0’         ; convert ASCII to numeric
      mov    AL,DL
      mov    ECX,2           ; maximum two more digits to read
convert_loop:
      inc    ESI
      mov    DL,[ESI]
      cmp    DL,0           ; NULL?
      je     convert_done ; if so, done reading the number
      sub    DL,’0’         ; else, convert ASCII to numeric
      mul    BL             ; multiply total (in AL) by 10
      add    AX,DX          ; and add the current digit
      loop   convert_loop
convert_done:
      cmp    AX,128
      ja     out_of_range ; if AX > 128, number out of range
      jb     number_OK      ; if AX < 128, number is valid
      cmp    BH,’-’         ; if AX = 128, must be a negative;
      jne    out_of_range ; otherwise, an invalid number
number_OK:
      cmp    BH,’-’         ; number negative?
      jne    number_done    ; if not, we are done
      neg    AL             ; else, convert to 2’s complement
number_done:
      clc                   ; CF = 0 (no error)
      jmp    done
out_of_range:
      stc                   ; CF = 1 (range error)
done:
      pop    ESI            ; restore registers
      pop    DX
      pop    ECX
      pop    BX
      ret
```

The assembly language code for the GetInt8 procedure is given in Program 7.2. The procedure uses GetCh to read the input digits into DL.

- The character input digits are converted to their numeric equivalent by subtracting the character code for 0 (`'0'`) on line 31.
- The multiplication is done on line 40, which produces a 16-bit result in AX. Note that the numeric value of the current digit (in DX) is added (line 41) 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 44 to 48 perform the out-of-range error check. To do this check, the following conditions are tested:

AX > 128	⇒	out of range
AX = 128	⇒	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 45 and 46 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 52).
- The `clc` (clear CF) and `stc` (set CF) instructions are used to indicate the error condition (lines 54 and 57).

7.4 Multiword Arithmetic

The arithmetic instructions like `add`, `sub`, and `mul` work on 8-, 16- or 32-bit operands. What if an application requires numbers larger than 32 bits? Such numbers obviously require arithmetic to be done on multiword operands. In this section, we provide an introduction to multiword arithmetic by discussing how the basic four arithmetic operations—addition, subtraction, multiplication, and division—are done on unsigned 64-bit integers.

7.4.1 Addition and Subtraction

Addition and subtraction operations on multiword operands are straightforward. Let us first look at the addition operation. We start the addition process by adding the rightmost 32 bits of the two operands. In the next step, the next 32 bits are added along with any carry generated by the previous addition. Remember that the `adc` instruction can be used for this purpose.

The procedure `add64` (in `arith64.asm` file), for example, performs addition of two 64-bit numbers in EBX:EAX and EDX:ECX. The result is returned in EBX:EAX. The overflow condition is indicated by setting the carry flag.

Program 7.3 Addition of two 64-bit numbers

```
;------------------------------------------------------------
;Adds two 64-bit numbers received in EBX:EAX and EDX:ECX.
;The result is returned in EBX:EAX. Overflow/underflow
;conditions are indicated by setting the carry flag.
;Other registers are not disturbed.
;------------------------------------------------------------
add64:
       add    EAX,ECX
       adc    EBX,EDX
       ret
```

The 64-bit subtraction is also simple and similar to the 64-bit addition. For this subtraction, substitute `sub` for `add` and `sbb` for `adc` in the `add64` procedure.

7.4.2 Multiplication

Multiplication of multiword operands is not as straightforward as the addition and subtraction. In this section, we give two procedures to multiply two unsigned 64-bit numbers. The first one uses the longhand multiplication (see Appendix A). The second procedure uses the `mul` instruction.

Longhand Multiplication

This procedure tests bits of the multiplier from right to left and “appropriately” adds the multiplicand depending on whether the tested bit is 1 or 0. The following algorithm is a modification of the basic longhand multiplication. The final 128-bit product is in P:A.

```
P := 0
A := multiplier
B := multiplicand
count := 64
while (count > 0)
    if (LSB of A = 1)
    then
        P := P + B
        CF := carry generated by P + B
    else
        CF := 0
    end if
    shift right CF:P:A by one bit position
```

```
        {LSB of multiplier is not used in the rest of the algorithm}
    count := count − 1
end while
```

Remember that multiplying two 64-bit numbers yields a 128-bit number. To implement the algorithm for multiplying two unsigned n-bit numbers, we need three n-bit registers. We could use the memory but using memory slows down multiplication operation substantially. Since we are interested in multiplying two 64-bit numbers, we have enough general-purpose registers for use by the algorithm.

To see the workings of the algorithm, let us trace the steps for two 4-bit numbers A = 13D and B = 5D. The table below shows the contents of CF:P:A after the addition and the shift operations.

	After P + B			After the shift		
	CF	P	A	CF	P	A
initial state	?	0000	1101	—	—	—
iteration 1	0	0101	1101	?	0010	1110
iteration 2	0	0010	1110	?	0001	0111
iteration 3	0	0110	0111	?	0011	0011
iteration 4	0	1000	0011	?	0100	0001

Program 7.4 Multiplication of two 64-bit numbers using the longhand multiplication algorithm

```
;-----------------------------------------------------------
;Multiplies two 64-bit unsigned numbers A and B. The input
;number A is received in EBX:EAX and B in EDX:ECX registers.
;The 128-bit result is returned in EDX:ECX:EBX:EAX registers.
;This procedure uses longhand multiplication algorithm.
;Preserves all registers except EAX, EBX, ECX, and EDX.
;-----------------------------------------------------------
%define  COUNT   word[EBP-2] ; local variable

mult64:
      enter   2,0            ; 2-byte local variable space
      push    ESI
      push    EDI
      mov     ESI,EDX        ; ESI:EDI = B
      mov     EDI,ECX
      sub     EDX,EDX        ; P = 0
      sub     ECX,ECX
      mov     COUNT,64       ; count = 64 (64-bit number)
```

```
19:  step:
20:        test    EAX,1           ; LSB of A is 1?
21:        jz      shift1          ; if not, skip add
22:        add     ECX,EDI         ; Otherwise, P = P+B
23:        adc     EDX,ESI
24:  shift1:                       ; shift right P and A
25:        rcr     EDX,1
26:        rcr     ECX,1
27:        rcr     EBX,1
28:        rcr     EAX,1
29:
30:        dec     COUNT           ; if COUNT is not zero
31:        jnz     step            ;  repeat the process
32:        ; restore registers
33:        pop     EDI
34:        pop     ESI
35:        leave                   ; clears local variable space
36:        ret
```

The procedure `mult64` (in the `arith64.asm` file) implements this algorithm to multiply two unsigned 64-bit numbers. The two numbers are received in EBX:EAX and EDX:ECX. The 128-bit result is returned in EDX:ECX:EBX:EAX.

- The procedure uses the ESI:EDI register to store the 64-bit multiplicand `B`. The multiplier `A` is mapped to EBX:EAX and `P` to EDX:ECX.
- A local variable COUNT is used. It is accessible at [EBP−2]. The `%define` statement on line 8 establishes a convenient label to refer to it.
- The `while` loop is implemented by lines 19–31. The `if` condition is implemented by the `test` instruction on line 20.
- The 64-bit addition (P+B) is done by lines 22 and 23. These two statements are similar to the code given in the `add64` procedure.
- Right shift of CF:P:A is done by the four 32-bit `rcr` statements (lines 25–28). Note that the `test` instruction (line 20) clears the carry flag independent of the result. Therefore, if the LSB of A is zero, CF is zero during the right-shift process.

Using the mul Instruction

We now look at an alternative procedure that uses the 32-bit `mul` instruction for multiplying two unsigned 64-bit integers. The input number `A` can be considered as consisting of A0 and A1, with A0 representing the lower-order 32 bits and A1 the higher-order 32 bits. Similarly, B0 and B1 represent components of `B`. Now we can use the `mul` instruction to multiply these 32-bit components. The algorithm is as follows:

```
temp = A0 × B0
result = temp
temp = A1 × B0
temp = left shift temp value by 32 bits
{the shift operation replaces zeros on the right}
result = result + temp
temp = A0 × B1
temp = left shift temp value by 32 bits
{the shift operation replaces zeros on the right}
result = result + temp
temp = A1 × B1
temp = left shift temp value by 64 bits
{the shift operation replaces zeros on the right}
result = result + temp
```

The procedure `mult64w` follows the above algorithm in a straightforward fashion. This procedure, like the `mult64` procedure, receives the two 64-bit operands in EBX:EAX and EDX:ECX register pairs. The 128-bit result is returned in registers EDX:ECX:EBX:EAX. It uses a 128-bit local variable for storing the result. Note that the result is divided into four components and the `%define` statements on lines 9–12 assign labels to them.

Program 7.5 Multiplication of two 64-bit numbers using the `mul` instruction

```
 1:  ;-----------------------------------------------------------
 2:  ;Multiplies two 64-bit unsigned numbers A and B. The input
 3:  ;number A is received in EBX:EAX and B in EDX:ECX registers.
 4:  ;The 64-bit result is returned in EDX:ECX:EBX:EAX registers.
 5:  ;It uses mul instruction to multiply 32-bit numbers.
 6:  ;Preserves all registers except EAX, EBX, ECX, and EDX.
 7:  ;-----------------------------------------------------------
 8:  ; local variables
 9:  %define  RESULT3  dword[EBP-4] ; most significant 32 bits of result
10:  %define  RESULT2  dword[EBP-8]
11:  %define  RESULT1  dword[EBP-12]
12:  %define  RESULT0  dword[EBP-16]; least significant 32 bits of result
13:
14:  mult64w:
15:        enter   16,0           ; 16-byte local variable space for RESULT
16:        push    ESI
17:        push    EDI
18:        mov     EDI,EAX        ; ESI:EDI = A
19:        mov     ESI,EBX
```

```
mov     EBX,EDX        ; EBX:ECX = B
; multiply A0 and B0
mov     EAX,ECX
mul     EDI
mov     RESULT0,EAX
mov     RESULT1,EDX
; multiply A1 and B0
mov     EAX,ECX
mul     ESI
add     RESULT1,EAX
adc     EDX,0
mov     RESULT2,EDX
sub     EAX,EAX        ; store 1 in RESULT3 if a carry
rcl     EAX,1          ;  was generated
mov     RESULT3,EAX
; multiply A0 and B1
mov     EAX,EBX
mul     EDI
add     RESULT1,EAX
adc     RESULT2,EDX
adc     RESULT3,0
; multiply A1 and B1
mov     EAX,EBX
mul     ESI
add     RESULT2,EAX
adc     RESULT3,EDX
; copy result to the registers
mov     EAX,RESULT0
mov     EBX,RESULT1
mov     ECX,RESULT2
mov     EDX,RESULT3
; restore registers
pop     EDI
pop     ESI
leave                 ; clears local variable space
ret
```

7.4.3 Division

There are several division algorithms to perform n-bit unsigned integer division. Here we describe and implement what is called the “nonrestoring” division algorithm. The division operation, unlike the multiplication operation, produces two results: a quotient and a remainder. Thus when dividing two n-bit integers—A ÷ B—the quotient and the remainder are n-bits long as well.

To implement the division algorithm, we need an additional register P that is `n+1` bits long. The algorithm consists of testing the sign of P and, depending on this sign we either add or subtract B from P. Then P:A is left-shifted while manipulating the rightmost bit of A. After repeating these steps n times, the quotient is in A and the remainder in P. The pseudocode of the algorithm is given below.

```
P := 0
A := dividend
B := divisor
count := 64
while (count > 0)
    if (P is negative)
    then
        shift left P:A by one bit position
        P := P + B
    else
        shift left P:A by one bit position
        P := P − B
    end if
    if (P is negative)
    then
        set low-order bit of A to 0
    else
        set low-order bit of A to 1
    end if
count := count − 1
end while
if (P is negative)
    P := P + B
end if
```

After executing the algorithm, the quotient is in A and the remainder is in P.

An implementation of this division algorithm is given in Program 7.6. This procedure, like `mult64`, receives a 64-bit dividend in EBX:EAX and a 64-bit divisor in the EDX:ECX register pairs. The quotient is returned in the EBX:EAX register pair and the remainder in the EDX:ECX register pair. If the divisor is zero, the carry is set to indicate overflow error; the carry flag is cleared otherwise.

The P register is mapped to SIGN:EDX:ECX, where SIGN is a local variable that is used to store the sign of P. The code on lines 18–21 checks if the divisor is zero. If zero, the carry flag is set (line 22) and the control is returned. As in `mult64`, the procedure uses `rcl` to left shift the 65 bits consisting of SIGN:EDX:ECX:EBX:EAX (lines 35–39). The rest of the code follows the algorithm.

Program 7.6 Division of two 64-bit numbers

```
;-----------------------------------------------------------
;Divides two 64-bit unsigned numbers A and B (i.e., A/B).
;The number A is received in EBX:EAX and B in EDX:ECX registers.
;The 64-bit quotient is returned in EBX:EAX registers and
;the remainder is retuned in EDX:ECX registers.
;Divide-by-zero error is indicated by setting
;the carry flag; carry flag is cleared otherwise.
;Preserves all registers except EAX, EBX, ECX, and EDX.
;-----------------------------------------------------------
; local variables
%define  SIGN       byte[EBP-1]
%define  BIT_COUNT  byte[EBP-2]
div64:
      enter   2,0            ; 2-byte local variable space
      push    ESI
      push    EDI
      ; check for zero divisor in EDX:ECX
      cmp     ECX,0
      jne     non_zero
      cmp     EDX,0
      jne     non_zero
      stc                    ; if zero, set carry flag
      jmp     SHORT skip     ; to indicate error and return
non_zero:
      mov     ESI,EDX        ; ESI:EDI = B
      mov     EDI,ECX
      sub     EDX,EDX        ; P = 0
      sub     ECX,ECX
      mov     SIGN,0
      mov     BIT_COUNT,64   ; BIT_COUNT = # of bits
next_pass:    ; ****** main loop iterates 64 times ******
      test    SIGN,1         ; if P is positive
      jz      P_positive     ; jump to P_positive
P_negative:
      rcl     EAX,1          ; right-shift P and A
      rcl     EBX,1
      rcl     ECX,1
      rcl     EDX,1
      rcl     SIGN,1
      add     ECX,EDI        ; P = P + B
      adc     EDX,ESI
      adc     SIGN,0
      jmp     test_sign
P_positive:
      rcl     EAX,1          ; right-shift P and A
      rcl     EBX,1
      rcl     ECX,1
      rcl     EDX,1
      rcl     SIGN,1
      sub     ECX,EDI        ; P = P + B
      sbb     EDX,ESI
      sbb     SIGN,0
test_sign:
      test    SIGN,1         ; if P is negative
      jnz     bit0           ; set lower bit of A to 0
bit1:                        ; else, set it to 1
      or      AL,1
      jmp     one_pass_done  ; set lower bit of A to 0
bit0:
      and     AL,0FEH        ; set lower bit of A to 1
      jmp     one_pass_done
one_pass_done:
      dec     BIT_COUNT      ; iterate for 32 times
      jnz     next_pass
div_done:                    ; division completed
      test    SIGN,1         ; if P is positive
      jz      div_wrap_up    ; we are done
      add     ECX,EDI        ; otherwise, P = P + B
      adc     EDX,ESI
div_wrap_up:
      clc                    ; clear carry to indicate no error
skip:
      pop     EDI            ; restore registers
      pop     ESI
      leave                  ; clears local variable space
      ret
```

7.5 Performance: Multiword Multiplication

In this section, we study the performance of the mul instruction for some specific values of multipliers. In particular, we would like to see if multiplication by 10 can be done any faster than using the mul instruction. Multiplication by 10 can also be done using only additions. Such multiplications, for example, are needed in the GetInt8 procedure. Suppose we want to multiply contents of AL (say X) by 10. This can be done as follows:

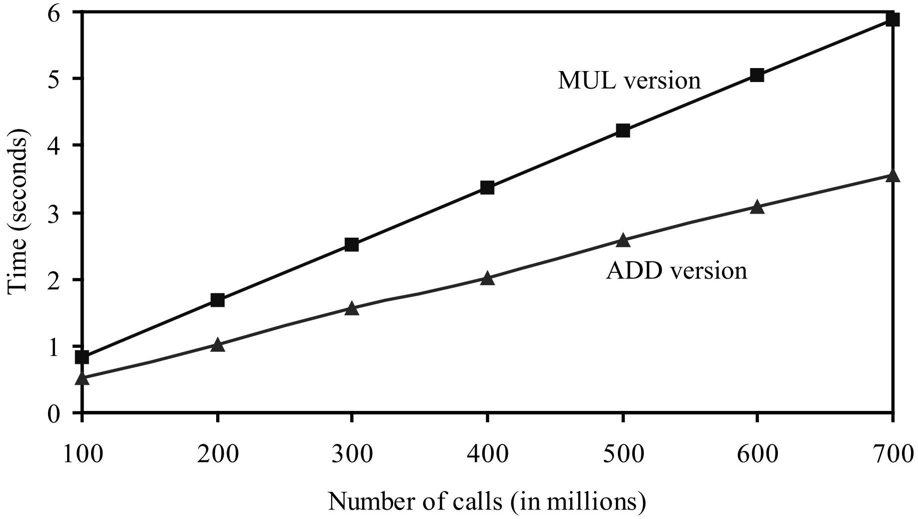


Figure 7.1 Performance of multiplication of a 32-bit number by 10.

```
sub    AH,AH      ; AH = 0
mov    BX,AX      ; BX = X
add    AX,AX      ; AX = 2X
add    AX,AX      ; AX = 4X
add    AX,BX      ; AX = 5X
add    AX,AX      ; AX = 10X
```

Figure 7.1 shows the performance of the these two versions for multiplying 255 by 10 on a 2.4-GHz Pentium 4 system. The add version is faster by about 60% despite the fact that it has more instructions than the mul version. In Chapter 9 we show that special multiplications by a power of 2 can be efficiently done by shift instructions. Such multiplications are often required to convert numbers from octal/hexadecimal number systems to the decimal system.

7.6 Summary

The status flags register the outcome of arithmetic and logical operations. Of the six status flags, zero flag, carry flag, overflow flag, and sign flag are the most important. The zero flag records whether the result of an operation is zero or not. The sign flag monitors the sign of the result. The carry and overflow flags record the overflow/underflow conditions of the arithmetic operations. The carry flag is set if the result on unsigned data is out of range; the overflow flag is used to indicate the out-of-range condition for signed data.

The instruction set includes instructions for addition, subtraction, multiplication, and division. While add and subtract instructions work on both unsigned and signed data, multiplication and division require separate instructions for signed and unsigned data. These arithmetic instructions can operate on 8-, 16-, and 32-bit operands. If numbers are represented using more than 32 bits, we need to devise methods for performing the arithmetic operations on multiword operands. We discussed how multiword arithmetic operations could be implemented.

We demonstrated that multiplication by special values (for example, multiplication by 10) can be done more efficiently by using addition. Chapter 9 discusses how the shift operations can be used to implement multiplication by a power of 2.

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
- Carry flag
- Multiword arithmetic
- Overflow flag
- Parity flag
- Sign flag
- Status flags
- Zero flag

7.7 Exercises

7–1 What is the significance of the carry flag?

7–2 What is the significance of the overflow flag?

7–3 Suppose the sign flag is not available. Is there a way to detect the sign of a number? Is there more than one way?

7–4 When is the parity flag set? What is a typical application that uses this flag?

7–5 When subtracting two numbers, suppose the carry flag is set. What does it imply in terms of the relationship between the two numbers?

7–6 In the last example, suppose the overflow flag is set. What does it imply in terms of the relationship between the two numbers?

7–7 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.

7–8 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.

7–9 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.

7–10 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.

7–11 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.

7–12 Explain why multiplication requires two separate instructions to work on signed and unsigned data.

7–13 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.

7.8 Programming Exercises

7–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`.

7–P2 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 input as an error and terminate after displaying an appropriate error message.

7–P3 Repeat the above exercise with the following modification—use multiplication instead of addition. Here is an example:

$7391928 = 7 * 3 * 9 * 1 * 9 * 2 * 8 = 27216$
$27216 = 2 * 7 * 2 * 1 * 6 = 168$
$168 = 1 * 6 * 8 = 48$
$48 = 4 * 8 = 32$
$32 = 3 * 2 = 6.$

7–P4 The `PutInt8` procedure uses 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.

7–P5 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.

7–P6 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-hand and the right-hand sides of the equation, verifies that they are equal, and displays the result.

7–P7 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%.

7–P8 Modify the above program to round the average test score. For example, if the average is 15.55, it should be rounded to 16.

7–P9 Modify the average test score program to display the fractional part as well. Display the average test score in `dd.dd` format.

7–P10 Write a program to convert temperature from Celsius to Fahrenheit. The formula is

$$F = \frac{9}{5} \times C + 32\,.$$

7–P11 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.

$$\begin{aligned} \text{Volume} &= L \times W \times H \\ \text{Surface volume} &= 2 \times (L \times H + L \times W + W \times H)\,. \end{aligned}$$

Chapter 8

Selection and Iteration

Objectives

- To discuss unconditional and conditional jump instructions
- To describe the loop family of instructions
- To explore how this set of instructions can be used to implement high-level language decision structures

Modern high-level languages provide a variety of decision structures. These structures include selection structures such as `if-then-else` *and iterative structures such as* `while` *and* `for` *loops. The assembly language, being a low-level language, does not provide these structures directly. However, the assembly language provides several basic instructions that could be used to construct these high-level language selection and iteration structures. These assembly language instructions include the unconditional jump, compare, conditional jump, and loop. We briefly introduced some of these instructions in Chapter 4. The first four sections of this chapter complement that discussion.*

Section 8.5 discusses how the jump, compare, and loop instructions can be used to implement high-level language selection and iteration structures. After giving some examples in Section 8.6, we describe the indirect jump instruction and its use in implementing multiway switch or case statements in Section 8.7. The chapter concludes with a summary.

8.1 Unconditional Jump

We introduced the unconditional jump (`jmp`) instruction in Chapter 4. It unconditionally transfers control to the instruction located at the target address. The general format, as we have seen before, is

```
jmp    target
```

There are several versions of the `jmp` instruction depending on how the target address is specified and where the target instruction is located.

Specification of Target

There are two distinct ways by which the target address of the `jmp` instruction can be specified: *directly* and *indirectly*. The vast majority of jumps are of the direct type. These are the types of unconditional jumps we discussed in Section 4.5.2. Therefore, we focus our attention on the direct jump instructions and briefly discuss the indirect jumps in Section 8.7.

Direct Jumps

In the direct jump instruction, the target address is specified directly as part of the instruction. In the following code fragment

```
          . . .
         mov    ECX,10
         jmp    ECX_init_done
init_ECX_20:
         mov    ECX,20
ECX_init_done:
         mov    EAX,ECX
repeat1:
         dec    ECX
          . . .
         jmp    repeat1
          . . .
```

both the `jmp` instructions directly specify the target. As an assembly language programmer, you only specify the target address by using a label; the assembler figures out the exact value by using its symbol table.

The instruction

```
jmp   ECX_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. Rather, it specifies the relative displacement in bytes between the target instruction and the instruction following the jump instruction (and not from the jump instructions itself!).

In order to see why this is so, we have to understand how jumps are executed. Recall that the EIP register always points to the next instruction to be executed (see Chapter 3). Thus, after fetching the `jmp` instruction, the EIP is automatically advanced to point to the instruction following the `jmp` instruction. Execution of `jmp` involves changing the EIP from where it is currently pointing to the target instruction location. This is achieved by adding the difference (i.e., the relative displacement) to the EIP 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 relative address as opposed to absolute address of the target instruction is appropriate for dynamically relocatable code (i.e., for position-independent code).

Where Is the Target?

If the target of a jump instruction is located in the same segment as the jump 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:

EIP = EIP $+$ relative-displacement

In the case of an intersegment jump, called *far jump*, the CS is also changed to point to the target segment, as shown below:

CS = target-segment
EIP = target-offset

Both target-segment and target-offset are specified directly in the instruction. Thus, for 32-bit segments, the instruction encoding for the intersegment jump takes seven bytes: one byte for the specification of the opcode, two bytes for the target-segment, and four bytes for the target-offset specification.

The majority of jumps are of the intrasegment type. There are two ways to specify intrasegment jumps depending on the distance of the target location from the instruction following the jump instruction—that is, depending on the value of the relative displacement.

If the relative displacement, which is a signed number, fits 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, 2 or 4 bytes are used to specify the relative displacement. A two-byte displacement is used for 16-bit segments, and a 4-byte displacement for 32-bit segments. As a result, the jump instruction requires either 3 or 5 bytes to encode in the machine language. This form is called the *near jump*.

If you want to use the short jump form, you can inform the assembler of your intention by using the operator SHORT, as shown below:

```
    jmp     SHORT ECX_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 you with an error message that the target can't be reached with a short jump.

In fact, specification of SHORT in a statement like

```
    jmp     repeat1
```

in the example code on page 240 is redundant. This is because the assembler can automatically select the SHORT jump, if appropriate, for all backward jumps. However, for forward jumps, the assembler needs your 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, use the SHORT operator only for forward jumps if appropriate.

Example 8.1 *Example encodings of short and near jumps.*
Figure 8.1 shows some example encodings for short and near jump instructions. The forward short jump on line 167 is encoded in the machine language as `EB 14`, where `EB` is the opcode for the short jump. The relative offset to target `ECX_init_done` is `14H`. From the code, it can be seen that this is the difference between the address of the target (address 0000001FH) and the instruction following the jump instruction on line 168 (address 0000000BH).

The jump instruction encoding on line 169 requires some explanation. This is a forward jump, but we have not specified that it could be a short jump (unlike the jump instruction on line 167). Thus, the assembler reserves five bytes for a near jump (the worst-case scenario). Even though this jump instruction could be encoded using a short jump, NASM uses near jump encoding for this instruction. For near jumps, the opcode is E9H and the relative offset is a 32-bit signed integer. For this instruction, the relative offset of the target is given as 0000000AH, which is the difference between the target address 0000001FH and the address of the instruction at 00000015H. Another example of a forward short jump that is encoded using five bytes is given on line 557.

The backward instruction on line 177 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 (equal to −3D), which is the offset from the instruction following the jump instruction at address 24H to `repeat1` at 21H.

The near jump on line 172 is encoded with the relative offset of 00000652H. The offset is presented in little-endian order (see our discussion in Section 2.5.3 on page 37). This offset represents the difference between addresses 00000671H and 0000001FH. □

8.2 Compare Instruction

Implementation of high-level language decision structures like `if-then-else` in the assembly language is a two-step process:

```
                                          . . .
167   00000009   EB14                              jmp    SHORT ECX_init_done
168   0000000B   B978563412                        mov    ECX,12345678H
169   00000010   E90A000000                        jmp    ECX_init_done
170                                       init_ECX:
171   00000015   B912EFCDAB                        mov    ECX,0ABCDEF12H
172   0000001A   E952060000                        jmp    near_jump
173                                       ECX_init_done:
174   0000001F   89C8                              mov    EAX,ECX
175                                       repeat1:
176   00000021   49                                dec    ECX
177   00000022   EBFD                              jmp    repeat1
                                          . . .
                                          . . .
557   00000662   E905000000                        jmp    short_jump
558   00000667   B9FFFF00FF                        mov    ECX, 0FF00FFFFH
559                                       short_jump:
560   0000066C   BA32547698                        mov    EDX, 98765432H
561                                       near_jump:
562   00000671   E99FF9FFFF                        jmp    init_ECX
                                          . . .
```

Figure 8.1 Example encodings of jump instructions. The first column gives the line number while the second column shows the memory address of the instruction. The third column gives the machine language encoding of the assembly language instruction shown in the fourth column.

1. An arithmetic or comparison instruction updates one or more of the arithmetic flags;
2. A conditional jump instruction causes selective execution of the appropriate code fragment based on the values of the flags.

We discussed the compare (`cmp`) instruction on page 82. The main purpose of the `cmp` instruction is to update the flags so that a subsequent conditional jump instruction can test these flags.

Example 8.2 *Some examples of the compare instruction.*
The four flags that are useful in establishing a relationship ($<$, $\leq$, $>$, and so on) between two integers are CF, ZF, SF, and OF. Table 8.1 gives some examples of executing the

```
cmp    AL,DL
```

instruction. Recall that the CF is set if the result is out of range when treating the operands as unsigned numbers. For this example, this range is 0 to 255D. Similarly, the OF is set if the result is out of range for signed numbers (for our example, this range is -128D to +127D).

Table 8.1 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

In general, the value of ZF and SF can be obtained in a straightforward way. Therefore, let us focus on the carry and overflow flags. In the first example, since 56−57 = −1, CF is set but not OF. The second example is not so simple. Treating the operands in AL and DL as unsigned numbers, 200−101 = 99, which is within the range of unsigned numbers. Therefore, CF = 0. However, when treating 200D (= C8H) as a signed number, it represents −56D. Therefore, compare performs −56−101 = −157, which is out of range for signed numbers, resulting in setting OF. We will leave verification of the rest of the examples as an exercise. □

8.3 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;
3. Jumps based on signed comparisons.

8.3.1 Jumps Based on Single Flags

The 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. These instructions are summarized in Table 8.2.

As shown in Table 8.2, the jump instructions that test the zero and parity flags have aliases (e.g., `je` is an alias for `jz`). These aliases are provided to improve program readability. For example,

```
if (count = 100)
then
    <statement1>
end if
```

Table 8.2 Jumps Based on Single Flag Value

Mnemonic	Meaning	Jumps if
Testing for zero:		
jz	jump if zero	ZF = 1
je	jump if equal	
jnz	jump if not zero	ZF = 0
jne	jump if not equal	
jecxz	jump if ECX = 0	ECX = 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	

can be written in the assembly language as

```
      cmp    count,100
      jz     S1
          . . .
S1:
      <statement1 code here>
          . . .
```

But our use of `jz` does not convey that we are testing for equality. This meaning is better conveyed by

```
        cmp     count,100
        je      S1
            . . .
S1:
        <statement1 code here>
            . . .
```

The assembler, however, treats both `jz` and `je` as synonymous instructions.

The only surprising instruction in Table 8.2 is the `jecxz` instruction. This instruction does not test any flag but tests the contents of the ECX register for zero. If the operand size is 16 bits, we can use the `jcxz` instruction instead of `jecxz`. Both instructions, however, use the same opcode, E3H. The operand size determines the register—CX or ECX—used.

This instruction is often used in conjunction with the `loop` instruction. Therefore, we postpone our discussion of this instruction to Section 8.4, which discusses the `loop` instruction.

8.3.2 Jumps Based on Unsigned Comparisons

When comparing two numbers

```
cmp     num1,num2
```

it is necessary to know whether these numbers `num1` and `num2` represent singed 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 treat their contents as representing unsigned numbers. This is because, in unsigned representation, AL = 183D and DL = 110D. However, if the contents of the AL and DL registers are treated as representing signed numbers, AL $<$ DL as the AL register has a negative number (-73D) while the DL register has a positive number (+110D).

Note that when using a `cmp` statement like

```
cmp     num1,num2
```

we are always comparing `num1` to `num2` (e.g., num1 $<$ num2, num1 $>$ num2, and so on). There are six possible relationships between two numbers:

$$\begin{aligned}
&\text{num1} = \text{num2}\\
&\text{num1} \neq \text{num2}\\
&\text{num1} > \text{num2}\\
&\text{num1} \geq \text{num2}\\
&\text{num1} < \text{num2}\\
&\text{num1} \leq \text{num2}
\end{aligned}$$

Table 8.3 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

For the unsigned numbers, the carry and the zero flags record the necessary information in order 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 8.3. Notice that "above" and "below" are used for > and < relationships for the unsigned comparisons, reserving "greater" and "less" for signed comparisons, as we shall see next.

8.3.3 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 8.4 for signed comparisons.

For signed comparisons, three flags record the necessary information: the sign flag (SF), the overflow flag (OF), and the 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 `cmp` instruction

```
    cmp     Snum1,Snum2
```

Table 8.4 Jumps Based on Signed Comparison

Mnemonic	Meaning	Condition tested
je	jump if equal	ZF = 1
jz	jump if zero	
jne	jump if not equal	ZF = 0
jnz	jump if not zero	
jg	jump if greater	ZF = 0 and SF = OF
jnle	jump if not less or equal	
jge	jump if greater or equal	SF = OF
jnl	jump if not less	
jl	jump if less	SF $\neq$ OF
jnge	jump if not greater or equal	
jle	jump if less or equal	ZF = 1 or SF $\neq$ OF
jng	jump if not greater	

Conditions for Snum1 $>$ Snum2

The following table shows several examples in which `Snum1` $>$ `Snum2` holds.

`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

It appears from these examples that `Snum1` $>$ `Snum2` if

	ZF	OF	SF
	0	0	0
or			
	0	1	1

That is, ZF = 0 and OF = SF. We cannot use just OF = SF 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 8.4, these are the conditions tested for the `jg` conditional jump.

Conditions for Snum1 < Snum2

Again, as in the previous case, we develop our intuition by means of a few examples. The following table shows several examples in which `Snum1` < `Snum2` holds.

`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

It appears from these examples that `Snum1` < `Snum2` if

	ZF	OF	SF
	0	0	1
or			
	0	1	0

That is, 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 8.4, this is the condition tested by the `jl` conditional jump instruction.

8.3.4 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 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, you can use the following trick to overcome this limitation of the conditional jump instructions.

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 a short jump, it should be replaced by

```
          . . .
target:
          . . .
      cmp    AX,BX
      jne    skip1    ; skip1 is a short jump
      jmp    target
skip1:
      mov    CX,10
          . . .
```

What we have done here is negated the test condition (`je` becomes `jne`) and used an unconditional jump to transfer control to target. Recall that `jmp` instruction has both *short* and *near* versions.

8.4 Looping Instructions

Instructions in this group use the CX or ECX register to maintain repetition count. The CX register is used if the operand size is 16 bits; ECX is used for 32-bit operands. In the following discussion, we assume that the operand size is 32 bits. The three loop instructions decrement the ECX register before testing it for zero. Decrementing ECX does not affect any of the flags. The format of these instructions along with the action taken are shown below:

Mnemonic	Meaning	Action

`loop target`	loop	ECX = ECX − 1
		if ECX $\neq$ 0
		jump to target

`loope target`
`loopz target`
	loop while equal
	loop while zero
		ECX = ECX − 1
		if (ECX $\neq$ 0 and ZF = 1)
		jump to target

`loopne target`
`loopnz target`
	loop while not equal
	loop while not zero
		ECX = ECX − 1
		if (ECX $\neq$ 0 and ZF = 0)
		jump to target

The destination specified in these instructions should be reachable by a short jump. This is a consequence of using the two-byte encoding with a single byte indicating the relative displacement, which should be within −128 to +127.

The use of the `loop` instruction is straightforward to understand; however, the other two loop instructions require some explanation. These instructions are useful in writing loops for applications that require two termination conditions. The following example illustrates this point.

Example 8.3 *A loop example.*

Let us say that we want to write a loop that reads a string of characters from the user. The character input can be terminated either when the buffer is full, or when the user types a carriage-return (CR) character, whichever occurs first.

```
CR     EQU    0DH
SIZE   EQU    81
.UDATA
buffer  resb  SIZE            ; buffer for string input
.CODE
         . . .
    mov     EBX,buffer        ; EBX points to buffer
    mov     ECX,SIZE          ; buffer size in ECX
read_more:
    GetCh   AL
    mov     [EBX],AL
    inc     EBX
    cmp     AL,CR             ; see if char is CR
    loopne  read_more
         . . .
```

We use `loopne` to test the two conditions for terminating the read loop. □

A problem with the above code is that if ECX is initially 0, the loop attempts to read 2^{16} or 65536D characters from the user unless terminated by typing a CR character. This is not what we want!

The instruction `jecxz` provides a remedy for this situation by testing the ECX register. The syntax of this instruction is

```
jecxz    target
```

which tests the ECX register, and if it is zero, control is transferred to the target instruction. Thus, it is equivalent to

```
cmp    ECX,0
jz     target
```

except that `jecxz` does not affect any of the flags, while the `cmp`/`jz` combination affects the status flags. By using this instruction, the previous example can be written as

```
      mov     EBX,buffer     ; BX points to buffer
      mov     ECX,SIZE       ; buffer size in ECX
      jecxz   read_done
read_more:
      GetCh   AL
```

```
        mov      [EBX],AL
        inc      EBX
        cmp      AL,CR          ; see if char is CR
        loopne   read_more
read_done:
              . . .
```

Notes on Execution Times of `loop` and `jcxz` Instructions

1. The functionality of the `loop` instruction can be replaced by

   ```
   dec    ECX
   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 or not. The `dec/jnz` instruction pair takes only two clocks. Of course, the `loop` instruction is better for program readability.
2. Similarly, the `jecxz` instruction takes five or six clocks, whereas the equivalent

   ```
   cmp    ECX,0
   jz     target
   ```

   takes only two clocks. Thus, for code optimization, these complex instructions should be avoided.

8.5 Implementing High-Level Language Decision Structures

In this section, we see how the jump and loop instructions can be used to implement high-level language selective and iterative structures.

8.5.1 Selective Structures

The selective structures allow the programmer to select from alternative actions. 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

```
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 8.4 *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:
           . . .
```

As you can see from this example, the condition testing is done by a pair of compare and conditional jump instructions. The label `then_part` is really not needed but is included to improve readability of the code. The first statement in the `then_part` is redundant, but Turbo C compiler generates it anyway. □

Example 8.5 *An* `if` *example with an* `and` *logical operator.*
The following code tests whether `ch` is a lowercase character or not. The condition in this example is a compound condition of two simple conditional statements connected by 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 the Turbo C compiler 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 pairs of compare and conditional jump instructions. Notice that `ch`$-$`32` is implemented as addition of -32. Also, you see redundancy in the code generated by the compiler. An advantage of writing in the assembly language is that we can avoid such redundancies. □

Example 8.6 *An* `if` *example with an* `or` *logical operator.*
As a last example, 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:
          . . .
```

The Turbo C compiler maps the variable `index` to the CX register. Also, the code uses the exclusive-or (`xor`) logical operator to zero CX. □

8.5.2 Iterative Structures

High-level languages provide several looping constructs. These include `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 loop* or the *entry-test loop*. The loop body is executed repeatedly as long as the condition is true.

Example 8.7 *An example* `while` *loop.*
Consider the following example code in C:

```
while(total < 700)
  {
    <loop body>
  }
```

The Turbo C compiler 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 loop* 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 8.8 *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. Here we consider two `for` loop examples.

Example 8.9 *Upward counting* `for` *loop.*

```
     for (i = 0; i < SIZE; i++)   /* for (i = 0 to SIZE−1) */
       {
         <loop body>
       };
```

The Turbo C compiler 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 8.10 *Downward counting* for *loop.*

```
         for (i = SIZE-1; i >= 0; i--)      /* for (i = SIZE−1 downto 0) */
           {
             <loop body>
           };
```

The Turbo C compiler 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 and or is used to test if i has reached zero. □

8.6 Illustrative Examples

In this section, we will present two 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.

Example 8.11 *Linear search of an integer array.*

In this example, the user is asked to input an array of nonnegative integers and then query whether a given number is in the array or not. The program uses a procedure that implements the linear search to locate a number in an 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 (line 37), with ECX initialized to MAX_SIZE (line 29), is used to iterate a maximum of MAX_SIZE times. The other loop termination condition (i.e., input of a negative number) is tested on lines 32 and 33. 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 instruction to test these two conditions for the termination of the search loop. The ECX is initialized (line 79) to the size of the array. In addition, a compare (line 84) 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 88 and 89) and returned in AX.

Program 8.1 Linear search of an integer array

```
 1:  ;Linear search of integer array      LIN_SRCH.ASM
 2:  ;
 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:  %include "io.mac"
11:
12:  MAX_SIZE       EQU   100
13:
14:  .DATA
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:
```

```
.UDATA
array          resw  MAX_SIZE

.CODE
        .STARTUP
        PutStr  input_prompt ; request input array
        mov     EBX,array
        mov     ECX,MAX_SIZE
array_loop:
        GetInt  AX           ; read an array number
        cmp     AX,0         ; negative number?
        jl      exit_loop    ; if so, stop reading numbers
        mov     [EBX],AX     ; otherwise, copy into array
        inc     EBX          ; increment array address
        inc     EBX
        loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
        mov     EDX,EBX      ; EDX keeps the actual array size
        sub     EDX,array    ; EDX = array size in bytes
        sar     EDX,1        ; divide by 2 to get array size
read_input:
        PutStr  query_number ; request number to be searched for
        GetInt  AX           ; read the number
        push    AX           ; push number, size & array pointer
        push    EDX
        push    array
        call    linear_search
        ; linear_search returns in AX the position of the number
        ; in the array; if not found, it returns 0.
        cmp     AX,0         ; number found?
        je      not_found    ; if not, display number not found
        PutStr  out_msg      ; else, display number position
        PutInt  AX
        jmp     SHORT user_query
not_found:
        PutStr  not_found_msg
user_query:
        nwln
        PutStr  query_msg    ; query user whether to terminate
        GetCh   AL           ; read response
        cmp     AL,'Y'       ; if response is not 'Y'
        jne     read_input   ; repeat the loop
done:                        ; otherwise, terminate program
        .EXIT
```

```

;-----------------------------------------------------------
; This procedure receives a pointer to an array of integers,
; the array size, and a number to be searched via the stack.
; If found, it returns in AX the position of the number in
; the array; otherwise, returns 0.
; All registers, except EAX, are preserved.
;-----------------------------------------------------------
linear_search:
       enter   0,0
       push    EBX           ; save registers
       push    ECX
       mov     EBX,[EBP+8]   ; copy array pointer
       mov     ECX,[EBP+12]  ; copy array size
       mov     AX,[EBP+16]   ; copy number to be searched
       sub     EBX,2         ; adjust index to enter loop
search_loop:
       add     EBX,2         ; update array index
       cmp     AX,[EBX]      ; compare the numbers
       loopne  search_loop
       mov     AX,0          ; set return value to zero
       jne     number_not_found  ; modify it if number found
       mov     EAX,[EBP+12]  ; copy array size
       sub     EAX,ECX       ; compute array index of number
number_not_found:
       pop     ECX           ; restore registers
       pop     EBX
       leave
       ret     10
```

Example 8.12 *Sorting of an integer array 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 the second position by exchanging values as in step 2;
5. Continue this process until the array is sorted.

The selection sort procedure implements the following pseudocode:

```
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 8.2, implements this pseudocode with the following mapping of variables: `position` is maintained in ESI, and EDI 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 ECX.

Program 8.2 Sorting of an integer array using the selection sort algorithm

```
;Sorting an array by selection sort    SEL_SORT.ASM
;
;        Objective: To sort an integer array using selection sort.
;            Input: Requests numbers to fill array.
;           Output: Displays sorted array.
%include "io.mac"

MAX_SIZE       EQU   100

.DATA
input_prompt   db    "Please enter input array: "
               db    "(negative number terminates input)",0
out_msg        db    "The sorted array is:",0

.UDATA
array          resw  MAX_SIZE

```

```
.CODE
        .STARTUP
        PutStr  input_prompt ; request input array
        mov     EBX,array
        mov     ECX,MAX_SIZE
array_loop:
        GetInt  AX           ; read an array number
        cmp     AX,0         ; negative number?
        jl      exit_loop    ; if so, stop reading numbers
        mov     [EBX],AX     ; otherwise, copy into array
        add     EBX,2        ; increment array address
        loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
        mov     EDX,EBX      ; EDX keeps the actual array size
        sub     EDX,array    ; EDX = array size in bytes
        sar     EDX,1        ; divide by 2 to get array size
        push    EDX          ; push array size & array pointer
        push    array
        call    selection_sort
        PutStr  out_msg      ; display sorted array
        nwln
        mov     ECX,EDX      ; ECX = array size
        mov     EBX,array
display_loop:
        PutInt  [EBX]
        nwln
        add     EBX,2
        loop    display_loop
done:
        .EXIT

;-----------------------------------------------------------
; This procedure receives a pointer to an array of integers
; and the array size via the stack. The array is sorted by
; using the selection sort. All registers are preserved.
;-----------------------------------------------------------
%define SORT_ARRAY  EBX
selection_sort:
       pushad                ; save registers
       mov     EBP,ESP
       mov     EBX,[EBP+36]  ; copy array pointer
       mov     ECX,[EBP+40]  ; copy array size
       sub     ESI,ESI       ; array left of ESI is sorted
sort_outer_loop:
       mov     EDI,ESI
       ; DX is used to maintain the minimum value and AX
       ; stores the pointer to the minimum value
       mov     DX,[SORT_ARRAY+ESI]  ; min. value is in DX
       mov     EAX,ESI       ; EAX = pointer to min. value
       push    ECX
       dec     ECX           ; size of array left of ESI
sort_inner_loop:
       add     EDI,2         ; move to next element
       cmp     DX,[SORT_ARRAY+EDI] ; less than min. value?
       jle     skip1         ; if not, no change to min. value
       mov     DX,[SORT_ARRAY+EDI] ; else, update min. value (DX)
       mov     EAX,EDI       ;       & its pointer (EAX)
skip1:
       loop    sort_inner_loop
       pop     ECX
       cmp     EAX,ESI       ; EAX = ESI?
       je      skip2         ; if so, element at ESI is its place
       mov     EDI,EAX       ; otherwise, exchange
       mov     AX,[SORT_ARRAY+ESI]  ; exchange min. value
       xchg    AX,[SORT_ARRAY+EDI]  ; & element at ESI
       mov     [SORT_ARRAY+ESI],AX
skip2:
       add     ESI,2         ; move ESI to next element
       dec     ECX
       cmp     ECX,1         ; if ECX = 1, we are done
       jne     sort_outer_loop
       popad                 ; restore registers
       ret     8
```

8.7 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 8.1 on page 243). We now look at indirect jumps. We limit our discussion to jumps within a segment.

In an indirect jump, the target address is specified indirectly through either memory or a general-purpose register. Thus, we can write

```
jmp    [ECX]
```

if the ECX register contains the offset of the target. In indirect jumps, the target offset is the absolute value (unlike the direct jumps, which use a relative offset value). The next example shows how indirect jumps can be used with a jump table stored in memory.

Example 8.13 *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 represented by the input. The listing is shown in Program 8.3. An input between 0 and 9 is valid. Any other input to the program may cause the system to hang up or crash. If the input is 0, 1, or 2, the program displays a simple message to indicate the class selection. Other digit inputs terminate the program. If a nondigit input is given to the program, it displays an error message and requests a valid digit input.

In order to use the indirect jump, we have to build a jump table of pointers (see lines 9–18). The input is tested for its validity on lines 33 to 36. If the input is a digit, it is converted to act as an index into the jump table and stored in ESI. This value is used in the indirect jump instruction (line 42). The rest of the program is straightforward to follow.

Program 8.3 An example demonstrating the use of the indirect jump

```
 1:  ;Sample indirect jump example    IJUMP.ASM
 2:  ;
 3:  ;        Objective: To demonstrate the use of indirect jump.
 4:  ;            Input: Requests a digit character from the user.
 5:  ;           Output: Appropriate class selection message.
 6:  %include "io.mac"
 7:
 8:  .DATA
 9:  jump_table  dd  code_for_0   ; indirect jump pointer table
10:              dd  code_for_1
11:              dd  code_for_2
12:              dd  default_code ; default code for digits 3-9
13:              dd  default_code
14:              dd  default_code
15:              dd  default_code
16:              dd  default_code
17:              dd  default_code
18:              dd  default_code
19:
20:  prompt_msg  db  "Type a digit: ",0
21:  msg_0       db  "Economy class selected.",0
22:  msg_1       db  "Business class selected.",0
23:  msg_2       db  "First class selected.",0
24:  msg_default db  "Not a valid code!",0
25:  msg_nodigit db  "Not a digit! Try again.",0
26:
27:  .CODE
```

```
        .STARTUP
read_again:
        PutStr   prompt_msg    ; request a digit
        sub      EAX,EAX       ; EAX = 0
        GetCh    AL            ; read input digit and
        cmp      AL,’0’        ; check to see if it is a digit
        jb       not_digit
        cmp      AL,’9’
        ja       not_digit
        ; if digit, proceed
        sub      AL,’0’        ; convert to numeric equivalent
        mov      ESI,EAX       ; ESI is index into jump table
        add      ESI,ESI       ; ESI = ESI * 4
        add      ESI,ESI
        jmp      [jump_table+ESI] ; indirect jump based on ESI
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

not_digit:
        PutStr   msg_nodigit
        nwln
        jmp      read_again
done:
        .EXIT
```

8.7.1 Multiway Conditional Statements

In high-level languages, a two- or three-way conditional execution can be controlled easily by using `if` statements. For large multiway conditional execution, writing the code with nested `if` statements is tedious and error-prone. High-level languages like C provide a special construct for multiway conditional execution. In this section we look at the C `switch` construct for multiway conditional execution.

Example 8.14 *Multiway conditional execution in C.*
As an example of the `switch` statement, 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`, execute the `count[0]++` statement. The `break` statement is necessary to escape out of 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 the other case statements.

The Turbo C compiler produces the assembly language code shown in Figure 8.2. The jump table is constructed in the code segment (lines 31–34). 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. □

8.8 Summary

We discussed the unconditional and conditional jump instructions as well as compare and loop instructions in detail. These assembly language instructions are useful in implementing high-level language selection and iteration constructs such as `if-then-else` and `while`

```
_main    PROC   NEAR
            . . .
            . . .
         mov    AL,ch
         cbw
         sub    AX,48    ; 48 = ASCII for '0'
         mov    BX,AX
         cmp    BX,3
         ja     default
         shl    BX,1     ; BX = BX*2
         jmp    WORD PTR CS:jump_table[BX]
case_0:
         inc    WORD PTR [BP-10]
         jmp    SHORT end_switch
case_1:
         inc    WORD PTR [BP-8]
         jmp    SHORT end_switch
case_2:
         inc    WORD PTR [BP-6]
         jmp    SHORT end_switch
case_3:
         inc    WORD PTR [BP-4]
         jmp    SHORT end_switch
default:
         inc    WORD PTR [BP-2]
end_switch:
           . . .
           . . .
_main    ENDP
jump_table  LABEL  WORD
            dw     case_0
            dw     case_1
            dw     case_2
            dw     case_3
              . . .
```

Figure 8.2 Assembly language code for the `switch` statement.

loops. Through detailed examples, we discussed how these high-level decision structures are implemented in the assembly language.

In the previous chapters, we extensively used direct jump instructions. In this chapter, we introduced the indirect jump instruction. In this jump instruction, the target of the jump is specified indirectly. Indirect jumps are useful to implement multiway conditional statements such as the switch statement in C. By means of an example, we have shown how such multiway statements of high-level languages are implemented in the assembly language.

8.9 Exercises

8–1 What is the difference between SHORT and NEAR jumps?

8–2 What is the range of SHORT and NEAR jumps? Explain the reason for this range limit.

8–3 What are forward and backward jumps?

8–4 Why does the assembler need your help for forward near jumps?

8–5 As you know, all conditional jumps are SHORT jumps. How do you handle conditional near jumps?

8–6 Describe the semantics of the jecxz instruction and explain how it is useful.

8–7 In Table 8.3, explain intuitively why the flags tested are necessary and sufficient to implement conditional jumps.

8–8 We have stated on page 249 that to detect Snum1<Snum2, the condition ZF = 0 is not necessary. Justify this statement.

8–9 In Table 8.4, explain intuitively why the flags tested are necessary and sufficient to implement conditional jumps.

8–10 Fill in the blanks in the following table, assuming that the

```
cmp     AH,AL
```

instruction is executed. Note that all numbers are in decimal.

AH	AL	CF	ZF	SF	OF
21	−21				
−21	−21				
−21	21				
255	−1				
129	−1				
128	−1				
128	−128				
128	127				

8–11 What is the difference between loop and loopne instructions?

8–12 Compare and contrast direct and indirect jumps.

8–13 What high-level language construct requires the use of the indirect jump for efficient implementation?

8.10 Programming Exercises

8–P1 Modify Program 8.1 so that the user can enter both positive and negative numbers (including zero). In order to facilitate this, the user will first enter a number indicating the number of elements of the array that he or she is going to enter next. For example, in the input

5 1987 −265 1349 0 5674

the first number 5 indicates the number of array entries to follow. Your program should perform array bound checks.

8–P2 Suppose we are given a sorted array of integers. Further assume that the array is sorted in ascending order. Then we can modify the linear search procedure to search for a number *S* so that it stops searching either when *S* is found or when a number greater than *S* is found. Modify the linear search program shown in Program 8.1 to work on a sorted array. For this exercise, assume that the user supplies the input data in sorted order.

8–P3 In the last exercise, you assumed that the user supplies data in sorted order. In this exercise, remove this restriction on the input data. Instead, use the selection sort procedure given in Program 8.2 to sort the array after reading the input data.

8–P4 Write an assembly language program to read a string of characters from the user and that prints the vowel count. 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
```

8–P5 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.

8–P6 Suppose that we want to list each uppercase and lowercase vowel 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.

8–P7 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 < n)
            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 enters the two input arrays in sorted (ascending) order. The merge sort procedure receives the five parameters via the stack.

Chapter 9

Logical and Bit Operations

Objectives

- To discuss logical family of instructions
- To describe shift and rotate family of instructions
- To see how these instructions are useful in bit manipulation and Boolean expressions

As we have seen in the last chapter, high-level languages provide several conditional and loop constructs. These constructs require Boolean or logical expressions for specifying conditions. The assembly language provides several logical instructions to implement logical expressions. These instructions, described in Section 9.1, are also useful in implementing bitwise logical operations.

Bit manipulation is an important aspect of any high-level language. The logical instructions discussed in Section 9.1 are useful in bit manipulation. In addition, several shift and rotate instructions are provided to facilitate bit manipulation. Shift instructions are discussed in Section 9.2, while rotate instructions are described in Section 9.3.

Issues related to logical expressions in high-level languages are discussed in Section 9.4. There are several instructions to test and modify bits and to scan for a bit. These instructions are discussed in Section 9.5. Section 9.6 gives some examples to illustrate the application of logical and shift/rotate instructions. The chapter concludes with a summary.

9.1 Logical Instructions

Logical instructions manipulate logical data just like the arithmetic instructions manipulate arithmetic data (e.g., integers) with operations such as addition and subtraction. The logical data can take one of two possible values: `true` or `false`.

As the logical data can assume only one of two values, a single bit is sufficient to represent these values. Thus, all logical instructions that we discuss here operate on a bit-by-bit basis. By convention, if the value of the bit is 0, it represents `false`, and a value of 1 represents `true`.

The assembly language provides logical operators in the logical family of instructions. There is a total of five logical instructions: `and`, `or`, `not`, `xor`, and `test`. Except for the `not` operator, all of the logical operators are binary operators (i.e., they require two operands). These instructions operate on 8-, 16-, or 32-bit operands.

All of these logical instructions affect the status flags. Since operands of these instructions are treated as a sequence of independent bits, these instructions do not generate carry or overflow. Therefore, the carry (CF) and overflow (OF) flags are cleared, and the status of the auxiliary flag (AF) is undefined.

Only the remaining three arithmetic flags—the zero flag (ZF), the sign flag (SF), and the parity flag (PF)—record useful information about the result of these logical instructions. Since we discussed these instructions in Chapter 4, we look at their typical use in the following subsections.

9.1.1 The `and` Instruction

The `and` instruction is useful mainly in three situations:

1. To support compound logical expressions and bitwise `and` operations of high-level languages;
2. To clear one or more bits of a byte, word, or doubleword;
3. To isolate one or more bits of a byte, word, or doubleword.

The use of the `and` instruction to express compound logical expressions and to implement bitwise `and` operations is discussed in Section 9.4. Here we concentrate on how `and` can be used to clear or isolate selected bits of an operand.

Clearing Bits

If you look at the truth table of the `and` operation (see page 88), you notice that the source b_i acts as a *masking* bit: if the masking bit is 0, the output is 0 no matter what the other input bit is; if the masking bit is 1, the other input bit is passed to the output. Consider the following example:

```
             AL = 11010110  ← operand to be manipulated
             BL = 11111100  ← mask byte
and   AL,BL = 11010100
```

Here, AL contains the operand to be modified by bit manipulation and BL contains a set of masking bits. Let us say that we want to force the least significant two bits to 0 without altering any of the remaining 6 bits. We select our mask in BL such that it contains 0's in those two bit positions and 1's in the remainder of the masking byte. As you can see from this example, the `and` instruction produces the desired result.

Here are some more examples that utilize the bit clearing capability of the `and` instruction.

Example 9.1 *Even-parity generation (partial code).*
Let us consider generation of even parity. Assume that the most significant bit of a byte represents the parity bit; the rest of the byte stores the data bits. The parity bit can be set or cleared so as to make the number of 1's in the whole byte even.

If the number of 1's in the least significant seven bits is even, the parity bit should be 0. Assuming that the byte to be parity-encoded is in the AL register, the following statement

```
and     AL,7FH
```

clears the parity bit without altering the remaining seven bits. Notice that the mask 7FH has a 0 only in the parity bit position. □

Example 9.2 *ASCII-to-numeric conversion of digit characters.*
In this example, we convert an ASCII decimal digit character to its equivalent 8-bit binary number. To see how this can be done by using the `and` instruction, take a look at the relationship between the ASCII code and the 8-bit binary representation of the 10 digits shown in Table 9.1.

It is clear from this table that if we mask out the third and fourth bits (from left) in the ASCII code byte, we can transform the byte into an equivalent 8-bit unsigned binary number representation.

In fact, we can mask out all of the upper four bits without worry, which is what the following code does. If AL has the ASCII code of a decimal digit, the statement

```
and     AL,0FH
```

produces the desired result in AL. □

Isolating Bits

Another typical use of the `and` instruction is to isolate selected bits for testing. This is done by masking out all the other bits, as shown in the next example.

Table 9.1 ASCII-to-Binary Conversion of Digits

Decimal digit	ASCII code (in binary)	8-bit binary code (in binary)
0	0011 0000	0000 0000
1	0011 0001	0000 0001
2	0011 0010	0000 0010
3	0011 0011	0000 0011
4	0011 0100	0000 0100
5	0011 0101	0000 0101
6	0011 0110	0000 0110
7	0011 0111	0000 0111
8	0011 1000	0000 1000
9	0011 1001	0000 1001

Example 9.3 *Finding an odd or even number.*

In this example, we want to find out if the unsigned 8-bit number in the AL register is an odd or an even number. A simple test to determine this is to check the least significant bit of the number: if this bit is 1, it is an odd number; otherwise, an even number.

Here is the code to perform this test using the and instruction.

```
      and    AL,1      ; mask = 00000001B
      jz     even_number
odd_number:
          . . .
      <code for processing odd number>
          . . .
even_number:
          . . .
      <code for processing even number>
          . . .
```

If AL has an even number, the least significant bit of AL is 0. Therefore,

```
and    AL,1
```

would produce a zero result in AL and sets the zero flag. The jz instruction is then used to test the status of the zero flag and to selectively execute the appropriate code fragment. This example shows the use of and to isolate a bit—the least significant bit in this case. □

9.1.2 The `or` Instruction

Like the `and` instruction, the `or` instruction is useful in two applications:

1. To support compound logical expressions and bitwise or operations of high-level languages;
2. To set one or more bits of a byte, word, or doubleword.

The use of the `or` instruction to express compound logical expressions and to implement bitwise or operations is discussed in Section 9.4. We now discuss how the `or` instruction can be used to set a given set of bits.

As you can see from the truth table for the `or` operation (see page 88), when the source b_i is 0, the other input is passed on to the output; when the source b_i is 1, the output is forced to take a value of 1 irrespective of the other input. This property is used to set bits in the output. This is illustrated in the following example.

```
                AL = 11010110B  ← operand to be manipulated
                BL = 00000011B  ← mask byte
                     ---------
or           AL,BL = 11010111B
```

The mask value in BL causes the least significant two bits to change to 1. Here are some examples that illustrate the use of the `or` instruction.

Example 9.4 *Even-parity encoding (partial code).*
Consider the even-parity encoding discussed in Example 9.1 (on page 273). If the number of 1's in the least significant 7 bits is odd, we have to make the parity bit 1 so that the total number of 1's is even. This is done by

```
or      AL,80H
```

assuming that the byte to be parity-encoded is in the AL register. This `or` operation forces the parity bit to 1 while leaving the remainder of the byte unchanged. □

Example 9.5 *Conversion of digits to ASCII characters.*
This is the counterpart of Example 9.2 on page 273. Here we would like to convert an 8-bit unsigned binary number (between 0 and 9, both inclusive) to the corresponding ASCII character code. Such a conversion is often required to print or display numbers.

The conversion process involves making the third and fourth bits (from left) of the binary number 1's (refer to Table 9.1 on page 274). If the AL register has the binary number to be converted, the instruction

```
or      AL,30H
```

will perform the desired conversion. Note that our mask input 00110000B (30H) will change the two bits to 1's without affecting the remaining 6 bits. □

Cutting and Pasting Bits

The `and` and `or` instructions can be used together to “cut and paste” bits from two or more operands. We have already seen that `and` can be used to isolate selected bits—analogous to the “cut” operation. The `or` instruction can be used to “paste” the bits. For example, the following code creates a new byte in AL by combining odd bits from AL and even bits from BL registers.

```
and    AL,55H   ; cut odd bits
and    BL,0AAH  ; cut even bits
or     AL,BL    ; paste them together
```

The first `and` instruction selects only the odd bits from the AL register by forcing all even bits to 0 by using the mask 55H (01010101B). The second `and` instruction selects the even bits by using the mask AAH (10101010B). The `or` instruction simply pastes these two bytes together to produce the desired byte in the AL register.

9.1.3 The `xor` Instruction

The `xor` instruction is useful mainly in three different situations:

1. To support compound logical expressions of high-level languages;
2. To toggle one or more bits of a byte, word, or doubleword;
3. To initialize registers to zero.

The use of the `xor` instruction to express a compound logical expression is discussed in Section 9.4. Here we focus on the use of `xor` to toggle bits and initialize registers to zero.

Toggling Bits

Using the `xor` instruction, we can toggle a specific set of bits. To do this, the mask should have 1 in the bit positions that are to be flipped. The following example illustrates this application of the `xor` instruction.

Example 9.6 *Parity conversion.*

Suppose we want to change the parity encoding of incoming data—if even parity, change to odd parity, and vice versa. To accomplish this change, all we have to do is flip the parity bit, which can be done by

```
xor    AL,80H
```

Thus, an even-parity encoded ASCII character A—01000001B—is transformed into odd-parity encoding, as shown below:

$$\begin{array}{rll} & \texttt{01000001B} & \leftarrow \text{even-parity encoded ASCII character A} \\ \text{xor} & \underline{\texttt{10000000B}} & \leftarrow \text{mask byte} \\ & \texttt{11000001B} & \leftarrow \text{odd-parity encoded ASCII character A} \end{array}$$

Notice that if we perform the same xor operation on odd-parity encoding of A, we get back the even-parity encoding! This is an interesting property of the xor operation: xoring twice gives back the original value. This is not hard to understand, as xor behaves like the not operation by selectively flipping bits. This property is used in the following example to encrypt a byte. □

Example 9.7 *Encryption of data.*

Data encryption is useful in applications that deal with sensitive data. We can write a simple encryption program by using the xor instruction. The idea is that we will use the encryption key as the mask byte of the xor instruction as shown below. Assume that the byte to be encrypted is in the AL register and the encryption key is A6H.

```
; read a data byte into AL
xor    AL,0A6H
; write the data byte back from AL
```

Suppose we have received character B, whose ASCII code is 01000010B. After encryption, the character becomes d in ASCII, as shown below.

```
01000010B  ← ASCII character B
00100110B  ← encryption key (mask)
---------
01100100B  ← ASCII character d
```

An encrypted data file can be transformed back into normal form by running the encrypted data through the same encryption process again. To continue with our example, if the above encrypted character code 64H (representing d) is passed through the encryption procedure, we get 42H, which is the ASCII code for the character B. □

Initialization of Registers

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 the contents of the AX register are. To see why this is so, look at the truth table for the xor operation given on page 88. Since we are using the same operand as both inputs, the input can be either both 0 or 1. In both cases, the result bit is 0—see the first and last rows of the xor truth table.

These two instructions, however, are not exactly equivalent. The xor instruction affects flags, whereas the mov instruction does not. Of course, 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.

9.1.4 The `not` Instruction

The not instruction is used for complementing bits. Its main use is in supporting logical expressions of high-level languages (see the discussion in Section 9.4).

Another possible use for the not instruction is to compute 1's complement. Recall that 1's complement of a number is simply the complement of the number. Since most systems use the 2's complement number representation system, generating the 2's complement of an 8-bit signed number using not involves

```
not    AL
inc    AL
```

However, the instruction set also provides the neg instruction to reverse the sign of a number. Thus, the not instruction is not used for this purpose.

9.1.5 The `test` Instruction

The test instruction is the logical equivalent of the compare (cmp) instruction. It performs the logical and operation but, unlike the and instruction, test does not alter the destination operand. That is, test is a nondestructive and instruction.

This instruction is used only to update the flags, and a conditional jump instruction normally follows it. For instance, in Example 9.3 on page 274, the instruction

```
and   AL,1
```

destroys the contents of the AL register. If our purpose is to test whether the unsigned number in the AL register is an odd number, we can do this using test without destroying the original number. For convenience, the example is reproduced below with the test instruction.

```
      test   AL,1     ; mask = 00000001B
      jz     even_number
odd_number:
           . . .
even_number:
           . . .
```

9.2 Shift Instructions

The instruction set provides two types of shift instructions: one type for logical shifts, and the other for arithmetic shifts. The logical shift instructions are

shl (SHift Left)
shr (SHift Right)

and the arithmetic shift instructions are

sal (Shift Arithmetic Left)
sar (Shift Arithmetic Right)

Another way of looking at these two types of shift instructions is that the logical type instructions work on unsigned binary numbers, and the arithmetic type works on signed binary numbers. We will get back to this discussion later in this section.

Effect on Flags

As in the logical instructions, the auxiliary flag is undefined following a shift instruction. The carry flag (CF), zero flag (ZF), and parity flag (PF) are updated to reflect the result of a shift instruction. The CF always contains the bit last shifted out of the operand. The OF is undefined following a multibit shift. In a single-bit shift, OF is set if the sign bit has been changed as a result of the shift operation; OF is cleared otherwise. The OF is rarely tested in a shift operation; we often test the CF and ZF flags.

9.2.1 Logical Shift Instructions

Since we discussed the logical shift instructions in Chapter 4, we discuss their usage here. These instructions are useful mainly in two situations:

1. To manipulate bits;
2. To multiply and divide unsigned numbers by a power of 2.

Bit Manipulation

The shift operations provide flexibility to manipulate bits as illustrated by the following example.

Example 9.8 *Another encryption example.*

Consider the encryption example discussed on page 277. In this example, we use the following encryption algorithm: encrypting a byte involves exchanging the upper and lower nibbles (i.e., 4 bits). This algorithm also allows the recovery of the original data by applying the encryption twice, as in the xor example on page 277.

Assuming that the byte to be encrypted is in the AL register, the following code implements this algorithm:

```
; AL contains the byte to be encrypted
mov    AH,AL
shl    AL,4     ; move lower nibble to upper
shr    AH,4     ; move upper nibble to lower
or     AL,AH    ; paste them together
; AL has the encrypted byte
```

To understand this code, let us trace the execution by assuming that AL has the ASCII character A. Therefore,

```
AH = AL = 01000001B
```

Table 9.2 Doubling and Halving of Unsigned Numbers

Binary number	Decimal value
00011100	28
00111000	56
01110000	112
11100000	224
10101000	168
01010100	84
00101010	42
00010101	21

The idea is to move the upper nibble to lower in the AH register, and the other way around in the AL register. To do this, we use the `shl` and `shr` instructions. The `shl` instruction replaces the shifted bits by 0's and after the `shl`

```
    AL = 00010000B
```

Similarly, `shr` introduces 0's in the vacated bits on the left. Thus, after the `shr` instruction

```
    AH = 00000100B
```

The `or` instruction pastes these two bytes together, as shown below:

```
                   AL = 00010000B
                   AH = 00000100B
                        ---------
    or         AL,AH = 00010100B
```

We will show in Section 9.3.1 that this can be done better by using a rotate instruction (see Example 9.9 on page 284). □

Multiplication and Division

Shift operations are very effective in performing doubling or halving of unsigned binary numbers. More generally, they can be used to multiply or divide unsigned binary numbers by a power of 2.

In the decimal number system, we can easily perform multiplication and division by a power of 10. For example, if we want to multiply 254 by 10, we will simply append a 0 at the right (analogous to shifting left by a digit with the vacated digit receiving a 0). Similarly, division of 750 by 10 can be accomplished by throwing away the 0 on the right (analogous to right shift by a digit).

Since computers use the binary number system, they can perform multiplication and division by a power of 2. This point is further clarified in Table 9.2. The first half of this table shows how shifting a binary number to the left by one bit position results in multiplying it by 2. Note that the vacated bits are replaced by 0's. This is exactly what the `shl` instruction does. Therefore, if we want to multiply a number by 8 (i.e., 2^3), we can do so by shifting the number left by three bit positions.

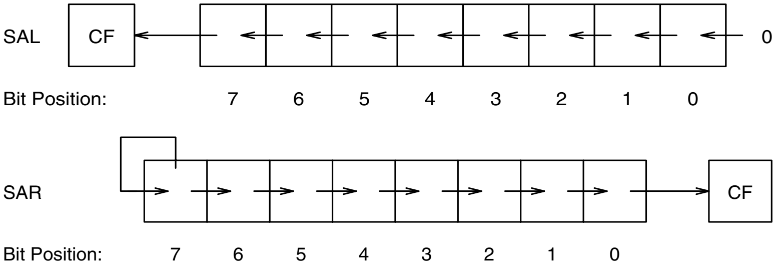
Similarly, as shown in the second half of the table, shifting right by one bit position is equivalent to dividing by 2. Thus, we can use the `shr` instruction to perform division by a power of 2. For example, to divide a number by 32 (i.e., 2^5), we right-shift the number by five bit positions. Remember that this division process corresponds to integer division, which discards any fractional part of the result.

9.2.2 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.



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 9.3 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, three bits are sufficient to represent this range. If we use

Table 9.3 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

a byte to represent the same range, the number is sign-extended by copying the sign bit into the higher-order five bits, as shown below.

$$+3 = \overbrace{00000}^{\substack{\textit{sign bit}\\ \textit{copied}}}011\text{B}$$

$$-3 = \overbrace{11111}^{\substack{\textit{sign bit}\\ \textit{copied}}}101\text{B}$$

Clearly, doubling a signed number is no different than doubling an unsigned number. Thus, no special shift left instruction is needed for the signed numbers. In fact, `sal` and `shl` are one and the same instruction—`sal` is an alias for `shl`.

Halving Signed Numbers

Can we also forget about treating the signed numbers separately 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 9.4. The `sar` instruction does precisely this—the sign bit is copied into the vacated bit on the left.

Remember that the right-shift operation performs integer division. For example, right-shifting 00001011B (+11D) by a bit results in 00000101B (+5D).

9.2.3 Why Use 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 32D. Using the `mul` instruction, we can write

Table 9.4 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

```
; 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. Thus, this code is both more space- and time-efficient than the `mul` version.

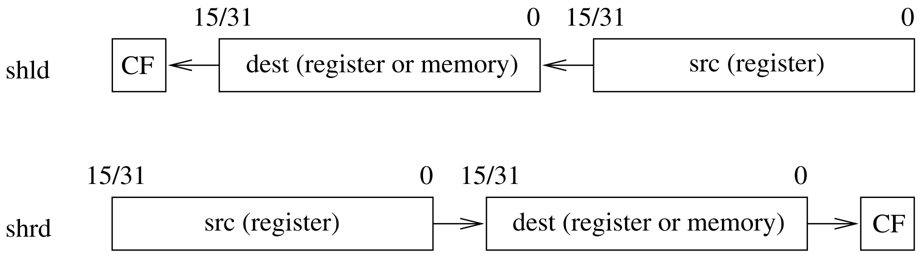
9.2.4 Doubleshift Instructions

The instruction set has two doubleshift instructions for 32-bit and 64-bit shifts. These two instructions operate on either word or doubleword operands and produce a word or doubleword result, respectively. The doubleshift instructions require three operands, as shown below:

```
shld    dest,src,count  ; left-shift
shrd    dest,src,count  ; right-shift
```

`dest` and `src` can be either a word or a doubleword. While the `dest` operand can be in a register or memory, the `src` operand must be in a register. The shift `count` can be specified as in the shift instructions—either as an immediate value or in the CL register.

A significant difference between shift and doubleshift instructions is that the `src` operand supplies the bits in doubleshift 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 doubleshift 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 doubleshift instructions (see Example 9.10 on page 285).

9.3 Rotate Instructions

A drawback with the shift instructions is that the bits shifted out are lost. There are situations where we want to keep these bits. The doubleshift instructions provide this capability on word or doubleword operands. The rotate family of instructions remedies this drawback on a variety of operands. These instructions can be divided into two types: rotate without involving the carry flag (CF), or rotate through the carry flag. Since we presented these two types of rotate instructions in Section 4.5.6, we discuss their typical usage in the next two subsections.

9.3.1 Rotate Without Carry

The rotate instructions are useful in rearranging bits of a byte, word, or doubleword. This is illustrated below by revisiting the data encryption example given on page 279.

Example 9.9 *Encryption example revisited.*

In Example 9.8, we encrypted a byte by interchanging the upper and lower nibbles. This can be done easily either by

```
mov    CL,4
ror    AL,CL
```

or by

```
mov    CL,4
rol    AL,CL
```

This is a much simpler solution than the one using shifts. □

9.3.2 Rotate Through Carry

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, as illustrated by the following example.

Example 9.10 *Shifting 64-bit numbers.*
We have seen that multiplication and division by a power of 2 is 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 doubleshift 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    ECX,4      ; 4 bit shift
    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
```

Doubleshift 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    ECX,4      ; 4 bit shift
    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
```

Doubleshift version:

```
        shrd   EAX,EDX,4 ; EDX is unaffected by shld
        shr    EDX,4
```

□

As you can see from these examples, we can avoid looping by using the doubleshift instructions.

9.4 Logical Expressions in High-Level Languages

This section discusses Boolean data representation and evaluation of compound logical expressions. Boolean variables can assume one of two values: `true` or `false`.

9.4.1 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, which does not provide an explicit Boolean data type, any data variable can be used in a logical expression to represent Boolean data. The rules mentioned above apply: if the value is 0, it is treated as `false` and any nonzero value is treated as `true`.

9.4.2 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 9.1.

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 later (see Section 9.4.4). 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).

```
      cmp    WORD PTR [BP-12],0  ; X = false?
      je     false1              ; if so, (X && Y) = false
      or     CX,CX               ; Y = false?
      je     false1
      mov    AX,1                ; (X && Y) = true
      jmp    SHORT skip1
false1:
      xor    AX,AX               ; (X && Y) = false
skip1:
      not    AX                  ; AX = ~(X && Y)
      push   AX                  ; save ~(X && Y)
      ; now evaluate the second term
      or     CX,CX               ; Y = true?
      jne    true2               ; if so, (Y || Z) = true
      cmp    WORD PTR [BP-14],0  ; Z = false?
      je     skip2
true2:
      mov    AX,1                ; (X || Y) = true
      jmp    SHORT skip3
skip2:
      xor    AX,AX               ; (X || Y) = false
skip3:
      pop    DX                  ; DX = ~(X && Y)
      xor    DX,AX               ; ~(X && Y) ^ (Y || Z)
      je     end_if              ; if zero, whole exp. false
if_body:
      mov    AX,CX               ; AX = Y
      add    AX,WORD PTR [BP-14] ; AX = Y + Z
      mov    WORD PTR [BP-12],AX ; X = Y + Z
end_if:
          . . .
```

Figure 9.1 Assembly language code for the example logical expression.

9.4.3 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 of 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 9.5 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.

Table 9.5 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`

9.4.4 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.

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 9.2 for this logical expression.

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
```

```
        cmp    ch,'Z'
        mov    AL,0
        ja     skip1
        inc    AX
skip1:
        mov    DL,AL
        cmp    ch,'A'
        mov    AL,0
        jb     skip2
        inc    AX
skip2:
        and    AL,DL
        mov    CL,AL
        cmp    ch,'z'
        mov    AL,0
        ja     skip3
        inc    AX
skip3:
        mov    DL,AL
        cmp    ch,'a'
        mov    AL,0
        jb     skip4
        inc    AX
skip4:
        and    AL,DL
        or     AL,CL
        or     AL,AL
        je     skip_if
        << if body here >>
skip_if:
        << code following the if >>
```

Figure 9.2 Assembly language code for full evaluation.

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.

```
        cmp    ch,'a'
        jb     skip1
        cmp    ch,'z'
        jbe    skip2
skip1:
        cmp    ch,'A'
        jb     skip_if
        cmp    ch,'Z'
        ja     skip_if
skip2:
        << if body here >>
skip_if:
        << code following the if >>
```

Figure 9.3 Assembly language code for partial evaluation.

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 9.3. 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.

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.

Of course, when full evaluation is used, we can rewrite the last condition to avoid the divide-by-zero error as

```
if (X > 0)
    if (Y/X > 100)
        . . .
```

9.5 Bit Instructions

The instruction set has bit test and modification instructions as well as bit scan instructions. This section discusses these two groups of instructions. An example that uses these instructions is given later (see Example 9.12).

9.5.1 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 all four instructions is the same. We use the `bt` instruction to illustrate the format of these instructions.

```
bt    operand,bit_pos
```

where `operand` can be a word or doubleword located either in a register or in 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.

9.5.2 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 in 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.

9.6 Illustrative Examples

This section presents three examples that use the shift and rotate family of instructions.

Example 9.11 *Multiplication using only shifts and adds.*

The objective of this example is to show how multiplication can be done entirely by using the 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 below:

Program 9.1 Multiplication of two 8-bit numbers using only shifts and adds

```
;8-bit multiplication using shifts   SHL_MLT.ASM
;
;        Objective: To multiply two 8-bit unsigned numbers
;                   using SHL rather than MUL instruction.
;            Input: Requests two unsigned numbers from user.
;           Output: Prints the multiplication result.
%include "io.mac"
.DATA
input_prompt   db  'Please input two short numbers: ',0
out_msg1       db  'The multiplication result is: ',0
query_msg      db  'Do you want to quit (Y/N): ',0

.CODE
        .STARTUP
read_input:
        PutStr  input_prompt ; request two numbers
        GetInt  AX           ; read the first number
        GetInt  BX           ; read the second number
        call    mult8        ; mult8 uses SHL instruction
        PutStr  out_msg1
        PutInt  AX           ; mult8 leaves result in AX
        nwln
        PutStr  query_msg    ; query user whether to terminate
        GetCh   AL           ; read response
        cmp     AL,'Y'       ; if response is not 'Y'
        jne     read_input   ; repeat the loop
done:                        ; otherwise, terminate program
        .EXIT

;------------------------------------------------------------
; mult8 multiplies two 8-bit unsigned numbers passed on to
; it in registers AL and BL. The 16-bit result is returned
; in AX. This procedure uses only the SHL instruction to do
; the multiplication. All registers, except AX, are preserved.
;------------------------------------------------------------
mult8:
       push    CX             ; save registers
       push    DX
       push    SI
       xor     DX,DX          ; DX = 0 (keeps mult. result)
       mov     CX,7           ; CX = # of shifts required
       mov     SI,AX          ; save original number in SI
repeat1:       ; multiply loop - iterates 7 times
       rol     BL,1           ; test bits of number2 from left
       jnc     skip1          ; if 0, do nothing
       mov     AX,SI          ; else, AX = number1*bit weight
       shl     AX,CL
       add     DX,AX          ; update running total in DX
skip1:
       dec     CX
       jnz     repeat1
       rol     BL,1           ; test the rightmost bit of AL
       jnc     skip2          ; if 0, do nothing
       add     DX,SI          ; else, add number1
skip2:
       mov     AX,DX          ; move final result into AX
       pop     SI             ; restore registers
       pop     DX
       pop     CX
       ret
```

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 292. The multiply loop (lines 43–51) tests the most significant 7 bits of the multiplier. The least significant bit is tested on lines 52 and 53. Notice that the procedure uses `rol` rather than `shl` to test each bit (lines 44 and 52). The use of `rol` automatically restores the BL register after 8 rotates. □

Example 9.12 *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 43–51) 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.

```
;-----------------------------------------------------------------
; mult8 multiplies two 8-bit unsigned numbers passed on to
; it in registers AL and BL. The 16-bit result is returned
; in AX. This procedure uses only the SHL instruction to do the
; multiplication. All registers, except AX, are preserved.
; Demonstrates the use of bit instructions BSF and BTC.
;-----------------------------------------------------------------
mult8:
       push    CX              ; save registers
       push    DX
       push    SI
       xor     DX,DX           ; DX = 0 (keeps mult. result)
       mov     SI,AX           ; save original number in SI
repeat1:
       bsf     CX,BX           ; returns first 1 bit position in CX
       jz      skip1           ; if ZF=1, no 1 bit in BX - done
       mov     AX,SI           ; else, AX = number1*bit weight
       shl     AX,CL
       add     DX,AX           ; update running total in DX
       btc     BX,CX           ; complement the bit found by BSF
       jmp     repeat1
skip1:
       mov     AX,DX           ; move final result into AX
       pop     SI              ; restore registers
       pop     DX
       pop     CX
       ret
```

The modified loop (lines 14–21) 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's 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. □

Example 9.13 *Conversion from octal to binary.*
An algorithm for converting octal numbers to binary is given in Appendix A. The main program is similar to that in the last example. The procedure `to_binary` receives an octal number as a character string via BX and the 8-bit binary value is returned in AL. The pseudocode of this procedure is as follows:

```
to_binary (octal_string)
    binary_value := 0
    for (i = 0 to 3)
        if (octal_string[i] = NULL)
            goto finished
        end if
        digit := numeric(octal_string[i])
        binary_value := binary_value * 8 + digit
    end for
finished:
end to_binary
```

The function `numeric` converts a digit character to its numeric equivalent. The program is shown in Program 9.2. Note that we use the `shl` instruction to multiply by 8 (line 54). The rest of the code follows the pseudocode.

Program 9.2 Octal-to-binary conversion

```
 1:  ;Octal-to-binary conversion using shifts   OCT_BIN.ASM
 2:  ;
 3:  ;        Objective: To convert an 8-bit octal number to the
 4:  ;                   binary equivalent using shift instruction.
 5:  ;            Input: Requests an 8-bit octal number from user.
 6:  ;           Output: Prints the decimal equivalent of the input
 7:  ;                   octal number.
 8:  %include "io.mac"
 9:
10:  .DATA
11:  input_prompt   db  'Please input an octal number: ',0
12:  out_msg1       db  'The decimal value is: ',0
```

```
query_msg      db  ’Do you want to quit (Y/N): ’,0

.UDATA
octal_number   resb  4        ; to store octal number

.CODE
      .STARTUP
read_input:
      PutStr  input_prompt    ; request an octal number
      GetStr  octal_number,4  ; read input number
      mov     EBX,octal_number ; pass octal # pointer
      call    to_binary    ; returns binary value in AX
      PutStr  out_msg1
      PutInt  AX           ; display the result
      nwln
      PutStr  query_msg    ; query user whether to terminate
      GetCh   AL           ; read response
      cmp     AL,’Y’       ; if response is not ’Y’
      jne     read_input   ; read another number
done:                      ; otherwise, terminate program
      .EXIT

;-----------------------------------------------------------
; to_binary receives a pointer to an octal number string in
; EBX register and returns the binary equivalent in AL (AH is
; set to zero). Uses SHL for multiplication by 8. Preserves
; all registers, except AX.
;-----------------------------------------------------------
to_binary:
      push    EBX          ; save registers
      push    CX
      push    DX
      xor     EAX,EAX      ; result = 0
      mov     CX,3         ; max. number of octal digits
repeat1:
      ; loop iterates a maximum of 3 times;
      ; but a NULL can terminate it early
      mov     DL,[EBX]     ; read the octal digit
      cmp     DL,0         ; is it NULL?
      je      finished     ; if so, terminate loop
      and     DL,0FH       ; else, convert char. to numeric
      shl     AL,3         ; multiply by 8 and add to binary
      add     AL,DL
      inc     EBX          ; move to next octal digit
```

```
            dec     CX              ; and repeat
            jnz     repeat1
    finished:
            pop     DX              ; restore registers
            pop     CX
            pop     EBX
            ret
```

9.7 Summary

We discussed logical, shift, and rotate instructions available in the assembly language. Logical instructions are useful to implement bitwise logical operators and Boolean expressions. However, in some instances Boolean expressions can also be implemented by using conditional jump instructions without using the logical instructions.

Shift and rotate instructions provide flexibility to bit manipulation operations. There are two types of shift instructions: one type works on logical and unsigned data, and the other type is meant for signed data. There are also two types of rotate instructions: rotate without, or rotate through carry. Rotate through carry is useful in shifting multiword data.

There are also two doubleshift instructions that work on either word or doubleword operands. In addition, four instructions for testing and modifying bits and two instructions to scan for a bit are available.

We discussed how the logical and shift instructions are used to implement logical expressions and bitwise logical operations in high-level languages. Logical expressions can be evaluated in one of two ways: partial evaluation or full evaluation. In partial evaluation, evaluation of a logical expression is stopped as soon as the final result of the complete logical expression is known. In full evaluation, the complete logical expression is evaluated. Partial evaluation has the obvious advantage of reduced evaluation time. Partial evaluation also has an important advantage of avoiding error conditions in some expressions.

Shift instructions can be used to multiply or divide by a number that is a power of 2. Shifts for such arithmetic operations are more efficient than the corresponding arithmetic instructions.

9.8 Exercises

9–1 What is the difference between `or` and `xor` logical operators?

9–2 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 see if you can implement by using only three `not` and one `or` operation.

9–3 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 see if you can implement by using only three `not` and one `and` operation.

9–4 Explain how and and or logical operations can be used to "cut and paste" a specific set of bits.

9–5 Suppose the instruction set did not include the not instruction. How do you implement it using only and and or instructions?

9–6 Can we use the logical shift instructions shl and shr on signed data?

9–7 Can we use the arithmetic shift instructions sal and sar on unsigned data?

9–8 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 instructions.

9–9 Repeat the above exercise using only the shift/rotate instructions.

9–10 Show an assembly language program fragment to complement only the odd bits of the AL register using only the logical operations.

9–11 Repeat the above exercise using only the shift/rotate instructions.

9–12 Explain the difference between bitwise and and logical and operations. Use an example to illustrate your point.

9–13 Repeat the above exercise for the or operation.

9–14 Describe the two methods of evaluating logical expressions.

9–15 Discuss the advantages of partial evaluation over full evaluation of logical expressions.

9–16 Consider the following statement:

> **if** ((X > 0) AND (X−Y > 0) AND ((X/Y)+(Z/(X−Y)) < 2))
> **then**
> . . .

Suppose your compiler uses only full evaluation of logical expressions. Modify the **if** statement so that it works without a problem for all values of X and Y.

9–17 Fill in the blanks in the following table:

	Before execution		After execution			
Instruction	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				

9–18 Assuming that the contents of the AL register is treated as 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		

9–19 Assuming that the CL register is initialized to 3, 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	?		

9.9 Programming Exercises

9–P1 Write a procedure to perform hexadecimal-to-binary conversion. Use only shift instructions for multiplication. Assume that signed 32-bit numbers are used. Test your procedure by writing a main program that reads a hexadecimal number as a character string and converts it to an equivalent binary number by calling the procedure. Finally, the main program should display the decimal value of the input by using `PutLInt`.

9–P2 Modify the octal-to-binary conversion program shown in Program 9.2 to include error checking for nonoctal input. For example, digit 8 in the input should be flagged as an error. In case of error, terminate the program.

9–P3 In Appendix A, we discuss the format of short floating-point numbers. Write a program that reads the floating-point internal representation from a 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

9–P4 Modify the program for the last exercise to work with the long floating-point representation.

9–P5 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 1's. Write a program to read an integer (should accept both positive and negative numbers) from the user and outputs whether it contains an odd or even number 1's. Your program should also print the number of 1's in the binary representation.

9–P6 Write a procedure `abs` that receives an 8-bit signed number in the AL register and returns its absolute value back in the same register. Remember that negative numbers are stored in 2's complement representation. It is simple to write such a procedure using arithmetic instructions. In this exercise, however, you are asked to write this procedure using *only the logical* instructions.

9–P7 Repeat the last exercise by using *only the shift and rotate* instructions.

9–P8 Display the status of the flags register. In particular, display the status of the carry, parity, zero, and sign flags. (See Chapters 3 and 7 for details on the flags register.) For each flag, use the format “flag = value”. For example, if carry flag is set, your program should display “CF = 1”. Each flag status should be displayed on a separate line. Before terminating your program, the four flag bits should be complemented and stored back in the flags register.

9–P9 Repeat the last exercise using the `lahf` and `sahf` instructions. The details of these instructions are as follows: the `lahf` (Load AH from Flags register) copies the lower-order byte of the flags register into the AH register. The `sahf` (Store AH to Flags register) stores the contents of the AH register in the lower-order byte of the flags register.

Chapter 10

String Processing

Objectives

- To discuss string representation schemes
- To describe string manipulation instructions
- To illustrate the use of indirect procedure calls
- To demonstrate the performance advantage of string instructions

A string is a sequence of characters. String manipulation is an important aspect of any programming task. Text processing applications, for example, heavily use string manipulation functions. Several high-level languages provide procedures or routines for string processing. This is the focus of this chapter.

Strings are represented in a variety of ways. Section 10.1 discusses some of the representation schemes used to store strings. The instruction set supports string processing by a special set of instructions. These instructions are described in Section 10.2. Several examples are presented in Section 10.3. The purpose of these examples is to illustrate the use of string instructions in developing procedures for string processing. Section 10.4 describes a program to test the procedures developed in the previous section. A novelty of this program is that it demonstrates the use of indirect procedure calls.

String processing procedures can be developed without using the string instructions. However, using the string instructions can result in more efficient code. The efficacy of the string instructions is demonstrated in Section 10.5. The chapter concludes with a summary.

10.1 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 of a string refers to the number of characters in the string. In this 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 that the program will ever handle. A potential problem with this representation is that we should anticipate this value, which may cause difficulties with program maintenance. A further disadvantage of using fixed-length representation is that memory space is wasted if majority of the strings are shorter than the 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. This length attribute is given in one of two ways:

1. Explicitly storing string length, or
2. Using a sentinel character.

These two methods are discussed next.

10.1.1 Explicitly Storing String Length

In this method, 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.

Table 10.1 String Instructions

Mnemonic	Meaning	Operand(s) required
`LODS`	LOaD String	source
`STOS`	STOre String	destination
`MOVS`	MOVe String	source & destination
`CMPS`	CoMPare Strings	source & destination
`SCAS`	SCAn String	destination

10.1.2 Using a Sentinel Character

In this method, strings are stored with a trailing sentinel character to delimit a string. Therefore, there is no need to store the string length explicitly. The assumption here is that the sentinel character is a special character that does not appear within a string. We normally use a special, nonprintable character that does not appear in strings. We have been using the ASCII NULL-character (00H) to terminate strings. Such NULL-terminated strings are called *ASCIIZ strings*. Here are some examples:

```
string1    DB    'This is OK',0
string2    DB    'Price = $9.99',0
```

The C language, for example, uses this representation to store strings. In the remainder of this chapter, we use this representation for strings.

10.2 String Instructions

There are five main string-processing instructions. These can be used to copy a string, to compare two strings, and so on. It is important to note that these instructions are not just for copying strings. We can use them to perform memory-to-memory copy of data. For example, we could use them to copy arrays of integers. The five basic instructions are shown in Table 10.1.

Specifying Operands

As indicated, each string instruction may require a source operand, a destination operand, or both. For 32-bit segments, string instructions use 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, SI and DI registers are used instead of ESI and EDI registers. If both the operands are in the same data segment, we can let both DS and ES point to the data segment to use the string instructions.

Table 10.2 Repetition Prefixes

unconditional repeat	
`rep`	REPeat
conditional repeat	
`repe/repz`	REPeat while Equal REPeat while Zero
`repne/repnz`	REPeat while Not Equal REPeat while Not Zero

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 1. These registers are updated by 2 and 4 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 (discussed in Section 10.2.2).

10.2.1 Repetition Prefixes

String instructions can be repeated by using a repetition prefix. As shown in Table 10.2, the three prefixes are divided into two categories: *unconditional* or *conditional* repetition. None of the flags is affected by these instructions.

rep

This is an unconditional repeat prefix and causes the instruction to repeat according to the value in the ECX register. Note that for 16-bit addresses, CX register is used. The semantics of `rep` are

```
while (ECX $\neq$ 0)
    execute the string instruction;
    ECX := ECX–1;
end while
```

The ECX register is first checked and if it is not 0, only then is the string instruction executed. Thus, if ECX 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 ECX is 0. Thus, with loop, ECX = 0 results in a maximum number of iterations, and usually a jecxz 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 (ECX ≠ 0)
    execute the string instruction;
    ECX := ECX–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 ECX, as in the rep prefix. But the actual number of times the instruction is repeated is determined by the status of ZF. 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 is ZF = 1 as shown below:

```
while (ECX ≠ 0)
    execute the string instruction;
    ECX := ECX–1;
    if (ZF = 1)
    then
        exit loop
    end if
end while
```

10.2.2 Direction Flag

The direction of string operations depends on the value of the direction flag. Recall that this is one of the bits of the flag's register. 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)

Both of these instructions do not require any operands. Each instruction is encoded using a single byte and takes two clock cycles to execute.

Usually, it does not matter whether a string is processed in the forward or backward direction. For sentinel character-terminated strings, 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., move backward) as in the following example.

Initial string →	a	b	c	0	?
After one shift →	a	b	c	0	0
After two shifts →	a	b	c	c	0
After three shifts→	a	b	b	c	0
Final string →	a	a	b	c	0

If we proceed from the head and in the forward direction, only the first character is copied through the string, as shown below:

Initial string →	a	b	c	0	?
After one shift →	a	a	c	0	?
After two shifts →	a	a	a	0	?
After three shifts→	a	a	a	a	?
Final string →	a	a	a	a	a

10.2.3 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 format of the `movs` instruction is:

```
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 of this chapter.

The `movs` instruction is used to copy a value (byte, word, or doubleword) from the source string to the destination string. As mentioned earlier, the source string value is pointed to by DS:ESI and the destination string location is indicated by ES:EDI in memory. After copying, the ESI and EDI 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 32-bit segments. For 16-bit segments, we use the SI and DI registers.

```
movsb — move a byte string
    ES:EDI := (DS:ESI)      ; copy a byte
    if (DF = 0)             ; forward direction
    then
        ESI := ESI+1
        EDI := EDI+1
    else                    ; backward direction
        ESI := ESI–1
        EDI := EDI–1
    end if
Flags affected: none
```

For word and doubleword operands, the index registers are updated by 2 and 4, respectively. This instruction, along with the `rep` prefix, is useful to copy 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
.UDATA
string2    resb    80
.CODE
    .STARTUP
    mov    AX,DS            ; set up ES
    mov    ES,AX            ;  to the data segment
    mov    ECX,strLen       ; strLen includes NULL
    mov    ESI,string1
    mov    EDI,string2
    cld                     ; forward direction
    rep    movsb
```

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 from the source string (pointed to by DS:ESI) in memory to AL (for byte operands—lodsb), AX (for word operands—lodsw), or EAX (for doubleword operands—lodsd).

```
lodsb — load a byte string
        AL := (DS:ESI)      ; copy a byte
        if (DF = 0)         ; forward direction
        then
            ESI := ESI+1
        else                ; backward direction
            ESI := ESI−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 after we describe 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:EDI) in memory.

```
stosb — store a byte string
        ES:EDI := AL        ; copy a byte
        if (DF = 0)         ; forward direction
        then
            EDI := EDI+1
        else                ; backward direction
            EDI := EDI−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 following code initializes array1 with −1.

```
.UDATA
array1    resw    100
.CODE
    .STARTUP
    mov    AX,DS          ; set up ES
    mov    ES,AX          ;  to the data segment
    mov    ECX,100
    mov    EDI,array1
    mov    AX,-1
    cld                   ; forward direction
    rep    stosw
```

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, the following code

```
    mov    ECX,strLen
    mov    ESI,string1
    mov    EDI,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 ESI and EDI registers.

10.2.4 String Compare Instruction

The `cmps` instruction can be used to compare two strings.

```
cmpsb — compare two byte strings
       Compare the two bytes at DS:ESI and ES:EDI and set flags
       if (DF = 0)        ; forward direction
       then
           ESI := ESI+1
```

```
            EDI := EDI+1
        else                    ; backward direction
            ESI := ESI−1
            EDI := EDI−1
        end if
```

Flags affected: As per `cmp` instruction

The `cmps` instruction compares the two bytes, words, or doublewords at DS:ESI and ES:EDI and sets the flags just like the `cmp` instruction. Like the `cmp` instruction, `cmps` performs

```
(DS:ESI) − (ES:EDI)
```

and sets the flags according to the result. The result itself is not stored. We can use conditional jumps like `ja`, `jg`, `jc`, etc. to test the relationship of the two values. As usual, the ESI and EDI registers are updated according to the value of the direction flag and the operand size. The `cmps` instruction is typically used with the `repe/repz` or `repne/repnz` prefix.

The following 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    ECX,strLen
    mov    ESI,string1
    mov    EDI,string2
    cld                    ; forward direction
    repe   cmpsb
```

leaves ESI pointing to `g` in `string1` and EDI to `f` in `string2`. Therefore, adding

```
    dec    ESI
    dec    EDI
```

leaves ESI and EDI pointing to the last 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. A more concrete example is given later (see the string comparison procedure on page 317).

The `repne/repnz` prefix can be used to continue comparison as long as the comparison fails and the loop terminates when a matching value 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    ECX,strLen
    mov    ESI,string1 + strLen - 1
    mov    EDI,string2 + strLen - 1
    std                   ; backward direction
    repne  cmpsb
    inc    ESI
    inc    EDI
```

leaves ESI and EDI pointing to the first character that matches in the backward direction.

10.2.5 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:EDI should point to the string to be searched.

```
scasb — scan a byte string
      Compare AL to the byte at ES:EDI and set flags
      if (DF = 0)          ; forward direction
      then
          EDI := EDI+1
      else                 ; backward direction
          EDI := EDI−1
      end if
Flags affected: As per cmp instruction
```

Like with the `cmps` instruction, the `repe/repz` or `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    ECX,strLen
    mov    EDI,string1
```

```
        mov    AL,'e'        ; character to be searched
        cld                  ; forward direction
        repne  scasb
        dec    EDI
```

This program leaves EDI pointing to e in string1. The following example can be used to skip the 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    ECX,strLen
        mov    EDI,string1
        mov    AL,' '        ; character to be searched
        cld                  ; forward direction
        repe   scasb
        dec    EDI
```

This program leaves EDI pointing to the first nonblank character in string1, which is a in our example.

10.3 Illustrative Examples

We now give some examples to 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. The pointer to a string is received in segment:offset form (i.e., two words from the stack). A string pointer is loaded into either DS and ESI or ES and EDI using the lds or les instructions, the details of which are discussed next.

LDS and LES Instructions

The syntax of these instructions is

```
    lds    register,source
    les    register,source
```

where register is a 32-bit general-purpose register, and source is a pointer to a 48-bit memory operand. The instructions perform the following actions:

```
lds
     register := (source)
           DS := (source + 4)
les
     register := (source)
           ES := (source + 4)
```

The 32-bit value at `source` in memory is copied to `register` and the next 16-bit value (i.e., at `source`+4) is copied to the DS or ES register. Both instructions affect none of the flags. By specifying ESI as the register operand, `lds` can be conveniently used to set up a source string. Similarly, a destination string can be set up by specifying EDI with `les`. For completeness, you should note that `lfs`, `lgs`, and `lss` instructions are available to load the other segment registers.

Examples

We will next present seven simple string processing procedures. Most of these are available in high-level languages such as C. All procedures use the carry flag (CF) to report input error—*not a string*. This error results if the input passed is not a string whose length is less than the `STR_MAX` constant defined in `string.asm`. The carry flag is set (i.e., CF = 1) if there is an input error; otherwise, the carry flag is cleared.

The following constants are defined in `string.asm`:

```
STR_MAX     EQU        128
%define     STRING1    [EBP+8]
%define     STRING2    [EBP+16]
```

Example 10.1 *String length procedure to return the length of* `string1`.

String length is the number of characters in a string, excluding the NULL character. We will 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 to the ES and EDI registers from the stack. `STR_MAX`, the maximum length of a string, is moved into ECX, 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 EDI) and the start of the string available at [EBP+8]. The EAX register is used to return the string length value. This procedure is similar to the C function `strlen`.

```
;-----------------------------------------------------------
;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 EAX with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
str_len:
    enter   0,0
    push    ECX
    push    EDI
    push    ES
    les     EDI,STRING1  ; copy string pointer to ES:EDI
    mov     ECX,STR_MAX  ; need to terminate loop if EDI
                         ; is not pointing to a string
    cld                  ; forward search
    mov     AL,0         ; NULL character
    repne   scasb
    jcxz    sl_no_string ; if ECX = 0, not a string
    dec     EDI          ; back up to point to NULL
    mov     EAX,EDI
    sub     EAX,[EBP+8]  ; string length in EAX
    clc                  ; no error
    jmp     SHORT sl_done
sl_no_string:
    stc                  ; carry set => no string
sl_done:
    pop     ES
    pop     EDI
    pop     ECX
    leave
    ret     8            ; clear stack and return
```

Example 10.2 *String copy procedure to copy* `string2` *to* `string1`.

To copy a string, the `movsb` instruction is used. We use `string2` as the source string and `string1` as the destination string. The `str_len` procedure is used to find the length of the source string `string2`, which is used to set up repeat count in ECX. This value is incremented by 1 to include the NULL character to properly terminate the destination string. C provides a similar function, which can be called as `strcpy(string1,string2)`. The direction of copy is from `string2` to `string1`, as in our assembly language procedure.

```
    ;-----------------------------------------------------------
    ;String copy procedure. Receives two string pointers
    ;(seg:offset) via the stack - string1 and string2.
    ;If string2 is not a string, CF is set;
    ;otherwise, string2 is copied to string1 and the
    ;offeset of string1 is returned in EAX with CF = 0.
    ;Preserves all registers.
    ;-----------------------------------------------------------
    str_cpy:
        enter   0,0
        push    ECX
        push    EDI
        push    ESI
        push    DS
        push    ES
        ; find string length first
        lds     ESI,STRING2  ; src string pointer
        push    DS
        push    ESI
        call    str_len
        jc      sc_no_string
        mov     ECX,EAX      ; src string length in ECX
        inc     ECX          ; add 1 to include NULL
        les     EDI,STRING1  ; dest string pointer
        cld                  ; forward search
        rep     movsb
        mov     EAX,[EBP+8]  ; return dest string pointer
        clc                  ; no error
        jmp     SHORT sc_done
sc_no_string:
        stc                  ; carry set => no string
sc_done:
        pop     ES
        pop     DS
        pop     ESI
        pop     EDI
        pop     ECX
        leave
        ret     16           ; clear stack and return
```

Example 10.3 *String concatenate procedure to concatenate* `string2` *to* `string1`.
This procedure is similar to the `str_cpy` procedure except that copying of `string2` starts from the end of `string1`. To do this, we first move EDI to point to the NULL character of `string1`. This procedure is analogous to the C procedure `strcat`, which can be called as `strcat(string1,string2)`. It concatenates `string2` to `string1`, as in our assembly language procedure.

```
;-----------------------------------------------------------
;String concatenate procedure. Receives two string pointers
;(seg:offset) via the stack - string1 and string2.
;If string1 and/or string2 are not strings, CF is set;
;otherwise, string2 is concatenated to the end of string1
;and the offset of string1 is returned in EAX with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
```

```
str_cat:
    enter   0,0
    push    ECX
    push    EDI
    push    ESI
    push    DS
    push    ES
    ; find string length first
    les     EDI,STRING1 ; dest string pointer
    mov     ECX,STR_MAX ; max string length
    cld                 ; forward search
    mov     AL,0        ; NULL character
    repne   scasb
    jcxz    st_no_string
    dec     EDI         ; back up to point to NULL
    lds     ESI,STRING2 ; src string pointer
    push    DS
    push    ESI
    call    str_len
    jc      st_no_string
    mov     ECX,EAX     ; src string length in ECX
    inc     ECX         ; add 1 to include NULL
    cld                 ; forward search
    rep     movsb
    mov     EAX,[EBP+8] ; return dest string pointer
    clc                 ; no error
    jmp     SHORT st_done
st_no_string:
    stc                 ; carry set => no string
st_done:
    pop     ES
    pop     DS
    pop     ESI
    pop     EDI
    pop     ECX
    leave
    ret     16          ; clear stack and return
```

Example 10.4 *String compare procedure to compare two strings.*

This function uses the `cmpsb` instruction to compare two strings. It returns in EAX a negative value if `string1` is lexicographically less than `string2`, 0 if `string1` is equal to `string2`, and a positive value if `string1` is lexicographically greater than `string2`.

To implement this procedure, we have to find the first occurrence of a character mismatch between the corresponding characters in the two strings (when scanning strings from left to right). The relationship between the strings is the same as that between the two differing characters. When we include the NULL character in this comparison, this algorithm works correctly even when the two strings are of different length.

The str_cmp instruction finds the length of string2 using the str_len procedure. It does not really matter whether we find the length of string2 or string1. We use this value (plus one to include NULL) to control the number of times the cmpsb instruction is repeated. Conditional jump instructions are used to test the relationship between the differing characters to return an appropriate value in the EAX register. The corresponding function in C is strcmp, which can be invoked by strcmp(sting1,string2). This function also returns the same values (negative, 0, or positive value) depending on the comparison.

```
;-----------------------------------------------------------
;String compare procedure. Receives two string pointers
;(seg:offset) via the stack - string1 and string2.
;If string2 is not a string, CF is set;
;otherwise, string1 and string2 are compared and returns a
;a value in EAX with CF = 0 as shown below:
;    EAX = negative value  if string1 < string2
;    EAX = zero            if string1 = string2
;    EAX = positive value  if string1 > string2
;Preserves all registers.
;-----------------------------------------------------------
str_cmp:
    enter   0,0
    push    ECX
    push    EDI
    push    ESI
    push    DS
    push    ES
    ; find string length first
    les     EDI,STRING2  ; string2 pointer
    push    ES
    push    EDI
    call    str_len
    jc      sm_no_string

    mov     ECX,EAX      ; string1 length in ECX
    inc     ECX          ; add 1 to include NULL
    lds     ESI,STRING1  ; string1 pointer
    cld                  ; forward search
    repe    cmpsb
    je      same
    ja      above
below:
    mov     EAX,-1          ; EAX = -1 => string1 < string2
    clc
    jmp     SHORT sm_done
same:
    xor     EAX,EAX         ; EAX = 0 => string match
    clc
    jmp     SHORT sm_done
above:
    mov     EAX,1           ; EAX = 1 => string1 > string2
    clc
    jmp     SHORT sm_done
sm_no_string:
    stc                     ; carry set => no string
sm_done:
    pop     ES
    pop     DS
    pop     ESI
    pop     EDI
    pop     ECX
    leave
    ret     16              ; clear and return
```

Example 10.5 *Character locate procedure to locate* chr *in* string1.

This is another function that uses scasb and is very similar in nature to the str_len procedure. The only difference is that, instead of looking for the NULL character, we will search for the given character chr. It returns a pointer to the position of the first match of chr in string1; if no match is found, a NULL (i.e., 0 value) is returned in EAX. Note that chr is passed as a 16-bit value, even though only the lower byte is used in searching. In C, the corresponding function is strchr, which can be called as strchr(string1,int_char). As in our program, the character to be located is passed as an int, which will be converted to a char. Our return values are compatible to the values returned by the C function.

```
;-----------------------------------------------------------
;String locate a character procedure. Receives a character
;and a string pointer (seg:offset) via the stack.
;char should be passed as a 16-bit word.
;If string1 is not a string, CF is set;
;otherwise, locates the first occurrence of char in string1
;and returns a pointer to the located char in EAX (if the
;search is successful; otherwise EAX = NULL) with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
str_chr:
    enter   0,0
```

```
        push    ECX
        push    EDI
        push    ES
        ; find string length first
        les     EDI,STRING1  ; src string pointer
        push    ES
        push    EDI
        call    str_len
        jc      sh_no_string

        mov     ECX,EAX      ; src string length in ECX
        inc     ECX
        mov     AX,[EBP+16]  ; read char. into AL
        cld                  ; forward search
        repne   scasb
        dec     EDI          ; back up to match char.
        xor     EAX,EAX      ; assume no char. match (EAX=NULL)
        jcxz    sh_skip
        mov     EAX,EDI      ; return pointer to char.
    sh_skip:
        clc                  ; no error
        jmp     SHORT sh_done
    sh_no_string:
        stc                  ; carry set => no string
    sh_done:
        pop     ES
        pop     EDI
        pop     ECX
        leave
        ret     10           ; clear stack and return
```

Example 10.6 *String convert procedure to convert* `string2` *to* `string1` *in which all lowercase letters are converted to the corresponding uppercase letters.*

The main purpose of this example is to illustrate the use of `lodsb` and `stosb` instructions. We move the string length (plus one to include NULL) of `string2` into ECX, which will be used as the count register for the `loop` instruction. The `loop` body consists of

```
loop1:  lodsb
        if (lowercase letter)
        then convert to uppercase
        stosb
        loop    loop1
```

The string convert procedure is shown below:

```
;-----------------------------------------------------------
;String convert procedure. Receives two string pointers
;(seg:offset) via the stack - string1 and string2.
;If string2 is not a string, CF is set;
;otherwise, string2 is copied to string1 and lowercase
;letters are converted to corresponding uppercase letters.
;string2 is not modified in any way.
;It returns a pointer to string1 in EAX with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
str_cnv:
    enter   0,0
    push    ECX
    push    EDI
    push    ESI
    push    DS
    push    ES
    ; find string length first
    lds     ESI,STRING2  ; src string pointer
    push    DS
    push    ESI
    call    str_len
    jc      sn_no_string

    mov     ECX,EAX      ; src string length in ECX
    inc     ECX          ; add 1 to include NULL
    les     EDI,STRING1  ; dest string pointer
    cld                  ; forward search
loop1:
    lodsb
    cmp     AL,’a’       ; lowercase letter?
    jb      sn_skip
    cmp     AL,’z’
    ja      sn_skip      ; if no, skip conversion
    sub     AL,20H       ; if yes, convert to uppercase
sn_skip:
    stosb
    loop    loop1
    rep     movsb
    mov     EAX,[EBP+8]  ; return dest string pointer
    clc                  ; no error
    jmp     SHORT sn_done
sn_no_string:
    stc                  ; carry set => no string
```

```
sn_done:
    pop     ES
    pop     DS
    pop     ESI
    pop     EDI
    pop     ECX
    leave
    ret     16            ; clear stack and return
```

10.4 Testing String Procedures

Now let us turn our attention to testing the string procedures developed in the last section. A partial listing of this program is given in Program 10.1. The full program can be found in the str_test.asm file.

Our main interest in this section is to show how using an indirect procedure call would substantially simplify calling the appropriate procedure according to the user request. Let us first look at the indirect call instruction for 32-bit segments.

Indirect Procedure Call

In our discussions so far, we have been using only the direct near procedure calls, where the offset of the target procedure is provided directly. Recall that, even though we write only the procedure name, the assembler will generate the appropriate offset value at assembly time.

In indirect near procedure calls, this offset is given with one level of indirection. That is, the call instruction contains either a memory word address (through a label) or a 32-bit general-purpose register. The actual offset of the target procedure is obtained from the memory word or the register referenced in the call instruction. For example, we could use

```
call    EBX
```

if EBX contains the offset of the target procedure. As part of executing this call instruction, the contents of the EBX register are used to load EIP to transfer control to the target procedure. Similarly, we can use

```
call    [target_proc_ptr]
```

if the memory at target_proc_ptr contains the offset of the target procedure. As we have seen in Chapter 8, the jmp is another instruction that can be used for indirect jumps in exactly the same way as the indirect call.

Back to the Example

We maintain a procedure pointer table proc_ptr_table to facilitate calling the appropriate procedure. The user query response is used as an index into this table to get the target procedure offset. The EBX register is used as the index into this table. The instruction

```
        call    [proc_ptr_table+EBX]
```

causes the indirect procedure call. The rest of the program is straightforward to follow.

Program 10.1 Part of string test program str_test.asm

```
                . . .
.DATA
proc_ptr_table  dd  str_len_fun,str_cpy_fun,str_cat_fun
                dd  str_cmp_fun,str_chr_fun,str_cnv_fun
MAX_FUNCTIONS   EQU ($ - proc_ptr_table)/4

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  'Invalid response terminates program.',CR,LF
                db  'Please enter your choice: ',0
            . . .
.UDATA
string1         resb  STR_MAX
string2         resb  STR_MAX

.CODE
            . . .
        .STARTUP
        mov     AX,DS
        mov     ES,AX
query_choice:
        xor     EBX,EBX
        PutStr  choice_prompt    ; display menu
        GetCh   BL               ; read response
        sub     BL,'1'
        cmp     BL,0
        jb      invalid_response
        cmp     BL,MAX_FUNCTIONS
        jb      response_ok
invalid_response:
        PutStr  invalid_choice
        nwln
        jmp     SHORT done
```

```
response_ok:
        shl     EBX,2                ; multiply EBX by 4
        call    [proc_ptr_table+EBX]; indirect call
        jmp     query_choice
done:
        .EXIT
            . . .
```

10.5 Performance: Advantage of String Instructions

A question that naturally arises is: How beneficial are these string instructions? We answer this question by looking at the movs instruction in performing memory-to-memory data transfer.

We should note here that even though these instructions are called string instructions, these are not restricted to strings alone. This group of instructions can work on any data—not just on bytes. To reinforce this notion and to study the performance advantages of the string instructions, we use the movsd instruction, which copies 32-bit data from one memory buffer to another.

There are two chief advantages in using the string instructions:

1. The index registers are automatically updated (either incremented or decremented depending on the direction flag);
2. They are capable of operating on two operands that are located in the memory. Recall that nonstring instructions do not allow memory-to-memory data transfer.

For example, copying of data from array1 to array2 can be done by

```
cld
rep     movsd
```

provided DS:ESI and ES:EDI point to the source and destination arrays and ECX keeps the size of the arrays. Such memory-to-memory transfer is not possible with the mov instruction, which requires an intermediate register to achieve the same, as indicated below:

```
repeat:
      mov     EAX,[DS:ESI]
      mov     [ES:EDI],EAX
      add     ESI,4
      add     EDI,4
      dec     ECX
      jnz     repeat
```

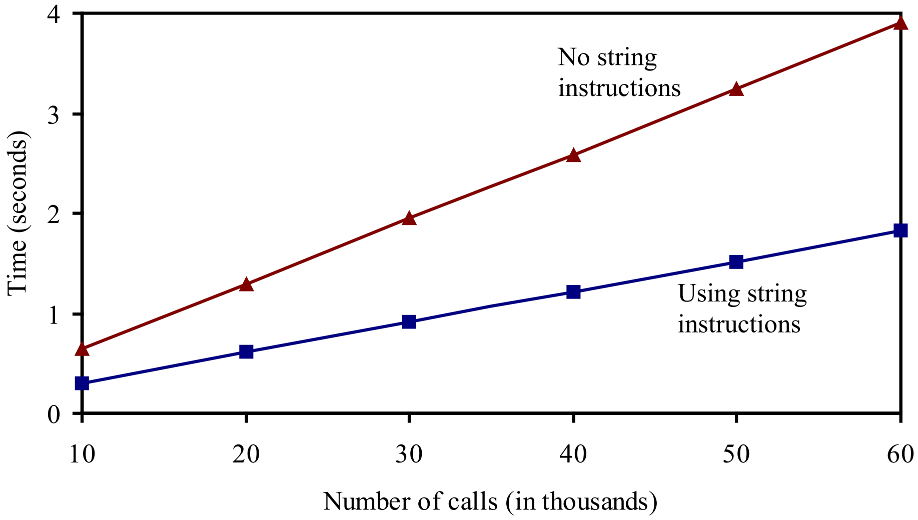


Figure 10.1 Performance advantage of string instructions for memory-to-memory data transfer.

Figure 10.1 shows the performance of these two versions on a 2.4-GHZ Pentium 4 system. The x-axis gives the number of times each procedure is called, and the y-axis gives the corresponding execution time in seconds. For this experiment, we fixed the array size as 50,000 elements. The version that uses the string instruction performs more than twice as fast as the nonstring version! The point to take away from this discussion is that the string instructions provide not only a simple and elegant solution but also a very efficient one.

10.6 Summary

We started this chapter with a brief discussion of various string representation schemes. Strings can be represented as either fixed-length or variable-length. Each representation has advantages and disadvantages. Variable-length strings can be stored either by explicitly storing the string length or by using a sentinel character to terminate the string. High-level programming languages like C use NULL-terminated storage representation for strings. We have also used the same representation to store strings.

There are five basic string instructions—`movs`, `lods`, `stos`, `cmps`, and `scas`. Each of these instructions can work on byte, word, or doubleword operands. These instructions do not require the specification of any operands. Instead, the required operands are assumed to be at DS:ESI and/or ES:EDI for 32-bit segments. For 16-bit segments, SI and DI registers are used instead of ESI and EDI registers, respectively. In addition, the direction flag is used to control the direction of string processing (forward or backward). Efficient code can be generated by combining string instructions with repeat prefixes. Three repeat prefixes—`rep`, `repe/repz`, and `repne/repnz`—are provided.

We also demonstrated, by means of an example, how indirect procedure calls can be used. Indirect procedure calls give us a powerful mechanism by which, for example, we can pass a procedure to be executed as a parameter using the standard parameter passing mechanisms.

The results presented in the last section indicate that using the string instructions results in significant performance advantages for memory-to-memory data transfers. We conclude from this discussion that string instructions are useful for memory-to-memory data copy operations.

10.7 Exercises

10–1 What are the advantages and disadvantages of the fixed-length string representation?

10–2 What are the advantages and disadvantages of the variable-length string representation?

10–3 Discuss the pros and cons of storing the string length explicitly versus using a sentinel character for storing variable-length strings.

10–4 We can write procedures to perform string operations without using the string instructions. What is the advantage of using the string instructions? Explain why?

10–5 Why doesn't it make sense to use the `rep` prefix with the `lods` instruction?

10–6 Explain why it does not make sense to use conditional repeat prefixes with the `lods`, `stos`, or `movs` string instructions.

10–7 Both `loop` and repeat prefixes use the ECX register to indicate the repetition count. Yet there is one significant difference between them in how they use the ECX register value. What is this difference?

10–8 Identify a situation in which the direction of string processing is important.

10–9 Identify a situation in which a particular direction of string processing is mandatory.

10–10 Suppose that the `lds` instruction is not supported by 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.

10–11 What is the difference between the direct procedure call and the indirect procedure call?

10–12 Explain how you can use the indirect procedure call to pass a procedure to be executed as a parameter.

10–13 Figure 10.1 shows that the string version performs better. Explain intuitively why this is so.

10–14 Discuss the advantages and disadvantages of the following two ways of declaring a message. The first version

```
msg1      db      'Test message'
msg1Len   dw      $-msg1
```

uses the `$` to compute the length, while the second version

```
msg1      db      'Test message'
msg1Len   dw      12
```

uses a constant.

10.8 Programming Exercises

10–P1 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` via the stack. Of course, the procedure receives only the string pointers, not the actual strings. It should copy at most the first `num` characters of `string2` to `string1`.

10–P2 Write a procedure `str_ncmp` to mimic the C function `strncmp`. The parameters passed to this function are the same as those of `str_ncpy`. It should compare at most the first `num` characters of the two strings and return a positive, negative, or 0 value like the `str_cmp` procedure does.

10–P3 A *palindrome* is a word, verse, sentence, or number that reads the same backward or forward. Blanks, punctuation marks, and capitalization do not count in determining palindromes. Here are some examples:

> 1991
> Able was I ere I saw Elba
> Madam! I'm Adam

Write a procedure to determine if a given string is a palindrome. The string is passed via the stack (i.e., the string pointer is passed to the procedure). The procedure returns 1 in EAX if the string is a palindrome; otherwise, it returns 0. The carry flag is used to indicate the *Not a string* error message, as we did in our examples in this chapter.

10–P4 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 passed is (⊔ indicates a blank character)

> ⊔⊔⊔⊔⊔Read⊔⊔my⊔lips.

it will be modified by removing all leading blanks as

> Read⊔⊔my⊔lips.

10–P5 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 passed is (⊔ indicates a blank character)

> ⊔⊔⊔⊔⊔Read⊔⊔⊔my⊔⊔⊔⊔⊔lips.

it will be modified by removing all leading and duplicate blanks as

> Read⊔my⊔lips.

10–P6 Write a procedure `str_str` that receives two pointers to strings `string` and `substring` via the stack and searches for `substring` in `string`. If a match is found, it returns in EAX the starting position of the first match. Matching should be case-sensitive. A negative value is returned in EAX if no match is found. For example, if `string` = Good things come in small packages.

and

`substring` = in

the procedure should return 8 in EAX, indicating a match of `in` in `things`.

10–P7 Write a procedure to read a string representing a person's name from the user 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–P8 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 capital letters.

10–P9 Write a procedure to match two strings that are received via the stack. The match should be case-insensitive, i.e., uppercase and lowercase letters are considered a match. For example, `Veda Anita` and `VeDa ANIta` are considered matching strings.

10–P10 Write a procedure to reverse the words in a string. It receives the string via the stack and modifies the string by reversing the words. Here is an example:

input string: `Politics in Science`
modified string: `Science in Politics`

10–P11 Write a main program using indirect procedure calls to test the procedures written in the previous exercises. You can simplify your job by modifying the `str_test.asm` program appropriately.

Chapter 11

ASCII and BCD Arithmetic

Objectives

- To introduce ASCII and BCD number representations
- To explain arithmetic operations in ASCII and BCD representations
- To describe the instructions that support arithmetic in ASCII and BCD representations
- To discuss the tradeoffs among the binary, ASCII, and BCD representations

In the previous chapters, we used binary representation and discussed several instructions that operate on binary data. When we enter numbers from the keyboard, they are entered as an ASCII string of digit characters. Therefore, a procedure like `GetInt` *is needed to convert the input ASCII string into the equivalent binary number. Similarly, output should be converted from binary to ASCII. This conversion overhead cannot be ignored for some applications.*

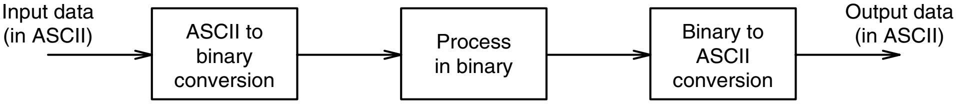
In this chapter, we present two alternative representations—ASCII and BCD—that avoid or reduce the conversion overhead. Section 11.1 provides a brief introduction to these two representations. The next two sections discuss how arithmetic operations can be done in these two representations.

While the ASCII and BCD representations avoid/reduce the conversion overhead, processing numbers in these two representations is slower than in the binary representation. This inherent tradeoff between conversion overhead and processing overhead among the three representations is explored in Section 11.4. The chapter ends with a summary.

11.1 ASCII and BCD Representations of Numbers

In previous chapters, the numeric data has been represented in the binary system. We discussed several arithmetic instructions that operate on such data. The binary representation is used internally for manipulation (e.g., arithmetic and logical operations).

When numbers are entered from the keyboard or displayed, they are in the ASCII form. Thus, it is necessary to convert numbers from ASCII to binary at the input end; we have to convert from binary to ASCII to output results as shown below:



We used `GetInt/GetLint` and `PutInt/PutLint` to perform these two conversions, respectively. These conversions represent an overhead, but we can process numbers much more efficiently in the binary form.

In some applications where processing of numbers is quite simple (for example, a single addition), the overhead associated with the two conversions might not be justified. In this case, it is probably more efficient to process numbers in the decimal form.

Another reason for processing numbers in decimal form is that we can use as many digits as necessary, and we can control rounding-off errors. This is important when representing dollars and cents for financial records.

Decimal numbers can be represented in one of two forms: ASCII or binary-coded-decimal (BCD). These two representations are discussed next.

11.1.1 ASCII Representation

In this representation, numbers are stored as strings of ASCII characters. For example, 1234 is represented as

31 32 33 34H

where 31H is the ASCII code for 1, 32H for 2, etc. As you can see, arithmetic on decimal numbers represented in ASCII form requires special care. There are two instructions to handle these numbers:

`aaa` — ASCII adjust after addition
`aas` — ASCII adjust after subtraction

We discuss these two instructions in Section 11.2.

11.1.2 BCD Representation

There are two types of BCD representation: unpacked BCD and packed BCD. In unpacked BCD representation, each digit is stored in a byte, while two digits are packed into a byte in the packed representation.

Unpacked BCD

This representation is similar to the ASCII representation except that each byte stores the binary equivalent of a decimal digit. Note that the ASCII codes for digits 0 through 9 are 30H through 39H. Thus, if we mask off the upper four bits, we get the unpacked BCD representation. For example, 1234 is stored in this representation as

01 02 03 04H

We deal with only positive numbers in this chapter. Thus, there is no need to represent the sign. But if a sign representation is needed, an additional byte can be used for the sign. The number is positive if this byte is 00H and negative if 80H.

There are two instructions to handle these numbers:

`aam` — ASCII adjust after multiplication
`aad` — ASCII adjust before division

Since this representation is similar to the ASCII representation, the four instructions—`aaa`, `aas`, `aam`, and `aad`—can be used with ASCII as well as unpacked BCD representations.

Packed BCD

In the last two representations, each digit of a decimal number is stored in a byte. The upper four bits of each byte contain redundant information. In packed BCD representation, each digit is stored using only four bits. Thus, two decimal digits can be packed into a byte. This reduces the memory requirement by half compared to the other two representations. For example, the decimal number 1234 is stored in packed BCD as

12 34H

which requires only two bytes as opposed to four in the other two representations. There are only two instructions that support addition and subtraction of packed BCD numbers:

`daa` — decimal adjust after addition
`das` — decimal adjust after subtraction

There is no support for multiplication or division operations. These two instructions are discussed in Section 11.3.

11.2 Processing in ASCII Representation

As mentioned before, four instructions are available to process numbers in the ASCII representation:

`aaa` — ASCII adjust after addition
`aas` — ASCII adjust after subtraction
`aam` — ASCII adjust after multiplication
`aad` — ASCII adjust before division

These instructions do not take any operands. They assume that the required operand is in the AL register.

11.2.1 ASCII Addition

To understand the need for the `aaa` instruction, look at the next two examples.

Example 11.1 *An ASCII addition example.*
Consider adding two ASCII numbers 4 (34H) and 5 (35H).

```
34H = 00110100B
35H = 00110101B
---   ---------
69H = 01101001B
```

The sum 69H is not correct. The correct value should be 09H in unpacked BCD representation. In this example, we get the right answer by setting the upper four bits to 0. This scheme, however, does not work in cases where the result digit is greater than 9, as shown in the next example. □

Example 11.2 *Another ASCII addition example.*
In this example, consider the addition of two ASCII numbers, 6 (36H) and 7 (37H).

```
36H = 00110110B
37H = 00110111B
---   ---------
6DH = 01101101B
```

Again, the sum 6DH is incorrect. We would expect the sum to be 13 (01 03H). In this case, ignore 6 as in the last example. But we have to add 6 to D to get 13. We add 6 because that is the difference between the bases of hex and decimal numbers. □

The `aaa` instruction performs these adjustments. This instruction is used after performing an addition operation by using either an `add` or `adc` instruction. The resulting sum in AL is adjusted to unpacked BCD representation. The `aaa` instruction works as follows.

1. If the least significant four bits of AL are greater than 9 or if the auxiliary flag is set, it adds 6 to AL and 1 to AH. Both CF and AF are set.
2. In all cases, the most significant four bits of AL are cleared (i.e., zeroed).

Here is an example that illustrates the use of the `aaa` instruction.

Example 11.3 *A typical use of the* `aaa` *instruction.*

```
sub     AH,AH     ; clear AH
mov     AL,'6'    ; AL = 36H
add     AL,'7'    ; AL = 36H+37H = 6DH
aaa               ; AX = 0103H
or      AL,30H    ; AL = 33H
```

To convert the result in AL to an ASCII result, we have to insert 3 into the upper four bits of AL. □

To add multidigit decimal numbers, we have to use a loop that adds one digit at a time starting from the rightmost digit. Program 11.1 shows how the addition of two 10-digit decimal numbers is done in ASCII representation.

11.2.2 ASCII Subtraction

The `aas` instruction is used to adjust the result of a subtraction operation (`sub` or `sbb`) and works like `aaa`. The actions taken by `aas` are

1. If the least significant four bits of AL are greater than 9 or if the auxiliary flag is set, it subtracts 6 from AL and 1 from AH. Both CF and AF are set.
2. In all cases, the most significant four bits of AL are cleared (i.e., zeroed).

It is straightforward to see that the adjustment is needed only when the result is negative, as shown in the following examples.

Example 11.4 *ASCII subtraction (positive result).*

```
sub     AH,AH     ; clear AH
mov     AL,'9'    ; AL = 39H
sub     AL,'3'    ; AL = 39H-33H = 6H
aas               ; AX = 0006H
or      AL,30H    ; AL = 36H
```

Notice that `aas` does not change the contents of the AL register, as the result is a positive number. □

Example 11.5 *ASCII subtraction (negative result).*

```
sub     AH,AH     ; clear AH
mov     AL,'3'    ; AL = 33H
sub     AL,'9'    ; AL = 33H-39H = FAH
aas               ; AX = FF04H
or      AL,30H    ; AL = 34H
```

The AL result indicates the magnitude; the `aas` instruction sets the carry flag to indicate that a borrow has been generated. □

Is the last result FF04H generated by aas useful? It is when you consider multidigit subtraction. For example, if we are subtracting 29 from 53 (i.e., 53−29), the first loop iteration performs 3−9 as in the last example. This gives us the result 4 in AL and the carry flag is set. Next we perform 5−2 using sbb to include the borrow generated by the previous subtraction. This leaves 2 as the result. After ORing with 30H, we will have 32 34H, which is the correct answer (24).

11.2.3 ASCII Multiplication

The aam instruction is used to adjust the result of a mul instruction. Unlike addition and subtraction, multiplication should not be performed on ASCII numbers but on unpacked BCD numbers. The aam works as follows: AL is divided by 10 and the quotient is stored in AH and the remainder in AL.

Example 11.6 *ASCII multiplication.*

```
mov    AL,3     ; multiplier in unpacked BCD form
mov    BL,9     ; multiplicand in unpacked BCD form
mul    BL       ; result 001BH is in AX
aam             ; AX = 0207H
or     AX,3030H ; AX = 3237H
```

Notice that the multiplication should be done using unpacked BCD numbers—not on ASCII numbers! If the digits in AL and BL are in ASCII as in the following code, we have to mask off the upper four bits.

```
mov    AL,'3'   ; multiplier in ASCII
mov    BL,'9'   ; multiplicand in ASCII
and    AL,0FH   ; multiplier in unpacked BCD form
and    BL,0FH   ; multiplicand in unpacked BCD form
mul    BL       ; result 001BH is in AX
aam             ; AX = 0207H
or     AL,30H   ; AL = 37H
```

The aam works only with the mul instruction but not with the imul instruction. □

11.2.4 ASCII Division

The aad instruction adjusts the numerator in AX *before* dividing two unpacked decimal numbers. The denominator has to be a single byte unpacked decimal number. The aad instruction multiplies AH by 10 and adds it to AL and sets AH to zero. For example, if AX = 0207H before aad, AX changes to 001BH after executing aad. As you can see from the last example, aad reverses the operation of aam.

Example 11.7 *ASCII division.*
Consider dividing 27 by 5.

```
mov    AX,0207H ; dividend in unpacked BCD form
mov    BL,05H   ; divisor in unpacked BCD form
aad             ; AX = 001BH
div    BL       ; AX = 0205H
```

The aad instruction converts the unpacked BCD number in AX to binary form so that div can be used. The div instruction leaves the quotient in AL (05H) and the remainder in AH (02H). □

11.2.5 Example: Multidigit ASCII Addition

Addition of multidigit numbers in ASCII representation is done one digit at a time starting with the rightmost digit. To illustrate the process involved, we discuss how addition of two 10-digit numbers is done (see the program listing below).

Program 11.1 ASCII addition of two 10-digit numbers

```
;Addition of two integers in ASCII form   ASCIIADD.ASM
;
;        Objective: To demonstrate addition of two integers
;                   in the ASCII representation.
;            Input: None.
;           Output: Displays the sum.
%include "io.mac"

.DATA
sum_msg   db  'The sum is: ',0
number1   db  '1234567890'
number2   db  '1098765432'
sum       db  '          ',0 ; add NULL char. to use PutStr

.CODE
      .STARTUP
      ; ESI is used as index into number1, number2, and sum
      mov     ESI,9          ; ESI points to rightmost digit
      mov     ECX,10         ; iteration count (# of digits)
      clc                    ; clear carry (we use ADC not ADD)
add_loop:
      mov     AL,[number1+ESI]
      adc     AL,[number2+ESI]
      aaa                    ; ASCII adjust
      pushf                  ; save flags because OR
```

```
26:        or      AL,30H          ;  changes CF that we need
27:        popf                    ;  in the next iteration
28:        mov     [sum+ESI],AL    ; store the sum byte
29:        dec     ESI             ; update ESI
30:        loop    add_loop
31:        PutStr  sum_msg         ; display sum
32:        PutStr  sum
33:        nwln
34:        .EXIT
```

The program adds two numbers `number1` and `number2` and displays the sum. We use ESI as an index into the input numbers, which are in the ASCII representation. The ESI register is initialized to point to the rightmost digit (line 18). The loop count 10 is set up in ECX (line 19). The addition loop (lines 21–30) adds one digit by taking any carry generated by the previous iteration into account. This is done by using the `adc` rather than the `add` instruction. Since the `adc` instruction is used, we have to make sure that the carry is clear initially. This is done on line 20 using the `clc` (clear carry) instruction.

Note that the `aaa` instruction produces the result in unpacked BCD form. To convert to the ASCII form, we have to `or` the result with 30H (line 26). This ORing, however, destroys the carry generated by the `adc` instruction that we need in the next iteration. Therefore, it is necessary to save (line 25) and restore (line 27) the flags.

The overhead in performing the addition is obvious. If the input numbers were in binary, only a single `add` instruction would have performed the required addition. This conversion-overhead versus processing-overhead tradeoff is discussed in Section 11.4.

11.3 Processing Packed BCD Numbers

In this representation, as indicated earlier, two decimal numbers are packed into a byte. There are two instructions to process packed BCD numbers:

`daa` — Decimal adjust after addition
`das` — Decimal adjust after subtraction

There is no support for multiplication or division. For these operations, we will have to unpack the numbers, perform the operation, and repack them.

11.3.1 Packed BCD Addition

The `daa` instruction can be used to adjust the result of an addition operation to conform to the packed BCD representation. To understand the sort of adjustments required, let us look at some examples next.

Example 11.8 *A packed BCD addition example.*
Consider adding two packed BCD numbers 29 and 69.

$$\begin{array}{l@{\;=\;}l} \texttt{29H} & \texttt{00101001B} \\ \underline{\texttt{69H}} & \underline{\texttt{01101001B}} \\ \texttt{92H} & \texttt{10010010B} \end{array}$$

The sum 92 is not the correct value. The result should be 98. We get the correct answer by adding 6 to 92. We add 6 because the carry generated from bit 3 (i.e., auxiliary carry) represents an overflow above 16, not 10, as is required in BCD. □

Example 11.9 *Another packed BCD addition example.*
Consider adding two packed BCD numbers 27 and 34.

$$\begin{array}{l@{\;=\;}l} \texttt{27H} & \texttt{00100111B} \\ \underline{\texttt{34H}} & \underline{\texttt{00110100B}} \\ \texttt{5BH} & \texttt{01011011B} \end{array}$$

Again, the result is incorrect. The sum should be 61. The result 5B requires correction, as the first digit is greater than 9. To correct the result add 6, which gives us 61. □

Example 11.10 *A final packed BCD addition example.*
Consider adding two packed BCD numbers 52 and 61.

$$\begin{array}{l@{\;=\;}l} \texttt{52H} & \texttt{01010010B} \\ \underline{\texttt{61H}} & \underline{\texttt{01100001B}} \\ \texttt{B3H} & \texttt{10110011B} \end{array}$$

This result also requires correction. The first digit is correct, but the second digit requires a correction. The solution is the same as that used in the last example—add 6 to the second digit (i.e., add 60H to the result). This gives us 13 as the result with a carry (effectively equal to 113). □

The `daa` instruction exactly performs adjustments like these to the result of `add` or `adc` instructions. More specifically, the following actions are taken by `daa`:

- If the least significant four bits of AL are greater than 9 or if the auxiliary flag is set, it adds 6 to AL and sets AF;
- If the most significant four bits of AL are greater than 9 or if the carry flag is set, it adds 60H to AL and sets CF.

Example 11.11 *Code for packed BCD addition.*
Consider adding two packed BCD numbers 71 and 43.

```
mov    AL,71H
add    AL,43H     ; AL = B4H
daa               ; AL = 14H and CF = 1
```

As indicated, the daa instruction restores the result in AL to the packed BCD representation. The result including the carry (i.e., 114H) is the correct answer in packed BCD. □

As in the ASCII addition, multibyte BCD addition requires a loop. After discussing the packed BCD subtraction, we present an example to add two 10-byte packed BCD numbers.

11.3.2 Packed BCD Subtraction

The das instruction can be used to adjust the result of a subtraction (i.e., the result of sub or sbb). It works similar to daa and performs the following actions:

- If the least significant four bits of AL are greater than 9 or if the auxiliary flag is set, it subtracts 6 from AL and sets AF;
- If the most significant four bits of AL are greater than 9 or if the carry flag is set, it subtracts 60H from AL and sets CF.

Here is an example illustrating the use of the das instruction.

Example 11.12 *Code for packed BCD subtraction.*
Consider subtracting 43 from 71 (i.e., 71 − 43).

```
mov    AL,71H
sub    AL,43H     ; AL = 2EH
das               ; AL = 28H
```

The das instruction restores the result in AL to the packed BCD representation. □

11.3.3 Example: Multibyte Packed BCD Addition

As in the ASCII representation, when adding two multibyte packed BCD numbers, we have to use a loop that adds a pair of decimal digits in each iteration starting from the rightmost pair. An example program that adds two 10-byte packed BCD numbers, number1 and number2, is shown in Program 11.2.

Program 11.2 Packed BCD addition of two 10-digit numbers

```
;Addition of integers in packed BCD form   BCDADD.ASM
;
;        Objective: To demonstrate addition of two integers
;                   in the packed BCD representation.
;            Input: None.
;          Output: Displays the sum.

%define SUM_LENGTH    10

%include "io.mac"

.DATA
sum_msg   db  'The sum is: ',0
number1   db  12H,34H,56H,78H,90H
number2   db  10H,98H,76H,54H,32H
ASCIIsum  db  '          ',0    ; add NULL char.

.UDATA
BCDsum    resb   5

.CODE
      .STARTUP
      mov     ESI,4
      mov     ECX,5             ; loop iteration count
      clc                       ; clear carry (we use ADC)
add_loop:
      mov     AL,[number1+ESI]
      adc     AL,[number2+ESI]
      daa                       ; ASCII adjust
      mov     [BCDsum+ESI],AL   ; store the sum byte
      dec     ESI               ; update index
      loop    add_loop
      call    ASCII_convert
      PutStr  sum_msg           ; display sum
      PutStr  ASCIIsum
nwln
      .EXIT

;-----------------------------------------------------------
; Converts the packed decimal number (5 digits) in BCDsum
; to ASCII represenation and stores it in ASCIIsum.
; All registers are preserved.
;-----------------------------------------------------------
ASCII_convert:
      pushad                    ; save registers
      ; ESI is used as index into ASCIIsum
      mov     ESI,SUM_LENGTH-1
      ; EDI is used as index into BCDsum
      mov     EDI,4
```

```
50:        mov     ECX,5              ; loop count (# of BCD digits)
51:  cnv_loop:
52:        mov     AL,[BCDsum+EDI]    ; AL = BCD digit
53:        mov     AH,AL              ; save the BCD digit
54:        ; convert right digit to ASCII & store in ASCIIsum
55:        and     AL,0FH
56:        or      AL,30H
57:        mov     [ASCIIsum+ESI],AL
58:        dec     ESI
59:        mov     AL,AH              ; restore the BCD digit
60:        ; convert left digit to ASCII & store in ASCIIsum
61:        shr     AL,4               ; right-shift by 4 positions
62:        or      AL,30H
63:        mov     [ASCIIsum+ESI],AL
64:        dec     ESI
65:        dec     EDI                ; update EDI
66:        loop    cnv_loop
67:        popad                      ; restore registers
68:        ret
```

The two numbers to be added are initialized on lines 14 and 15. The space for the sum (BCDsum) is reserved using resb on line 19.

The code is similar to that given in Program 11.1. However, since we add two decimal digits during each loop iteration, only five iterations are needed to add the 10-digit numbers. Therefore, processing numbers in packed BCD representation is faster than in ASCII representation. In any case, both representations are considerably slower in processing numbers than the binary representation.

At the end of the loop, the sum is stored in BCDsum as a packed BCD number. To display this number, we have to convert it to the ASCII form (an overhead that is not present in the ASCII version).

The procedure ASCII_convert takes BCDsum and converts it to equivalent ASCII string and stores it in ASCIIsum. For each byte read from BCDsum, two ASCII digits are generated. Note that the conversion from packed BCD to ASCII can be done by using only logical and shift operations. On the other hand, conversion from binary to ASCII requires a more expensive divide operation (thus increasing the conversion overhead).

11.4 Performance: Decimal Versus Binary Arithmetic

Now you know three representations to perform arithmetic operations: binary, ASCII, and BCD. The majority of operations are done in binary. However, there are tradeoffs associated with these three representations.

Table 11.1 Tradeoffs Associated with the Three Representations

Representation	Storage overhead	Conversion overhead	Processing overhead
Binary	Nil	High	Nil
Packed BCD	Medium	Medium	Medium
ASCII	High	Nil	High

First we will look at the storage overhead. The binary representation is compact and the most efficient one. The ASCII and unpacked BCD representations incur high overhead as each decimal digit is stored in a byte (see Table 11.1). The packed BCD representation, which stores two decimal digits per byte, reduces this overhead by approximately half. For example, using two bytes, we can represent numbers from 0 to 65,535 in binary representation and from 0 to 9999 in packed BCD representation, but only from 0 to 99 in ASCII and unpacked BCD representations.

In applications where the input data is in ASCII form and the output is required to be in ASCII, binary arithmetic may not always be the best choice. This is because there are overheads associated with the conversion between ASCII and binary representations. However, processing numbers in binary can be done much more efficiently than in either ASCII or BCD representations. Table 11.1 shows the tradeoffs associated with these three representations.

When the input and output use the ASCII form and there is little processing, processing numbers in ASCII is better. This is so because ASCII version does not incur any conversion overhead. On the other hand, due to high overhead in converting numbers between ASCII and binary, the binary version takes more time than the ASCII version. The BCD version also takes substantially more time than the ASCII version but performs better than the binary version mainly because conversions between BCD and ASCII are simpler.

When there is significant processing of numbers, the binary version tends to perform better than the ASCII and BCD versions. In this scenario, the ASCII version provides the worst performance as its processing overhead is high (see Table 11.1). The BCD version, while slower than the binary version, performs much better than the ASCII version.

The moral of the story is that a careful analysis of the application should be done before deciding on the choice of representation for arithmetic in some applications. This is particularly true for business applications, where the data might come in the ASCII form.

11.5 Summary

In previous chapters we converted decimal data into binary for storing internally as well as for manipulation. This chapter introduced two alternative representations for storing the decimal data—ASCII and BCD. The BCD representation can be either unpacked or packed.

In ASCII and unpacked BCD representations, one decimal digit is stored per byte, whereas the packed BCD representation stores two digits per byte. Thus, the storage overhead is substantial in ASCII and unpacked BCD. Packed BCD representation uses the storage space more efficiently (typically requiring half as much space). Binary representation, on the other hand, does not introduce any overhead.

There are two main overheads that affect the execution time of a program: conversion overhead and processing overhead. When the ASCII form is used for data input and output, the data should be converted between ASCII and binary/BCD. This conversion overhead for binary representation can be substantial, as multiplication and division are required. There is much less overhead for the BCD representations, as only logical and shift operations are needed.

On the other hand, number processing in binary is much faster than in ASCII or BCD representations. Packed BCD representation is better than ASCII representation, as each byte stores two decimal digits. We discussed these tradeoffs in the last section.

11.6 Exercises

11–1 Briefly give the reasons for using either ASCII or BCD representations.

11–2 How is a sign represented in ASCII and BCD representations?

11–3 What is the difference between packed and unpacked BCD representations? Discuss the tradeoffs between the two BCD representations.

11–4 How are the following numbers represented in (i) binary, (ii) ASCII, (iii) unpacked BCD, and (iv) packed BCD?

(a) 500
(b) 32,025
(c) 2491
(d) 4385

11–5 Explain why `pushf` and `popf` instructions are needed in Program 11.1.

11–6 For each code fragment given, find the contents of AX after executing the `aaa` instruction.

```
(a)                         (b)
     sub     AH,AH               sub     AH,AH
     mov     AL,’5’              mov     AL,’7’
     add     AL,’3’              add     AL,’8’
     aaa                         aaa
(c)                         (d)
     sub     AH,AH               sub     AH,AH
     mov     AL,’9’              mov     AL,’9’
     add     AL,’7’              add     AL,’9’
     aaa                         aaa
```

11–7 For each code fragment given, find the contents of AX after executing the `aas` instruction.

```
(a)                   (b)
    sub   AH,AH           sub   AH,AH
    mov   AL,’9’          mov   AL,’4’
    sub   AL,’4’          sub   AL,’9’
    aas                   aas
(c)                   (d)
    sub   AH,AH           sub   AH,AH
    mov   AL,’3’          mov   AL,’4’
    sub   AL,’7’          sub   AL,’5’
    aas                   aas
```

11–8 For each code fragment given, find the contents of AX after executing the `daa` instruction.

```
(a)                   (b)
    mov   AL,21H          mov   AL,37H
    add   AL,57H          add   AL,45H
    daa                   daa
(c)                   (d)
    mov   AL,21H          mov   AL,55H
    add   AL,96H          add   AL,66H
    daa                   daa
```

11–9 For each code fragment given, find the contents of AX after executing the `das` instruction.

```
(a)                   (b)
    mov   AL,66H          mov   AL,64H
    sub   AL,45H          sub   AL,37H
    das                   das
(c)                   (d)
    mov   AL,34H          mov   AL,45H
    sub   AL,51H          sub   AL,57H
    das                   das
```

11–10 For each code fragment given, find the contents of AX after executing the `aam` instruction.

```
(a)                   (b)
    mov   AL,’3’          mov   AL,’9’
    mov   BL,’2’          mov   BL,’9’
    mul   BL              mul   BL
    aam                   aam
(c)                   (d)
    mov   AL,’4’          mov   AL,’7’
    mov     BL,'4'              mov     BL,'3'
    mul     BL                  mul     BL
    aam                         aam
```

11–11 Discuss the conversion versus processing overhead tradeoffs associated with the binary, ASCII, and BCD (both packed and unpacked) representations.

11.7 Programming Exercises

11–P1 Assuming that an ASCII digit is in the AL register, write an assembly language code fragment to convert it to unpacked BCD representation.

11–P2 Assuming that the digit in the AL register is in unpacked BCD representation, write an assembly language code fragment to convert it to ASCII representation.

11–P3 Suppose that two ASCII digits are in AH and AL, with the least significant digit in AL. Write an assembly language code fragment to convert it to packed BCD representation and store it in AL.

11–P4 Modify `asciiadd.asm` (Program 11.1) to read two decimal numbers from the user instead of taking them from memory. The two numbers from the user should be read as strings. You can use `GetStr` to read the input numbers.

11–P5 Modify `asciiadd.asm` (Program 11.1) to read two decimal numbers from the user instead of taking them from memory (as in the last exercise). It should then subtract the second number from the first and display the result using `PutStr`. The two numbers from the user should be read as strings using `GetStr`.

11–P6 Modify `bcdadd.asm` (Program 11.2) to receive two decimal numbers from the user instead of taking them from memory. The two numbers from the user should be read as ASCII strings using `GetStr`. The input numbers should be converted to the packed BCD representation for performing the addition as in Program 11.2. The result should be converted back to ASCII so that it can be displayed by `PutStr`.

11–P7 Modify the program for the last exercise to perform subtraction.

11–P8 Write an assembly language program to perform multiplication in ASCII representation. In this exercise, assume that the multiplier is a single digit. The two numbers to be multiplied are given as input in the ASCII form (your program reads them using `GerStr`). The result should be displayed by using `PutStr`.

Hint: You need to use a loop that mimics the behavior of the longhand multiplication (i.e., multiply one digit at a time).

11–P9 Write an assembly language program to perform multiplication in ASCII representation. Unlike in the last exercise, both numbers can be multidigit (up to 5 digits) numbers. The two numbers to be multiplied are given as input in the ASCII form (your program reads them using `GerStr`). The result should be displayed by using `PutStr`.

Hint: You need two (nested) loops, where the inner loop is similar to that in the last exercise.

PART III

MIPS Assembly Language

This part focuses on the MIPS assembly language. It consists of two chapters: Chapters 12 and 13. The first chapter describes the RISC design principles. It also covers the MIPS processor details.

MIPS assembly language is presented in Chapter 13. This chapter also gives details on the SPIM simulator. All the programming examples given in this chapter can be run on a Pentium-based PC using the SPIM simulator. It helps if you are familiar with the material in Appendix D before attempting to run the programs given in this chapter. This appendix gives details on the SPIM simulator and on how you can assemble and debug MIPS assembly language programs.

Chapter 12

MIPS Processor

Objectives

- To discuss the motivation for RISC processors
- To present RISC design principles
- To give details on MIPS processors

We start the chapter with an introduction to RISC processors. Section 12.2 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. These characteristics include simple instructions and few addressing modes. RISC processors use a 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. Section 12.4 gives details on the MIPS processor. We conclude the chapter with a summary.

12.1 Introduction

One of the important abstractions that a programmer uses is the instruction set architecture (ISA). The ISA defines the personality of a processor and specifies how a processor functions: what instructions it executes, what interpretation is given to these instructions, and so on. The ISA, 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. For example, the Intel 32-bit processors like Celeron, Pentium 4, and Xeon are implementations of their 32-bit architecture known as the

IA-32 architecture. The Celeron is the cheaper version targeted for low-end PC market. The Xeon, on the other hand, is designed for high-performance workstations and multiprocessor server systems.

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.

The IA-32 architecture, 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 the instructions set architecture. We define in the next couple of sections what we mean by complex ISA. For now, it is sufficient to know that IA-32 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 supports closely 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 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. 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 Computers (RISC). Even though the main goal was not to reduce the number of instructions, but rather the complexity, the term has stuck. SPARC, PowerPC, MIPS, and Itanium are all examples of RISC designs.

There is no precise definition of what constitutes a RISC design. 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 CISC route in the first place. Since CISC and RISC have their advantages and disadvantages, some designs take features from both classes. For example PowerPC, which follows the RISC philosophy, has quite a few complex instructions.

We look at the MIPS architecture in this chapter. The next chapter describes its assembly language in detail.

12.2 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.

The introduction of microprogramming facilitated cost-effective implementation of complex instructions by using microcode. Microprogramming has not only aided in implementing complex instructions, it also provided some additional advantages. Since microprogrammed control units use small, faster 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 `loop` instruction of the IA-32 architecture 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 IA-32 string instructions, which auto-increment the index registers. Each string instruction typically requires two instructions on a RISC processor. As another example, consider 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 the following table:

	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

To illustrate how code density affects execution efficiency, consider the auto-increment 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, and 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, 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.

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. 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 slower 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 the 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 are 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—simple data types such as integers and characters to complex data structures such as records/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. The missing complex data types can be synthesized using the simple data types.

Simple Addressing Modes

CISC designs provide a large number of addressing modes. The main motivation is (1) to support complex data structures, and (2) to provide flexibility to access operands. For example, the IA-32 architecture provides “Based-Indexed addressing with Scale-factor” to access complex data structures like multidimensional arrays (see Chapter 6 for details). It also allows one of the source operands to be in memory or register. While 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, instruction length in IA-32 can range from 1 byte to 12 bytes. Variable instruction lengths lead to inefficient instruction decoding and scheduling.

Large Register Set

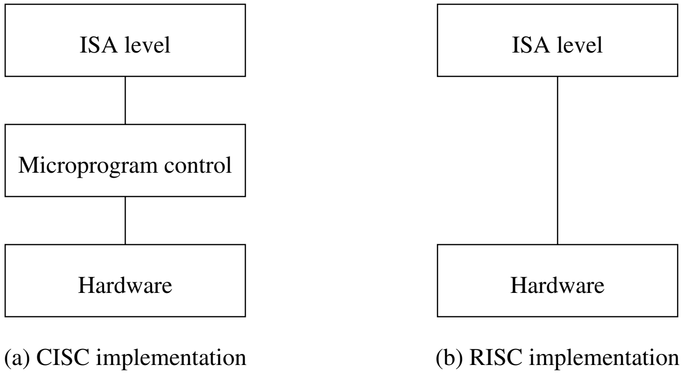
Several researchers have studied the characteristics of procedure calls in HLLs. We quote two studies—one by Patterson and the other by Tanenbaum—in this section. Several other studies, in fact, support the findings of these two studies.

Patterson’s study of C and Pascal programs found that procedure call/return constitutes about 12–15% of HLL statements. As a percentage of the total machine language instructions, call/return instructions are about 31–33%. More interesting is the fact that call/return generate 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 151). In Pentium, for example, 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 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 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 eight general-purpose registers in the Intel 32-bit processors are a limiting factor in providing such support. Itanium, which is the Intel’s 64-bit processor, provides a large register set (128 registers), and most procedure calls on Itanium can completely avoid accessing memory.

12.3 RISC Design Principles

The best way to understand RISC is to treat it as a concept to design processors. While initial RISC processors had fewer instructions compared to their CISC counterparts, the new generation of RISC processors have 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 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 execution takes only one clock cycle demands register-based operands, which in turn suggests that we need a large number of registers.



(a) CISC implementation

(b) RISC implementation

Figure 12.1 The instruction set architecture (ISA) is implemented directly in RISC processors whereas CISC processors implement through a microprogrammed control.

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 (see Figure 12.1). 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 capture these instructions, the overall execution efficiency can be as good as a microcoded CISC machine.

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 IA-32 architecture, for instance, allows register-to-register as well as register-to-memory operations; it does not allow memory-to-memory operations. It, however, supports memory-to-memory operations with the string instructions (see Chapter 10).

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 one-instruction-per-cycle rate. Restricting most instruction operands to registers also simplifies the control unit. RISC processors—SPARC, PowerPC, Itanium, and MIPS—use this load/store architecture.

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. For example, MIPS supports simple register-indirect addressing modes as described in the next section.

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.

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, the 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. The 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 instruction unit. Since RISC architectures use fixed-length instructions, the pipelines tend to have fewer stages than the comparable CISC processors. Since RISC processors use simple instructions, there will be more instructions in a program than 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 like fixed-size instructions and simple addressing modes.

12.4 MIPS 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 follows 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 shall see later, the R2000 provides several load and store instructions to transfer different sizes of data—byte, halfword, word, and doubleword.

Like most recent processors, R2000 supports both little-endian and big-endian formats. Recall that IA-32 processors use the little-endian format to store multibyte data. RISC processors typically have a large number of registers. For example, Itanium has 128 registers while PowerPC provides 32 registers. We start our discussion with the registers of R2000.

Registers

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 12.2.

Unlike in 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.

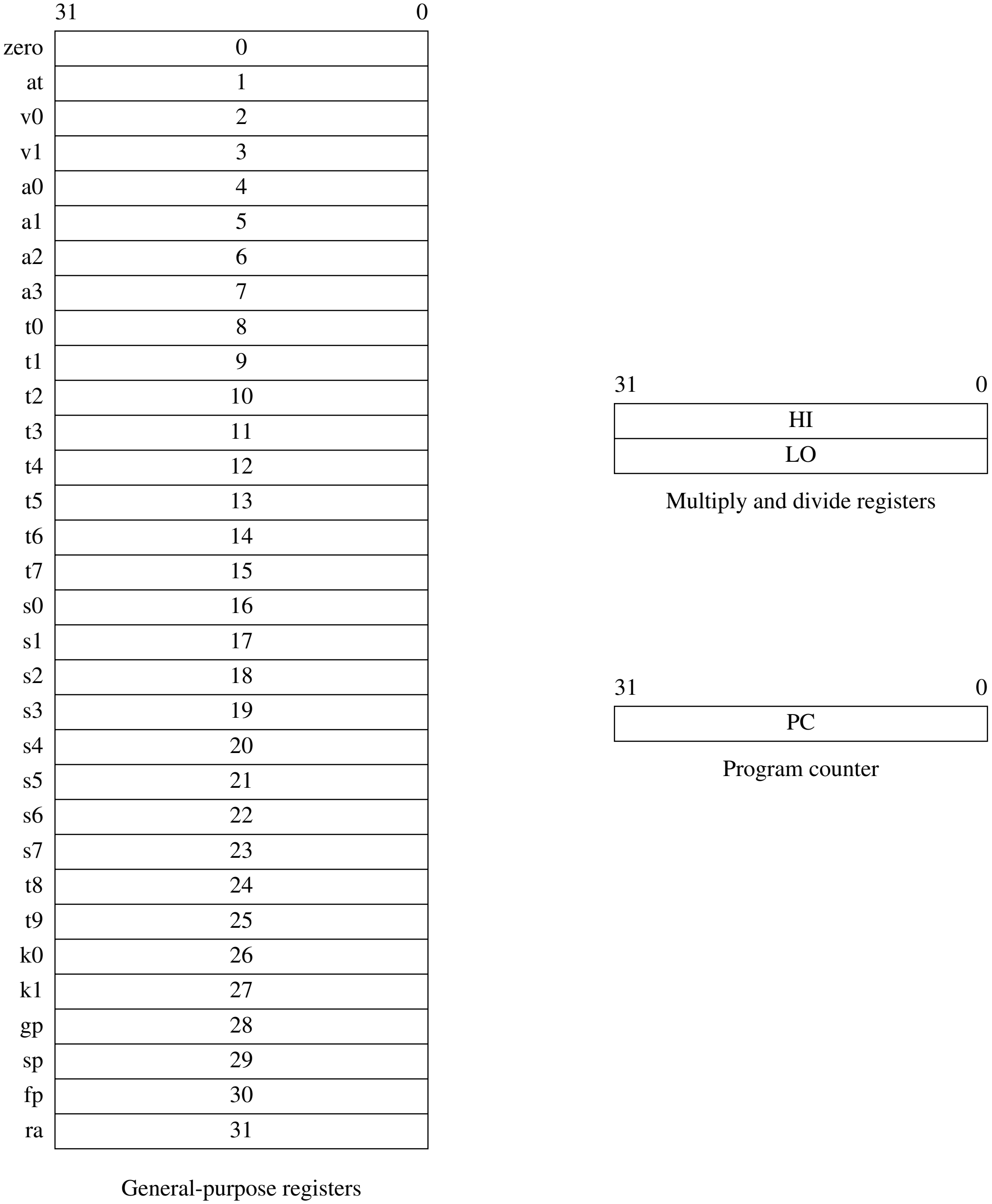
- Register `$0` is hardwired to the zero value. This register is often used as a source register when a zero value is needed. You can use this register as the destination register of an instruction if you want the result to be discarded.
- The last register `$31` is used as a link register by the jump and link (`jal`) instruction (discussed in Section 13.4 on page 386). This instruction is equivalent to the `call` instruction in the IA-32 architecture. Register `$31` is used to store the return address of a procedure call. We discuss this issue in detail in Section 13.4.

The PC register serves the same purpose as the instruction pointer (EIP) register in IA-32 processors. The two special-purpose registers—called HI and LO—are used to hold the results of integer multiply and divide instructions.

- In an integer multiply operation, HI and LO registers hold the 64-bit result, with the higher-order 32 bits in HI and the lower-order 32-bits in the LO register.
- In integer divide operations, the 32-bit quotient is stored in LO and the remainder in the HI register.

General-Purpose Register Usage Convention

Although there is no requirement from the processor hardware, MIPS has established a convention on how the general-purpose registers should be used. Table 12.1 shows the suggested



	31 … 0
zero	0
at	1
v0	2
v1	3
a0	4
a1	5
a2	6
a3	7
t0	8
t1	9
t2	10
t3	11
t4	12
t5	13
t6	14
t7	15
s0	16
s1	17
s2	18
s3	19
s4	20
s5	21
s6	22
s7	23
t8	24
t9	25
k0	26
k1	27
gp	28
sp	29
fp	30
ra	31

General-purpose registers

31 … 0
HI
LO

Multiply and divide registers

31 … 0
PC

Program counter

Figure 12.2 MIPS R2000 processor registers. All registers are 32 bits long.

Table 12.1 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)

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.

Registers `$v0` and `$v1` are used to return results from a procedure. Registers `$a0`–`$a3` are used to pass the first four arguments to procedures. The remaining arguments are passed via the stack.

Registers `$t0`–`$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 `$s0`–`$s7` are callee-saved registers that should be preserved across procedure calls.

Register `$sp` is the stack pointer and serves the same purpose as the `esp` register in the IA-32 architecture. It points to the last location in use on the stack. The MIPS compiler does not use a frame pointer. As a result, the frame pointer register `$fp` is used as a callee-saved register `$s8`. The `$ra` is used to store the return address in a procedure call. We will discuss these registers in Section 13.4.

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 pseudo-instructions. We will see some examples of this later (see page 367).

Addressing Modes

The IA-32 architecture provides several addressing modes, which is a characteristic of CISC designs. Since MIPS uses the load/store architecture, only the load and store instructions access memory. Thus, 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. The address is computed as

Effective address = Contents of base register `Rx` + `disp`.

Thus, compared to the IA-32 architecture, MIPS provides only based/indexed addressing mode. In 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. The table below shows the addressing modes supported by the virtual machine.

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`)

Note that most load and store instructions operate only on aligned data. 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 41.

Memory Usage

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 12.3. 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 toward lower memory address. This placement of segments allows sharing of unused memory by both data and stack segments.

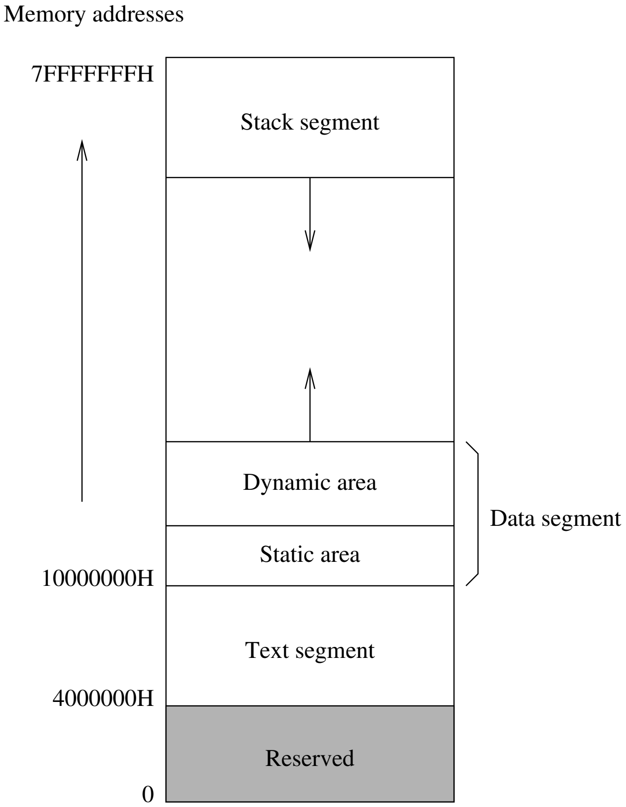


Figure 12.3 MIPS memory layout.

12.5 Summary

We have introduced important characteristics that differentiate RISC designs from their CISC counterparts. CISC designs provide complex instructions and a large number of addressing modes compared to RISC designs. 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 the processor design philosophy. RISC principles, based on empirical studies on CISC processors, have been proposed as an alternative to CISC designs. Most of the current processor designs are based on these RISC principles.

Like all RISC processors, MIPS 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. For example, MIPS 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 register serves the purpose of the instruction pointer. All these registers are 32 bits wide.

The MIPS architecture does not provide as many addressing modes as the IA-32 architecture does. This is one of the differences between CISC and RISC designs. In fact, MIPS supports only one addressing mode. However, the assembler augments this by a few other addressing modes. The MIPS assembly language is discussed in the next 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:

- Load/store architecture
- MIPS addressing modes
- MIPS architecture
- MIPS registers
- RISC characteristics
- RISC design principles

12.6 Exercises

12–1 We have seen that CISC designs typically use variable-length instructions. Discuss the reasons for this.

12–2 A typical CISC processor supports several addressing modes. For example, VAX 11/780 provides 22 addressing modes. What is the motivation for supporting a large number of addressing modes?

12–3 We have stated that RISC is a design philosophy. Discuss the main RISC characteristics.

12–4 Recent processors tend to follow the RISC design philosophy. For example, Intel moved from CISC to RISC designs for its 64-bit processors. Explain why.

12–5 The MIPS processor we discussed in this chapter uses the load/store architecture. Explain why RISC processors use the load/store architecture.

12–6 RISC processors provide a large number of registers compared to CISC processors. What is the rationale for this?

12–7 Describe the addressing modes supported by MIPS processor.

12–8 In the MIPS processor, the general-purpose register `$zero` is hardwired to zero. What is the reason for this?

12–9 What is the purpose of HI and LO registers?

12–10 What is the difference between registers `t0–t9` and `S0–S7`?

12–11 What is a typical use of registers `v0` and `v1`?

12–12 What is a typical use of registers `a0–a3`?

Chapter 13

MIPS Assembly Language

Objectives

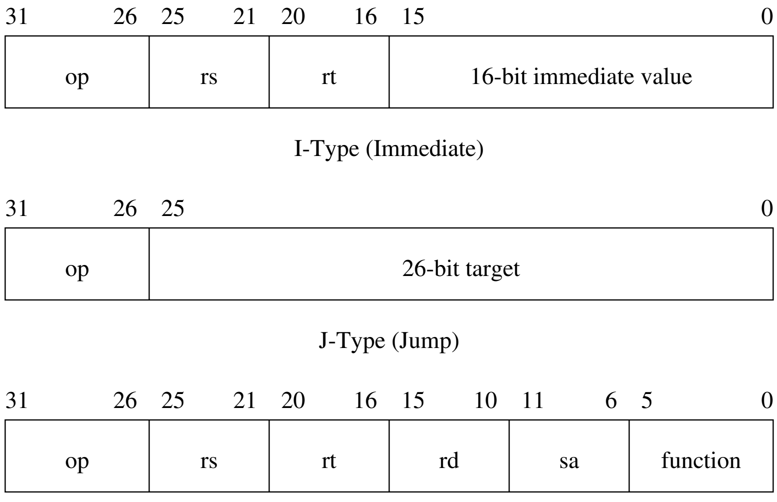
- To present MIPS instruction set details
- To describe SPIM simulator systems calls and directives
- To illustrate how MIPS assembly language programs are written
- To discuss MIPS stack implementation and procedures

We start this chapter with a description of the MIPS instruction set. SPIM provides several system calls to facilitate input/output from the assembly language programs. These calls are given in Section 13.2.1. Like the Pentium assemblers, SPIM also provides several directives, which are described in Section 13.2.2. Some example MIPS assembly programs are given in Section 13.3.

Simple procedures in MIPS do not have to use the stack. Section 13.4 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 13.5 with some examples. We conclude the chapter with a summary.

13.1 MIPS Instruction Set

MIPS instruction set consists of *instructions* and *pseudoinstructions*. 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.



I-Type (Immediate)

J-Type (Jump)

R-Type (Register)

Figure 13.1 Three MIPS instruction formats.

13.1.1 Instruction Format

MIPS, being a RISC processor, uses a fixed-length instruction format. Each instruction is 32 bits long as shown in Figure 13.1. It uses only three different instruction formats:

- *Immediate (I-type)*: All load and store instructions use this instruction format. 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 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.

The use of a limited number of instruction formats simplifies instruction decoding. However, three instruction formats and a single addressing mode means 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.

13.1.2 Data Transfer Instructions

MIPS provides load and store instructions to move data between memory and registers. Load instructions move data from memory into registers while the store instructions move 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, 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.

Other load instructions facilitate movement of larger data items. These instructions are summarized in Table 13.1.

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 `marks` array into the `$a0` register.

The `li` instruction shown in Table 13.1 is implemented as

```
ori      Rdest,$0,imm
```

The `ori` (OR immediate) instruction is discussed in Section 13.1.4.

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 13.2.

To move data between registers, we can use the `move` pseudoinstruction. The format is

```
move†    Rdest,Rsrc
```

It copies 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 366.

Table 13.1 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. The 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 half-word is treated as an unsigned number.
`lw    Rdest,address`	Load Word. Loads the word (four bytes) at `address` in memory into `Rdest`.
Assembler pseudoinstructions	
`la`† `Rdest,var`	Load Address. Loads the address of `var` into `Rdest`.
`li`† `Rdest,imm`	Load Immediate. Loads the immediate value `imm` into `Rdest`.

13.1.3 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 the `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, as overflow exception is generated; use `addiu` if overflow exception is not needed.

Table 13.2 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.

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. We can use addu† if overflow exception is not needed.

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; 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:
```

In the translation, 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 371.

Multiply Instructions

MIPS provides two multiply instructions—one for signed numbers (mult) and the other for unsigned numbers (multu). The instruction

```
mult   Rsrc1,Rsrc2
```

multiplies the 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 contents of HI register to the general register Rdest. Use mflo to move data from the LO register. For movement of data into these special registers, use mthi (Move To HI) or mtlo (Move To LO).

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-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, use the `mulo` instruction. Both these instructions treat the numbers as signed. To multiply two unsigned numbers, 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 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.

Assembler provides three-operand divide pseudoinstructions similar to the multiply instructions. The instruction

```
div†    Rdest,Rsrc1,Src2
```

places the quotient of two signed number division `Rsrc1/Src2` in `Rdest`. As in the other instructions, `Src2` can be a register or an immediate value. For unsigned numbers, use the `divu`† pseudoinstruction. The quotient is rounded toward zero. Overflow is signaled when dividing -2^{31} by -1 as the quotient is greater than $2^{31} - 1$. Assembler generates the real `div` instruction if we use

Table 13.3 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`.

```
div     $0,Rsrc1,Src2.
```

To get the remainder instead of quotient, use

```
rem†     Rdest,Rsrc1,Src2
```

for signed numbers.

13.1.4 Logical Instructions

MIPS supports the logical instructions `and`, `or`, `nor` (OR followed by NOT), 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 operands. As with most instructions, all operands must be registers. However, `and`, `or`, and `xor` instructions can take one immediate operand.

A summary of these instructions is given in Table 13.3. 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
```

Table 13.4 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`.

13.1.5 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 five bits are used as the shift count.

The basic left-shift instruction `sll` (Left Shift 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 will see 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. The reason for this is that when we shift right, we have the option of filling the vacated left bits by zeros (called

Table 13.5 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.

logical shift) or copying the sign bit (arithmetic shift). The difference between the logical and arithmetic shifts is explained in detail in Section 9.2 on page 278.

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 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 13.4.

13.1.6 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, assembler provides two rotate pseudoinstructions—rotate left (`rol`) and rotate right (`ror`).

In Rotate Left, the bits shifted out at the left (i.e., the sign-bit side) are inserted on the right-hand side. In Rotate Right, bits falling of the right side are inserted on the sign-bit side.

Table 13.5 summarizes the two rotate instructions. Both rotate instructions 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
```

Note that `$t2` maps to the `$10` register.

13.1.7 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 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 13.6 summarizes the comparison instructions provided by the assembler.

All comparison instructions in Table 13.6 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, $6, respectively. The branch instructions are discussed next.

13.1.8 Branch and Jump Instructions

Conditional execution is implemented by jump and branch instructions. We will first look at the jump instructions. The basic jump instruction

```
j      target
```

Table 13.6 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. Source operands are treated as signed numbers.
`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. Source operands are treated as signed numbers.
`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. Source operands are treated as signed numbers.
`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. Source operands are treated as signed numbers.
`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 the `target` address. There are other jump instructions that are useful in procedure calls. There instructions are discussed in Section 13.4.

Branch instructions provide a more flexible test and jump execution. MIPS supports several branch instructions. We describe some of these instructions in this section.

The unconditional branch instruction

```
b†    target
```

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 a 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 `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 is 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 13.7 summarizes some of the branch instructions provided by the MIPS assembler.

13.2 SPIM Simulator

This section presents the SPIM system calls and the directives provided by it. Other details on this simulator are given in Appendix D.

13.2.1 SPIM System Calls

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 `$v0` register. Any required arguments are placed in `$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).

Table 13.7 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 13.8 SPIM Assembler System Calls

Service	System call code (in `$v0`)	Arguments	Result
`print_int`	1	`$a0` = integer	
`print_float`	2	`$f12` = float	
`print_double`	3	`$f12` = double	
`print_string`	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		

All 10 system calls are summarized in Table 13.8. The first three 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.

13.2.2 SPIM Assembler Directives

SPIM supports a subset of the assembler directives provided by the MIPS assembler. This section presents some of the most common SPIM directives. SPIM reference manual 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`. SPIM allows only instructions or words (using `.WORD`) in the text segment.

The data directive has a similar format as `.TEXT` except that the statement following it must refer to data items.

String Directives

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 to break 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

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, use the `.WORD` directive. 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.

Before closing this section, 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 SPIM's trap file (see Appendix D for details on the trap file). The program template given next shows how the global directive is used.

13.2.3 MIPS Program Template

Our MIPS programs follow the template shown in Figure 13.2.

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

      la    $a0,in_name    ; read name
      li    $a1,31
      li    $v0,8
      syscall
          . . .
```

```
# Title of the program                           Filename
#
# Objective:
#     Input:
#    Output:
#
#    Register usage
#
###################### Data segment ##########################

      .data
          . . .
      data goes here
          . . .

###################### Code segment ##########################
      .text
      .globl main
main:
          . . .
      code goes here
          . . .
```

Figure 13.2 MIPS assembly language program template.

13.3 Illustrative Examples

We use four example programs to illustrate the MIPS assembly language features. On purpose, we have selected the examples from Chapter 4 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 4.8 starting on page 98, understanding the MIPS versions becomes easier, as the underlying algorithms are the same. These examples can be run on the MIPS simulator SPIM. Appendix D gives details about downloading and using the SPIM simulator.

Example 13.1 *Displays the ASCII value of the input character in binary representation.* This is the MIPS version of the example discussed on page 98. It takes a character as input and displays its ASCII value in binary. Since 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 4.6 on page 98. 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 the `and` and `beqz` instructions 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 13.1 Conversion of an 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:
      .text
      .globl main
main:
      la    $a0,ch_prompt      # prompt user for a character
      li    $v0,4
      syscall

      la    $a0,ch             # read the input character
      li    $a1,2
      li    $v0,8
      syscall

      la    $a0,out_msg        # write output message
      li    $v0,4
      syscall

      lb    $t0,ch             # t0 holds the character
      la    $t1,ascii_string   # t1 points to output string
      li    $t2,0x80           # t2 holds the mask byte
      li    $t4,’0’
      li    $t5,’1’

loop:
      and   $t3,$t0,$t2
      beqz  $t3,zero
      sb    $t5,($t1)           # store 1
      b     rotate
zero:
      sb    $t4,($t1)           # store 0
rotate:
      srl   $t2,$t2,1           # shift mask byte
      addu  $t1,$t1,1
      bnez  $t2,loop            # exit loop if mask byte is 0

      sb    $0,($t1)            # append NULL
      la    $a0,ascii_string    # output ASCII value
      li    $v0,4
      syscall

      la    $a0,newline         # output newline
      li    $v0,4
      syscall
```

Example 13.2 *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 same example on page 105, 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 13.2 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
```

```

      la    $a0,in_name      # read user input string
      li    $a1,31
      li    $v0,8
      syscall

      la    $a0,out_msg      # write output message
      li    $v0,4
      syscall

      la    $t0,in_name
loop:
      lb    $t1,($t0)
      beqz  $t1,exit_loop    # if NULL, we are done
      blt   $t1,'a',no_change
      bgt   $t1,'z',no_change
      addu  $t1,$t1,-32      # convert to uppercase
                             # 'A'-'a' = -32
no_change:
      sb    $t1,($t0)
      addu  $t0,$t0,1        # increment pointer
      j     loop
exit_loop:
      la    $a0,in_name
      li    $v0,4
      syscall
```

Example 13.3 *Addition of individual digits of an integer—string version.*

We have done the Pentium version of this example on page 107. 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 12345 as the 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, the character 5 is represented in ASCII as `00110101` when expressed in binary. To convert this to the number 5, we force the upper four bits to zero. This is what the loop body (lines 42–50) 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 13.3 Addition of individual digits—string version

```
# Add individual digits of a number          ADDIGITS.ASM
#
# Objective: To add individual digits of an integer.
#            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
```

```
39:
40:        la    $t0,number         # pointer to number
41:        li    $t2,0              # init sum to zero
42:  loop:
43:        lb    $t1,($t0)
44:        beq   $t1,0xA,exit_loop  # if CR, we are done, or
45:        beqz  $t1,exit_loop      # if NULL, we are done
46:        and   $t1,$t1,0x0F       # strip off upper 4 bits
47:        addu  $t2,$t2,$t1        # add to running total
48:
49:        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 13.4 *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 `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 12345 by 10 gives us 5 as the remainder and 1234 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 the HI and LO special registers. Special move instructions `mflo` and `mfhi` are used to copy these two values into the `t0` and `t3` registers (lines 44 and 45). The loop terminates if the quotient (in `t0`) is zero.

Program 13.4 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
```

```
#    t2 - maintains the running sum
#    t3 - holds the remainder
#
################### Data segment #####################

      .data
number_prompt:
      .asciiz  "Please enter an integer: \n"
out_msg:
      .asciiz  "The sum of individual digits is: "

################### Code segment #####################

      .text
      .globl main
main:
      la    $a0,number_prompt  # prompt user for input
      li    $v0,4
      syscall

      li    $v0,5              # read the input number
      syscall                  # input number in $v0
      move  $t0,$v0
      abs   $t0,$t0            # get absolute value

      la    $a0,out_msg        # write output message
      li    $v0,4
      syscall

      li    $t1,10             # $t1 holds divisor 10
      li    $t2,0              # init sum to zero
loop:
      divu  $t0,$t1            # $t0/$t1
      # leaves quotient in LO and remainder in HI
      mflo  $t0                # move quotient to $t0
      mfhi  $t3                # move remainder to $t3
      addu  $t2,$t2,$t3        # add to running total
      beqz  $t0,exit_loop      # exit loop if quotient is 0
      j     loop
exit_loop:
      move  $a0,$t2            # output sum
      li    $v0,1
      syscall
```

13.4 Procedures

MIPS provides two instructions to support procedures—jal and jr. These correspond to the call and ret instructions of the IA-32 architecture. The jal (jump and link) instruction

```
jal    proc_name
```

transfers control to proc_name just like a jump instruction. Since we need the return address, it also stores the address of the instruction following jal in ra register.

To return from a procedure, we use

```
jr     $ra
```

which reads the return address from the ra register and transfers control to this address.

In the IA-32 architecture, 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 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 an IA-32 processor. However, this advantage is lost when we use recursive or nested procedures. This point will become clear when we discuss recursive procedure examples in Chapter 16.

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 4. In the IA-32 architecture, register-based parameter passing is fairly restrictive due to the small number of registers available. However, the large number of registers in MIPS makes this method attractive. We now use two examples to illustrate how procedures are written in the MIPS assembly language.

Example 13.5 *Finds minimum and maximum of three numbers.*

This is a simple program to explain the basics of procedures in 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 back to the main program (see lines 83 and 97).

Program 13.5 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.
```

```
#     Input: Requests three numbers from keyboard.
#    Output: Outputs the minimum and maximum.
#
#    a1, a2, a3 - three numbers are passed via these registers
#
###################### Data segment ##########################
      .data
prompt:
      .asciiz     "Please enter three numbers: \n"
min_msg:
      .asciiz     "The minimun is: "
max_msg:
      .asciiz     "\nThe maximum is: "
newline:
      .asciiz     "\n"

###################### Code segment ##########################

      .text
      .globl main
main:
      la    $a0,prompt         # prompt user for input
      li    $v0,4
      syscall

      li    $v0,5              # read the first number into $a1
      syscall
      move  $a1,$v0

      li    $v0,5              # read the second number into $a2
      syscall
      move  $a2,$v0

      li    $v0,5              # read the third number into $a3
      syscall
      move  $a3,$v0

      jal   find_min
      move  $s0,$v0

      jal   find_max
      move  $s1,$v0

      la    $a0,min_msg        # write minimum message
      li    $v0,4
      syscall

      move  $a0,$s0            # output minimum
      li    $v0,1
      syscall

      la    $a0,max_msg        # write maximum message
      li    $v0,4
      syscall

      move  $a0,$s1            # output maximum
      li    $v0,1
      syscall

      la    $a0,newline        # write newline
      li    $v0,4
      syscall

      li    $v0,10             # exit
      syscall

#------------------------------------------------------------
# FIND_MIN receives three integers in $a0, $a1, and $a2 and
# returns the minimum of the three in $v0
#------------------------------------------------------------
find_min:
      move  $v0,$a1
      ble   $v0,$a2,min_skip_a2
      move  $v0,$a2
min_skip_a2:
      ble   $v0,$a3,min_skip_a3
      move  $v0,$a3
min_skip_a3:
      jr    $ra

#------------------------------------------------------------
# FIND_MAX receives three integers in $a0, $a1, and $a2 and
# returns the maximum of the three in $v0
#------------------------------------------------------------
find_max:
      move  $v0,$a1
      bge   $v0,$a2,max_skip_a2
      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 13.6 *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 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 13.3). These two termination conditions are detected on lines 63 and 64.

Program 13.6 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:
      la    $a0,prompt          # prompt user for input
      li    $v0,4
      syscall

      la    $a0,in_string       # read input string
      li    $a1,31              # buffer length in $a1
      li    $v0,8
      syscall

      la    $a0,in_string       # call string length proc.
      jal   string_len
      move  $t0,$v0             # string length in $v0

      la    $a0,out_msg         # write output message
      li    $v0,4
      syscall

      move  $a0,$t0             # output string length
      li    $v0,1
      syscall

      la    $a0,newline         # write newline
      li    $v0,4
      syscall

      li    $v0,10              # exit
      syscall

#-----------------------------------------------------------
# STRING_LEN receives a pointer to a string in $a0 and
# returns the string length in $v0
#-----------------------------------------------------------
string_len:
      li    $v0,0               # init $v0 (string length)
loop:
      lb    $t0,($a0)
      beq   $t0,0xA,done        # if CR
      beqz  $t0,done            # or NULL, we are done
      addu  $a0,$a0,1
      addu  $v0,$v0,1
      b     loop
done:
      jr    $ra
```

13.5 Stack Implementation

MIPS does not explicitly support stack operations. In contrast, recall that the IA-32 architecture provides instructions such as push and pop to facilitate stack operations. In addition, a special stack pointer register (ESP) keeps the top-of-stack information. In MIPS, a register plays the role of the stack pointer. We have to manipulate this register to implement the stack.

MIPS stack implementation has some similarities to the IA-32 implementation. For example, 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
```

Example 13.7 *Passing a variable number of parameters via the stack.*
We use the variable parameter example discussed on page 146 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, MIPS allows a more flexible access to parameters. In Pentium, because the return address is pushed onto the stack, we had to use the EBP 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. We do not have that problem in the MIPS implementation.

Program 13.7 Passing variable number of parameters to a procedure

```
# Sum of variable number of integers         VAR_PARA.ASM
#
# Objective: Finds sum of variable number of integers.
#            Stack is used to pass variable number of integers.
#            To demonstrate stack-based parameter passing.
#     Input: Requests integers from the user;
#            terminated by entering a zero.
#    Output: Outputs the sum of input numbers.
#
#    a0 - number of integers passed via the stack
#
###################### Data segment ##########################
      .data
prompt:
      .ascii     "Please enter integers. \n"
      .asciiz    "Entering zero terminates the input. \n"
sum_msg:
      .asciiz     "The sum is: "
newline:
      .asciiz     "\n"

###################### Code segment ##########################

      .text
      .globl main
main:
      la    $a0,prompt         # prompt user for input
      li    $v0,4
      syscall

      li    $a0,0
read_more:
      li    $v0,5              # read a number
      syscall
      beqz  $v0,exit_read
      subu  $sp,$sp,4          # reserve 4 bytes on stack
      sw    $v0,($sp)          # store the number on stack
      addu  $a0,$a0,1
      b     read_more
exit_read:
      jal   sum                # sum is returned in $v0
      move  $s0,$v0

      la    $a0,sum_msg        # write output message
      li    $v0,4
      syscall

      move  $a0,$s0            # output sum
      li    $v0,1
      syscall

      la    $a0,newline        # write newline
      li    $v0,4
      syscall

      li    $v0,10             # exit
      syscall

#-----------------------------------------------------------
# SUM receives the number of integers passed in $a0 and the
# actual numbers via the stack. It returns the sum in $v0.
#-----------------------------------------------------------
sum:
      li    $v0,0
sum_loop:
      beqz  $a0,done
      lw    $t0,($sp)
      addu  $sp,$sp,4
      addu  $v0,$v0,$t0
      subu  $a0,$a0,1
      b     sum_loop
done:
      jr    $ra
```

13.6 Summary

We have discussed the MIPS assembly language in detail. MIPS provides several basic instructions; the assembler supplements these instructions by several useful pseudoinstructions. Unlike the IA-32 architecture, all instructions take 32 bits to encode. MIPS uses only three different types of instruction formats.

MIPS architecture 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 register as she wishes. However, certain convention has 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 is to bring out the differences between the two assembly languages.

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
- Call-by-reference
- Call-by-value
- Instruction format
- MIPS stack implementation
- Procedure call
- SPIM assembler directives
- SPIM segments
- SPIM system calls
- Relative address

13.7 Exercises

13–1 What is the difference between a pseudoinstruction and an instruction?

13–2 What is the difference between `.ASCII` and `.ASCIIZ` assembler directives?

13–3 The IA-32 architecture provides instructions and registers to implement the stack. In the MIPS architecture there is no such support for stack implementation. Describe how the stack is implemented in MIPS.

13–4 Discuss the differences between the multiply instructions of the IA-32 and MIPS architectures.

13–5 Discuss the differences between the divide instructions of the IA-32 and MIPS architectures.

13–6 Describe the procedure invocation and return mechanisms in MIPS.

13–7 Write a simple assembly language program to see how the branch (`b`) and jump (`j`) instructions are translated. Use SPIM to do the translation. You can use the "Text Segment" display window of SPIM to check the translation (see Appendix D for details on SPIM windows).

13–8 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
```

13–9 Discuss the differences between IA-32 and MIPS architectures to pass a variable number of parameters to procedures.

13–10 In the IA-32 architecture, register-based parameter passing mechanism is not pragmatic. Why does it make sense to use this method in MIPS architecture?

13–11 Why are simple procedure calls and returns faster in MIPS than in IA-32?

13–12 Discuss the differences between IA-32 and MIPS architectures in accessing multidimensional arrays.

13.8 Programming Exercises

13–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
```

13–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 whether he or 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.

13–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.

13–P4 Modify the above program to display the octal equivalent instead of the hexadecimal equivalent of the input number.

13–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.

13–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.

13–P7 Write a procedure that receives a string (i.e., 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.

13–P8 Write a procedure that receives a string (i.e., 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.

13–P9 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 size of the matrices. Note that both **A** and **B** should have the same size. The `main` procedure calls another procedure to display **C**.

13–P10 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.

13–P11 We have discussed merge sort in Programming Exercise 8–P7 (page 269). Write a MIPS assembly language program to implement the merge sort.

13–P12 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`.

PART IV

Interrupt Processing

This part is dedicated to Pentium's interrupt processing mechanism. We cover both protected-mode and real-mode interrupt processing. It consists of two chapters: Chapters 14 and 15.

Chapter 14 gives details on protected-mode interrupt processing. This chapter uses the Linux system calls to facilitate our discussion of software interrupts.

Chapter 15 discusses the real-mode interrupt processing. This is the only chapter in this book that uses DOS to explore programmed I/O and interrupt-driven I/O. The programs given in Chapter 15 have been tested in the DOS window (Command Prompt) under Windows XP. To run these programs, you have to copy the DOS versions of the I/O files—`io.mac` and `io.obj`.

Chapter 14

Protected-Mode Interrupt Processing

Objectives

- To describe the protected-mode interrupt mechanism of Pentium
- To explain software and hardware interrupts
- To introduce Pentium exceptions
- To illustrate how Linux system calls can be used to interface with system I/O devices

Interrupts, like procedures, can be used to alter a program's control flow to a procedure called an interrupt service routine. Unlike procedures, which can be invoked by a `call` *instruction, interrupt service routines can be invoked either in software (called software interrupts) or by hardware (called hardware interrupts). After introducing interrupts in Section 14.1 we discuss a taxonomy of interrupts in Section 14.2. This chapter looks at the protected-mode interrupt mechanism. The real-mode interrupts are discussed in the next chapter. The interrupt invocation mechanism in the protected mode is described in Section 14.3. Pentium exceptions are discussed in Section 14.4. The next two sections deal with software interrupts and file I/O. We use Linux system calls to illustrate how we can access I/O devices like the keyboard and the display (Section 14.7). Hardware interrupts are introduced in Section 14.8. The last section summarizes the chapter.*

14.1 Introduction

Interrupt is a mechanism by which a program's flow control can be altered. We have seen two other mechanisms to do the same: *procedures* and *jumps*. While jumps provide a one-way transfer of control, 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* (ISR). An ISR is also called a *handler*. When the ISR is completed, the interrupted 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 much of the power of interrupts. This capability gives us an efficient way by which external devices (outside the processor) can get the processor's attention.

Software-initiated interrupts—called simply *software interrupts*—are caused by executing the `int` instruction. Thus these 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 immediate attention of the processor? Suppose that you have written a program to display the first 90 Fibonacci numbers on the screen. While running the program, however, you 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, this can be done by `ctrl-c` in Linux (`ctrl-break` on Windows). The important point is that this is not an anticipated event—so cannot be effectively programmed into the code.

The interrupt mechanism provides an efficient way to handle unanticipated events. Referring to the previous example, the `ctrl-c` could cause an interrupt to draw the attention of the processor away from the user program. The interrupt service routine associated with `ctrl-c` can terminate the program and return control to the operating system.

Another difference between procedures and interrupts is that ISRs are normally memory-resident. In contrast, procedures are loaded into memory along with application programs. Some other differences—such as using numbers to identify interrupts rather than names, using an invocation mechanism that automatically pushes the flags register onto the stack, and so on—are pointed out in later sections.

14.2 A Taxonomy of Interrupts

We have already identified two basic categories of interrupts—software-initiated and hardware-initiated (see Figure 14.1). 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 execution if the divisor is 0. We discuss exceptions in Section 14.4.

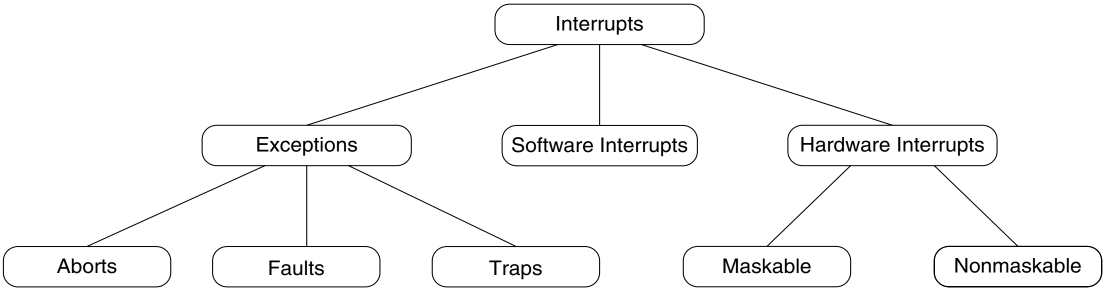


Figure 14.1 A taxonomy of Pentium interrupts.

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, etc. Software interrupts can be further classified into *system-defined* and *user-defined*.

Hardware interrupts are generated by hardware devices to get the attention of the processor. For example, when you strike a key, the keyboard hardware generates an external interrupt, causing the processor to suspend its present activity and execute the keyboard interrupt service routine to process the key. After completing the keyboard ISR, the processor resumes what it was doing before the interruption.

Hardware interrupts can be either *maskable* or *nonmaskable*. The processor always attends the nonmaskable interrupt (NMI) immediately. One example of 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 processor is executing a `main` program. An interrupt occurs. As a result, the processor suspends `main` as soon as it finishes the current instruction and transfers control to the ISR1. If ISR1 has to be executed without any interruption, the processor can mask further interrupts until it is completed. Suppose that, while executing ISR1, another maskable interrupt occurs. Service to this interrupt would have to wait until ISR1 is completed. We discuss hardware interrupts in a later section.

14.3 Interrupt Processing in the Protected Mode

Let's look at interrupt processing in the protected mode. Unlike procedures, where a name is given to identify a procedure, interrupts are identified by a type number. Pentium supports 256 different interrupt types. The interrupt type ranges from 0 to 255. The interrupt type number, which is also called a *vector*, is used as an index into a table that stores the addresses of ISRs. This table is called the *interrupt descriptor table* (IDT). Like the global and local descriptor tables GDT and LDT (discussed in Chapter 3), each descriptor is essentially a pointer to an ISR and requires eight bytes. The interrupt type number is scaled by 8 to form an index into the IDT.

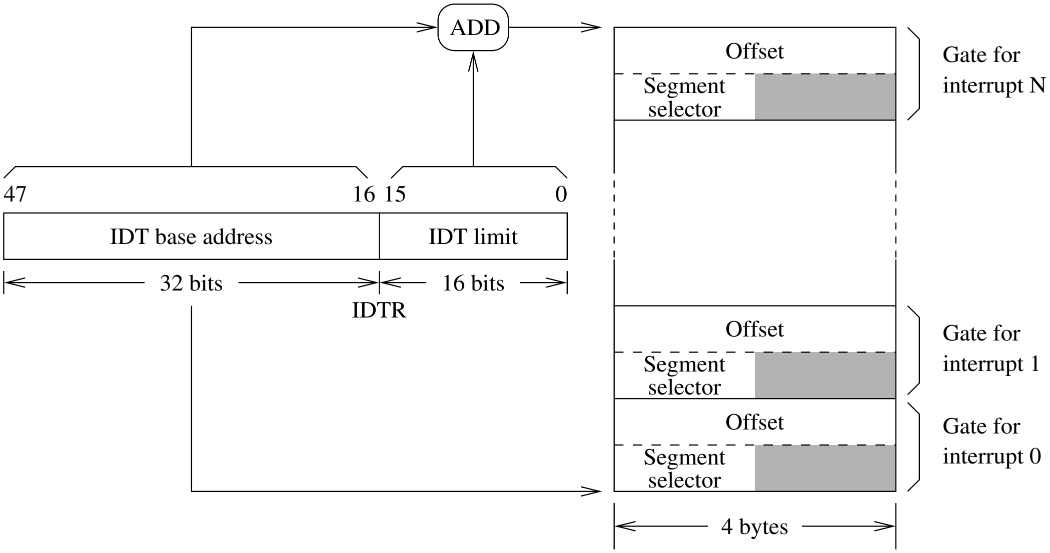


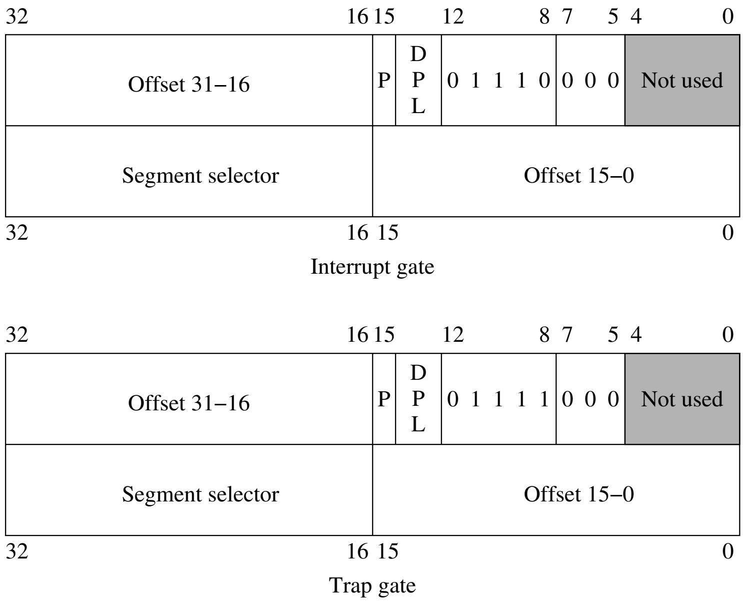
Figure 14.2 Organization of the IDT. The IDTR register stores the 32-bit IDT base address and a 16-bit value indicating the IDT size.

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 the 32-bit IDT base address and a 16-bit IDT limit value as shown in Figure 14.2. 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 the referenced descriptor is outside the IDT limit, the processor enters the shutdown mode. In this mode, instruction execution is stopped until either a nonmaskable interrupt or a reset signal is received.

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.

The IDT can have three types of descriptors: interrupt gate, trap gate, and task gate. We will not discuss task gates, as they are not directly related to the interrupt mechanism that we are interested in. The format of the other two gates is shown in Figure 14.3. Both gates store identical information: a 16-bit segment selector, a 32-bit offset, a descriptor privilege level (DPL), and a P bit to indicate whether the segment is present or not.

When an interrupt occurs, the segment selector is used to select a segment descriptor that is in either the GDT or the current LDT. Recall from our discussion in Chapter 3 that the TI bit of the segment descriptor identifies whether the GDT or the current LDT should be used. The segment descriptor provides the base address of segment that contains the interrupt service routine as shown in Figure 14.4. The offset part comes from the interrupt gate.



Interrupt gate

Trap gate

Figure 14.3 Pentium interrupt descriptors.

What happens when an interrupt occurs depends on whether there is a privilege change or not. In the remainder of the chapter, we look at the simple case of no privilege change. In this case, the following actions are taken on an interrupt:

1. Push the EFLAGS register onto the stack;
2. Clear the interrupt and trap flags;
3. Push CS and EIP registers onto the stack;
4. Load CS with the 16-bit segment selector from the interrupt gate;
5. Load EIP with the 32-bit offset values from the interrupt gate.

On receiving an interrupt, the flags register is automatically saved on the stack. The interrupt and trap flags are cleared to disable further interrupts. Usually, this flag is set in ISRs 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. Both of these instructions require no operands. There are no special instructions to manipulate the trap flag. We have to use `popf` and `pushf` to modify the trap flag. We give an example of this in the next chapter (see page 441).

The current CS and EIP values are pushed onto the stack. The CS and EIP registers are loaded with the segment selector and offset from the interrupt gate, respectively. Note that

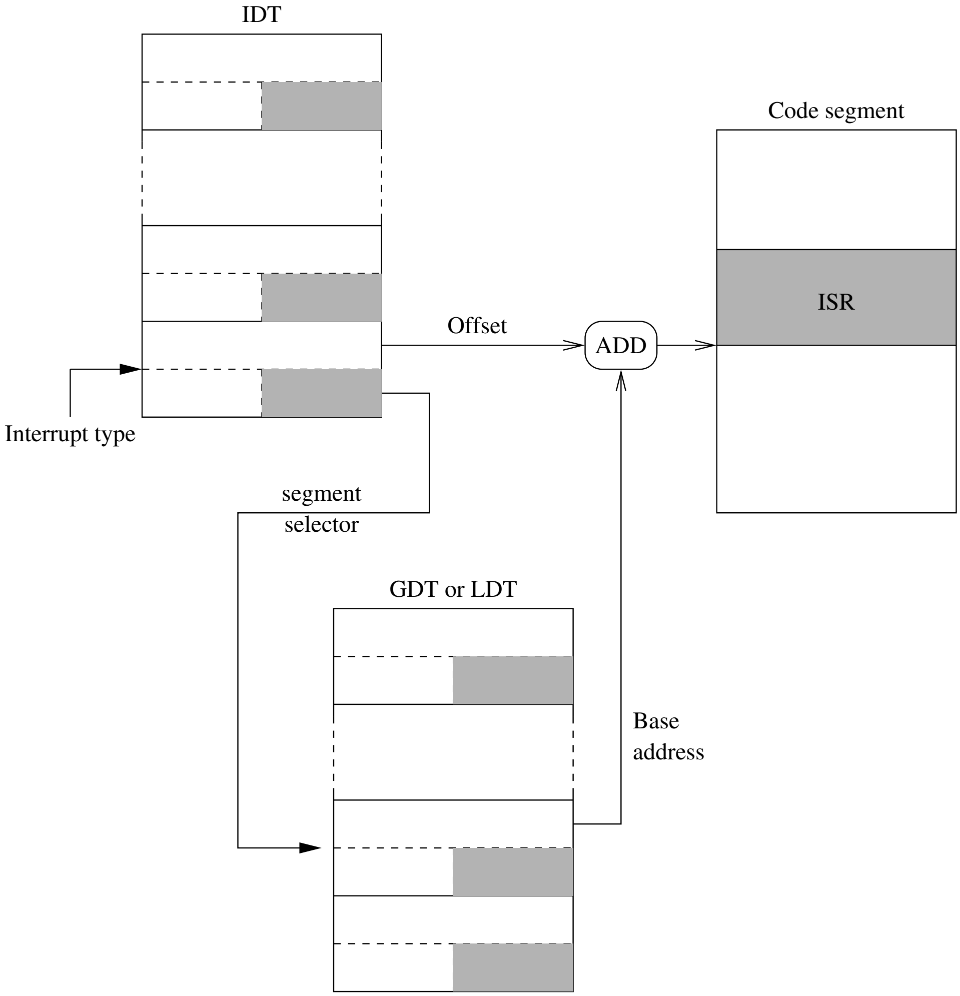


Figure 14.4 Protected-mode Pentium interrupt invocation.

when we load the CS register with the 16-bit segment selector, the invisible part consisting of the base address, segment limit, access rights, and so on is also loaded. The stack state after an interrupt is shown in Figure 14.5a.

Interrupt processing through a trap gate is similar to that through an interrupt gate except for the fact that trap gates do not modify the IF flag.

While the previous discussion holds for all interrupts and traps, some types of exceptions also push an error code onto the stack as shown Figure 14.5b. The exception handler can use this error code in identifying the cause for the exception.

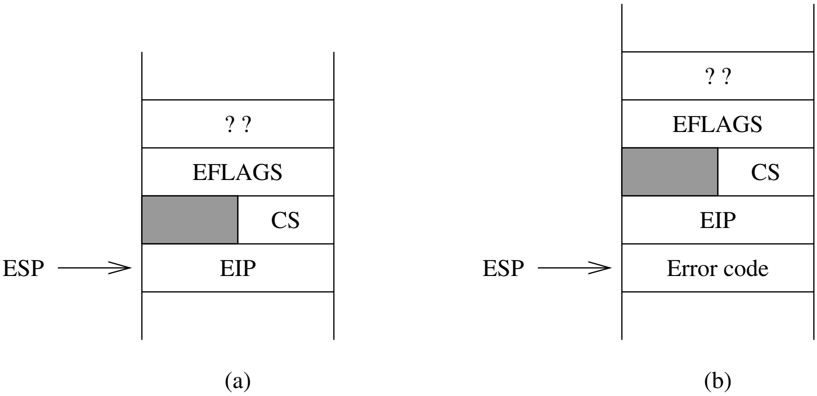


Figure 14.5 Stack state after an interrupt invocation.

Returning from an interrupt handler Just like procedures, ISRs should end with a return statement to send control back to the interrupted program. The interrupt return (`iret`) is used for this purpose. The last instruction of an ISR should be the `iret` instruction. It serves the same purpose as `ret` for procedures. The actions taken on `iret` are

1. Pop the 32-bit value on top of the stack into the EIP register;
2. Pop the 16-bit value on top of the stack into the CS register;
3. Pop the 32-bit value on top of the stack into the EFLAGS register.

14.4 Exceptions

The 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.

Table 14.1 The First Five Dedicated Interrupts

Interrupt type	Purpose
0	Divide error
1	Single-step
2	Nonmaskable interrupt (NMI)
3	Breakpoint
4	Overflow

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 predefined interrupts. These are called *dedicated* interrupts. These include the first five interrupts as shown in Table 14.1. The NMI is a hardware interrupt and is discussed in Section 14.8. A brief description of the remaining four interrupts is given here.

Divide Error Interrupt: The processor 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 on Linux displays a *Floating point exception* 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 worry about them here.

The interrupt handler for the 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. In the next chapter, we present an example program that initiates and stops single-stepping (see Section 15.5 on page 441).

To end single stepping, the TF should be cleared. The instruction set, however, does not have instructions to directly manipulate the TF bit. Instead, we have to resort to indirect means. This is illustrated in the next chapter by means of an example (see page 441).

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`.

14.5 Software Interrupts

Software interrupts are initiated by executing an interrupt instruction. The format of this instruction is

```
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, Linux provides a large number of services via `int 0x80`. In fact, it provides more than 180 different system calls! All these system calls are invoked by `int 0x80`. The required service is identified by placing the system call number in the EAX register. If the number of arguments required for a systems call is less than six, these are placed in other registers. Usually, the system call also returns values in registers. We give details on some of the file access services provided by `int 0x80` in the next section.

Linux System Calls

Of the 256 interrupt vectors supported by Pentium, Linux uses the first 32 vectors (i.e., from 0 to 31) for exceptions and nonmaskable interrupts. The next 16 vectors (from 32 to 47) are used for hardware interrupts generated through interrupt request lines (IRQs) (discussed in the next chapter). It uses one vector (128 or 0x80) for software interrupt to provide system services. Even though only one interrupt vector is used for system services, Linux provides several services using this interrupt.

14.6 File I/O

In this section we give several examples to perform file I/O operations. In Linux as in UNIX, the keyboard and display are treated as stream files. So reading from the keyboard is not any different from reading a file from the disk. If you have done some file I/O in C, it is relatively easy to understand the following examples. Don't worry if you are not familiar with the file I/O; we give enough details here.

The system sees the input and output data as a stream of bytes. It does not make any logical distinction whether the byte stream is coming from a disk file or the keyboard. This makes it easy to interface with the I/O devices like keyboard and display. Three standard file streams are defined: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). The default association for the standard input is the keyboard; for the other two, it is the display.

14.6.1 File Descriptor

For each open file, a small 16-bit integer is assigned as a file id. These magic numbers are called the *file descriptors*. Before accessing a file, it must first be opened or created. To open or create a file, we need the file name, mode in which it should be opened or created, and so on. The file descriptor is returned by the file `open` or `create` system calls. Once a file is open or created, we use the file descriptor to access the file.

We don't have to open the three standard files mentioned above. They are automatically opened for us. These files are assigned the lowest three integers: `stdin` (0), `stdout` (1), and `stderr` (2).

14.6.2 File Pointer

A file pointer is associated with each open file. The file pointer specifies an offset in bytes into the file relative to the beginning of the file. A file itself is viewed as a sequence of bytes or characters. The file pointer specifies the location in the file for the subsequent read or write operation.

When a file is opened, the file pointer of that file is set to zero. In other words, the file pointer points to the first byte of the file. Sequential access to the file is provided by updating the file pointer to move past the data read or written. Direct access, or opposed to sequential access, to a file is provided by simply manipulating the file pointer.

14.6.3 File System Calls

System calls described in this section provide access to the data in disk files. As discussed previously, before accessing the data stored in a file, we have to open the file. We can only open a file if it already exists. Otherwise, we have to create a new file, in which case there is no data and our intent should be to write something into the file. Linux provides two separate functions—one to open an existing file (system call 5) and the other to create a new file (system call 8).

Once a file is opened or created, the data from that file can be read or data can be written into the file. We can use system call 3 to read data from a file and data can be written to a file by using system call 4. In addition, since disks allow direct access to the data stored, data contained in a disk file can be accessed directly or randomly. To provide direct access to the data stored in a file, the file pointer should be moved to the desired position in the file. The system call 19 facilitates this process. Finally, when processing of data is completed we should close the file. We use system call number 6 to close an open file.

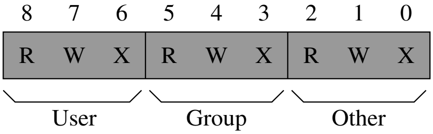
A file name (you can include the path if you wish) is needed only to open or create file. Once a file is opened or created, a file descriptor is returned and all subsequent accesses to the file should use this file descriptor.

The remainder of this section describes some of the file system calls.

System call 8 — Create and open a file

Inputs:	EAX	=	8
	EBX	=	file name
	ECX	=	file permissions
Returns:	EAX	=	file descriptor
Error:	EAX	=	error code

This system call can be used to create a new file. The EBX should point to the file name string, which can include the path. The ECX should be loaded with file permissions for owner, group and others as you would in the Linux (using `chmod` command) to set the file permissions. File permissions are represented by three groups of three bits as shown below:



For each group, you can specify read (R), write (W), and execute (X) permissions. For example, if you want to give read, write, and execute for the owner but no access to anyone else, set the three owner permission bits to 1 and other bits to 0. Using the octal number system, we represent this number as 0700. If you want to give read, write, and execute for the owner, read permission to the group, and no access to others, you can set the permissions as 0740. (Note that octal numbers are indicated by prefixing them with a zero as in the examples here.)

The file is opened in read/write access mode and a file descriptor (a positive integer) is returned in EAX if there is no error. In case of an error, the error code (a negative integer) is placed in EAX. For example, a create error may occur due to nonexistent directory in the specified path, specified file already exists, or if there are device access problems, and so on. As we see next, we can also use file open to create a file.

System call 5 — Open a file

Inputs:	EAX	=	5
	EBX	=	file name
	ECX	=	file access mode
	EDX	=	file permissions
Returns:	EAX	=	file descriptor
Error:	EAX	=	error code

This function can be used to open an existing file. It takes the file name and file mode information as in the file-create system call. In addition, it takes the file access mode in ECX register. This field gives information on how the file can be accessed. Some interesting values are read-only (0), write-only (1), and read-write (2). Why is access mode specification important? The simple answer is to provide security. A file that is used as an input file to a program can be opened as a read-only file. Similarly, an output file can be opened as a write-only file. This eliminates accidental writes or reads. This specification facilitates, for example, access to files for which you have read-only access permission.

We can use this system call to create a file by specifying 0100 for file access mode. This is equivalent to the file-create system call we discussed before. We can erase contents of a file by specifying 01000 for the access mode. This leaves the file pointer at the beginning of the file. If we want to append to the existing contents, we can specify 02000 to leave the file pointer at the end.

As with the create system call, file descriptor and error code are returned in the EAX register.

System call 3 — Read from a file

Inputs:	EAX	=	3
	EBX	=	file descriptor
	ECX	=	pointer to input buffer
	EDX	=	buffer size (maximum number of bytes to read)
Returns:	EAX	=	number of bytes read
Error:	EAX	=	error code

Before calling this function to read data from a previously opened or created file, the number of bytes to read should be specified in EDX and ECX should point to a data buffer into which the data read from the file is placed. The file is identified by giving its descriptor in EBX.

The system attempts to read EDX bytes from the file starting from the current file pointer location. Thus, by manipulating the file pointer (see `lseek` system call discussed later), we can use this function to read data from a random location in a file.

After the read is complete, the file pointer is updated to point to the byte after the last byte read. Thus, successive calls would give us sequential access to the file.

Upon completion, if there is no error, EAX contains the actual number of bytes read from the file. If this number is less than that specified in EDX, the only reasonable explanation is that the end of file has been reached. Thus, we can use this condition to detect `end-of-file`.

System call 4 — Write to a file

Inputs:	EAX	=	4
	EBX	=	file descriptor
	ECX	=	pointer to output buffer
	EDX	=	buffer size (number bytes to write)
Returns:	EAX	=	number of bytes written
Error:	EAX	=	error code

This function can be used to write to a file that is open in write or read/write access mode. Of course, if a file created, it is automatically opened in read/write access mode. The input parameters have similar meaning as in the read system call. On return, if there is no error, EAX contains the actual number of bytes written to the file. This number should normally be equal to that specified in EDX. If not, there was an error—possibly due to disk full condition.

System call 6 — Close a file

Inputs:	EAX	=	6
	EBX	=	file descriptor
Returns:	EAX	=	—
Error:	EAX	=	error code

This function can be used to close an open file. It is not usually necessary to check for errors after closing a file. The only reasonable error scenario is when EBX contains an invalid file descriptor.

System call 19 — lseek (Updates file pointer)

Inputs:	EAX	=	19
	EBX	=	file descriptor
	ECX	=	offset
	EDX	=	whence
Returns:	EAX	=	byte offset from the beginning of file
Error:	EAX	=	error code

Thus far, we processed files sequentially. The file pointer remembers the position in the file. As we read from or write to the file, the file pointer is advanced accordingly. If we want to have random access to a file rather than accessing sequentially, we need to manipulate the file pointer.

This system call allows us to reposition the file pointer. As usual, the file descriptor is loaded into EBX. The offset to be added to the file pointer is given in ECX. This offset can added relative to the beginning of file, end of file, or current position. The `whence` value in EDX specifies this reference point:

Reference position	**whence value**
Beginning of file	0
Current position	1
End of file	2

These system calls allow us to write file I/O programs. Since keyboard and display are treated as files as well, we can write assembly language programs to access these I/O devices.

14.7 Illustrative Examples

We present three examples that use the file I/O system calls described in the last section. The first two are taken from the I/O routines we have used (see Appendix B for details).

Example 14.1 *Procedure to write a character.*

In this example we look at the `PutCh` procedure we used to write a character to the display. This is done by using the write system call. We specify `stdout` as the file to be written. The procedure is shown in Program 14.1. Since the character to be displayed is received in the AL register, we store it in `temp_char` before loading EAX with system call number 4. We load the `temp_char` pointer in ECX. Since we want to read just one character, we load 1 into EDX (line 10). We preserve the registers by using `pusha` and `popa` on lines 5 and 12.

Program 14.1 Procedure to write a character to the display

```
 1:  ;------------------------------------------------------------
 2:  ; Put character procedure receives the character in AL.
 3:  ;------------------------------------------------------------
 4:  putch:
 5:        pusha
 6:        mov    [temp_char],AL
 7:        mov    EAX,4             ; 4 = write
 8:        mov    EBX,1             ; 1 = std output (display)
 9:        mov    ECX,temp_char     ; pointer to char buffer
10:        mov    EDX,1             ; # bytes = 1
11:        int    0x80
12:        popa
13:        ret
```

Example 14.2 *Procedure to read a string.*

In this example, we look at the string read function `getstr`. We can read a string by using a single file read system call as shown in Program 14.2. Since we use the `dec` instruction, which modifies the flags register, we preserve its contents by saving and restoring the flags register using `pushf` (line 7) and `popf` (line 16). Since the file read system call returns the number of characters read in EAX, we can add this value (after decrementing) to the buffer pointer to append a NULL character (line 15). This returns the string in the NULL-terminated format.

Program 14.2 Procedure to read a string from the keyboard

```
 1:  ;-----------------------------------------------------------
 2:  ; Get string procedure receives input buffer pointer in EDI
 3:  ; and the buffer size in ESI.
 4:  ;-----------------------------------------------------------
 5:  getstr:
 6:        pusha
 7:        pushf
 8:        mov    EAX,3            ; file read service
 9:        mov    EBX,0            ; 0 = std input (keyboard)
10:        mov    ECX,EDI          ; pointer to input buffer
11:        mov    EDX,ESI          ; input buffer size
12:        int    0x80
13:        dec    EAX
14:  done_getstr:
15:        mov    byte[EDI+EAX],0  ; append NULL character
16:        popf
17:        popa
18:        ret
```

Example 14.3 *A file copy program.*

This example uses file copy to show how disk files can be manipulated using the file I/O system calls. The program requests the input and output file names (lines 27–31). It opens the input file in read-only mode using the open file system call (lines 33–39). If the call is successful, it returns the file descriptor (a positive integer) in EAX. In case of an error, a negative value is returned in EAX. This error check is done on line 41. If there is an error in opening the file, the program displays the error message and quits. Otherwise, it creates the output file (lines 47–53). A similar error check is done for the output file (lines 55–59).

File copy is done by reading a block of data from the input file and writing it to the output file. The block size is determined by the buffer size allocated for this purpose (see line 23).

The copy loop on lines 61–79 consists of three parts:

- Read a block of `BUF_SIZE` bytes from the input file (lines 62–67);
- Write the block to the output file (lines 69–74);
- Check to see if the end of file has been reached. As discussed before, this check is done by comparing the number of bytes read by the file-read system call (which is copied to EDX) to `BUF_SIZE`. If the number of bytes read is less than `BUF_SIZE`, we know we have reached the end of file (lines 76 and 77).

After completing the copying process, we close the two open files (lines 81–85).

Program 14.3 File copy program using the file I/O services

```
 1:  ;   A file copy program                            file_copy.asm
 2:  ;
 3:  ;        Objective: To copy a file using the int 0x80 services.
 4:  ;            Input: Requests names of the input and output files.
 5:  ;           Output: Creates a new output file and copies contents
 6:  ;                   of the input file.
 7:
 8:  %include "io.mac"
 9:
10:  %define   BUF_SIZE   256
11:
12:  .DATA
13:  in_fn_prompt     db  ’Please enter the input file name: ’,0
14:  out_fn_prompt    db  ’Please enter the output file name: ’,0
15:  in_file_err_msg  db  ’Input file open error.’,0
16:  out_file_err_msg db  ’Cannot create output file.’,0
17:
18:  .UDATA
19:  in_file_name     resb  30
20:  out_file_name    resb  30
21:  fd_in            resd  1
22:  fd_out           resd  1
23:  in_buf           resb  BUF_SIZE
24:
25:  .CODE
26:          .STARTUP
27:          PutStr  in_fn_prompt     ; request input file name
28:          GetStr  in_file_name,30  ; read input file name
29:
30:          PutStr  out_fn_prompt    ; request output file name
31:          GetStr  out_file_name,30 ; read output file name
```

```

        ;open the input file
        mov     EAX,5            ; file open
        mov     EBX,in_file_name ; pointer to input file name
        mov     ECX,0            ; file access bits (0 = read only)
        mov     EDX,0700         ; file permissions
        int     0x80
        mov     [fd_in],EAX      ; store fd for use in read routine

        cmp     EAX,0            ; open error if fd < 0
        jge     create_file
        PutStr  in_file_err_msg
        nwln
        jmp     done

create_file:
       ;create output file
        mov     EAX,8            ; file create
        mov     EBX,out_file_name; pointer to output file name
        mov     ECX,0700         ; read/write/exe by owner only
        int     0x80
        mov     [fd_out],EAX     ; store fd for use in write routine

        cmp     EAX,0            ; create error if fd < 0
        jge     repeat_read
        PutStr  out_file_err_msg
        nwln
        jmp     close_exit       ; close the input file & exit

repeat_read:
        ; read input file
        mov     EAX,3            ; file read
        mov     EBX,[fd_in]      ; file descriptor
        mov     ECX,in_buf       ; input buffer
        mov     EDX,BUF_SIZE     ; size
        int     0x80

        ; write to output file
        mov     EDX,EAX          ; byte count
        mov     EAX,4            ; file write
        mov     EBX,[fd_out]     ; file descriptor
        mov     ECX,in_buf       ; input buffer
        int     0x80

        cmp     EDX,BUF_SIZE     ; EDX = # bytes read
        jl      copy_done        ; EDX < BUF_SIZE
                                 ; indicates end-of-file
        jmp     repeat_read
copy_done:
        mov     EAX,6            ; close output file
        mov     EBX,[fd_out]
close_exit:
        mov     EAX,6            ; close input file
        mov     EBX,[fd_in]
done:
        .EXIT
```

14.8 Hardware Interrupts

We have seen how interrupts can be caused by the software instruction `int`. Since these instructions are placed in a program, software interrupts are called *synchronous* events. Hardware interrupts, on the other hand, are of hardware origin and *asynchronous* in nature. These interrupts are used by I/O devices such as the keyboard to get the processor's attention.

As discussed before, hardware interrupts can be further divided into either *maskable* or *nonmaskable* interrupts (see Figure 14.1). A nonmaskable interrupt (NMI) can be triggered by applying an electrical signal on the NMI pin of 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 cause this type of interrupt, an electrical signal should be applied to the INTR (INTerrupt Request) input of Pentium. Pentium recognizes the INTR interrupt only if the interrupt enable flag (IF) bit of the flags register is set to 1. Thus, these interrupts can be masked or disabled by clearing the IF bit. Note that we can use `sti` and `cli` to set and clear this bit in the flags register, respectively.

How Does the CPU Know the Interrupt Type?

Recall that every interrupt should be identified by a vector (a number between 0 and 255), which is used as an index into the interrupt vector table to obtain the corresponding ISR 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 processor initiates an interrupt acknowledge sequence. As part of this sequence, the processor sends out an interrupt acknowledge (INTA) signal, and the interrupting device is expected to place the interrupt vector on the data bus (see Figure 15.4 on page 447). The processor reads this value and uses it as the interrupt vector.

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 Pentium. While 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 processor.

When more than one device interrupts, we have to have a mechanism to prioritize these interrupts (if they come simultaneously) and forward only one interrupt request at a time to the processor while keeping the other interrupt requests pending for their turn. This mechanism can be implemented by using a special chip—the Intel 8259 Programmable Interrupt Controller. We give details of this chip in the next chapter (see Section 15.7.1 on page 446).

14.9 Summary

Interrupts provide a mechanism to transfer control to an interrupt service routine. The mechanism is similar to that of a procedure call. However, while procedures can be invoked only by a procedure call in software, interrupts can be invoked by both hardware and software.

Software interrupts are generated using the `int` instruction. Hardware interrupts are generated by I/O devices. These interrupts are used by I/O devices to interrupt the processor to service their requests.

Software interrupts are often used to support access to the system I/O devices. Linux provides a high-level interface to the hardware with software interrupts. We introduced Linux system calls and discussed how these calls can be used to access I/O devices. The system calls are invoked using `int 0x80`. We used several examples to illustrate the utility of these calls in reading from the keyboard, writing to the screen, and accessing files.

All interrupts, whether hardware-initiated 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. We briefly introduced hardware interrupts here; a more detailed discussion is provided in the next 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:

- Exceptions
- File descriptor
- File I/O
- File pointer
- Hardware interrupts
- Interrupt descriptors
- Linux system calls
- Protected-mode interrupt processing
- Software interrupts
- Taxonomy of interrupts

14.10 Exercises

14–1 What is the difference between a procedure and an interrupt service routine?

14–2 In invoking an interrupt service routine, 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.

14–3 How would you categorize the interrupts generated by the keyboard?

14–4 Explain how one can disable all maskable hardware interrupts.

14–5 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?

14–6 Is there any difference between how an ISR is invoked if the interrupt is caused by a software `int` instruction or hardware interrupt or exception?

14–7 What is a file descriptor?

14–8 What is the file pointer? How is it used to facilitate random access to a file?

14–9 How do you detect the end-of-file condition in a file read operation?

14.11 Programming Exercises

14–P1 Write a procedure using Linux system calls to write a string with semantics similar to the `PutStr` procedure we used to output a string. You should also write a main program to test this procedure.

14–P2 Write a procedure using Linux system calls to read a 16-bit integer with semantics similar to the `GetInt` procedure we used to read a number. You should also write a main program to test this procedure.

14–P3 Write a procedure to concatenate two files. This procedure takes two files names (i.e., pointers to files names strings) as parameters and appends contents of the second file to the first. You should also write a main program to test this procedure.

14–P4 Linux provides a system call to change the current working directory. Details about this system call are given below:

System call 12 — Change directory

Inputs:	EAX	=	12
	EBX	=	path
Returns:	EAX	=	0 if no error
Error:	EAX	=	error code

This function changes the current working directory to that given in path (a pointer to a character string like the file names). If the call is successful, it returns 0 in EAX. Otherwise, it returns an error code as in other system calls.

Write a procedure that takes a pointer to a path and changes the current directory to that path. You should also write a main program to test the procedure. The main program should output an OK message (something like "Directory exists." is fine) if the system call is successful. Otherwise, it should display an error message.

14–P5 Linux provides two system calls to create and remove a directory. Details about these system calls are given below:

System call 39 — Creates a directory

Inputs:	EAX	=	39
	EBX	=	path
	ECX	=	permissions
Returns:	EAX	=	0 if no error
Error:	EAX	=	error code

This function creates the directory given in the path (a pointer to a character string like the filenames). For details on permissions, see our discussion for the file-create system call. If the call is successful, it returns 0 in EAX. Otherwise, it returns an error code as in other system calls.

The system call to remove a directory is similar (it does not require permissions):

System call 40 — Removes a directory

Inputs:	EAX	=	40
	EBX	=	path
Returns:	EAX	=	0 if no error
Error:	EAX	=	error code

Write two procedures: one to create and the other to remove a directory. Each procedure takes a pointer to a path and creates the directory. (The create directory procedure uses default permissions.) You should also write a main program to test the procedure. Devise suitable error reporting mechanism (see the last exercise).

Chapter 15

Real-Mode Interrupts

Objectives

- To describe the real-mode interrupt processing by Pentium
- To discuss DOS and BIOS interrupt services
- To illustrate writing user defined interrupt service routines
- To provide an understanding of some peripheral support chips
- To discuss how programmed I/O is done using `in` and `out` instructions
- To give an example of interrupt-driven I/O

We discussed protected-mode interrupt processing in the last chapter. In this chapter, we use DOS to explore the real-mode interrupt processing. We begin the chapter with a description of the real-mode interrupt processing. Both DOS and BIOS provide several software interrupt services. Section 15.3 discusses the keyboard services of DOS and BIOS. The next section discusses the DOS services for text output on the display screen. Section 15.5 gives an example that explores single-stepping, a technique often used to debug programs. This section also gives details on how we can manipulate the trap flag.

I/O devices can be accessed in three ways. DOS and BIOS provide two ways of interacting with the system I/O devices. The third method involves direct I/O access. This method is low-level in nature and more complicated than the high-level access provided by DOS and BIOS. Direct access of I/O devices is supported by `in` *and* `out` *instructions. Sections 15.6 through 15.8 discuss this topic. Section 15.8 gives details on programmed I/O and interrupt-driven I/O. The last section summarizes the chapter.*

15.1 Interrupt Processing in the Real Mode

In the last chapter, we described the protected-mode interrupt processing. Briefly, there can be up to 256 different types of interrupts. Each interrupt type is identified by a number called a vector. This vector is used as an index into the Interrupt Descriptor Table (IDT). The IDT stores a descriptor for each interrupt type. Each descriptor, which is eight bytes long, is essentially a pointer to the interrupt service routine (ISR). Each descriptor stores a 16-bit segment selector, a 32-bit offset to the ISR, and so on.

In the real mode, the Pentium follows the interrupt mechanism used by the 8086 processor. In this mode, the IDT is located at base address 0. Each vector takes only four bytes as opposed to eight bytes in the protected mode. Each vector consists of a CS:IP pointer to the associated ISR: two bytes for specifying the code segment (CS), and two bytes for the offset (IP) within the code segment. Figure 15.1 shows the interrupt vector table layout in the memory.

Since each entry in the interrupt vector table is four bytes long, the interrupt type is multiplied by 4 to get the corresponding ISR pointer in the table. For example, `int 2` can find its ISR pointer at memory address 2×4 = 00008H. The first two bytes at the specified address are taken as the offset value and the next two bytes as the CS value.

When a real-mode interrupt occurs, the following actions are taken:

1. Push flags register onto the stack;
2. Clear interrupt and trap flags to disable further interrupts;
3. Push CS and IP registers onto the stack;
4. Load CS with the 16-bit data at memory address (interrupt-type $*$ 4 + 2);
5. Load IP with the 16-bit data at memory address (interrupt-type $*$ 4).

From our discussion in the last chapter, it is clear that these actions are similar to the steps taken in the protected mode with appropriate adjustment for the 16-bit mode. Likewise, the actions taken on `iret` are very similar and are given below:

1. Pop the 16-bit value on top of the stack into IP register;
2. Pop the 16-bit value on top of the stack into CS register;
3. Pop the 16-bit value on top of the stack into the flags register.

A typical ISR structure is shown below.

```
<save the registers used in the ISR>
sti     ; enable further interrupts
. . .
<ISR body>
. . .
<restore the saved registers>
iret    ; return to the interrupted program
```

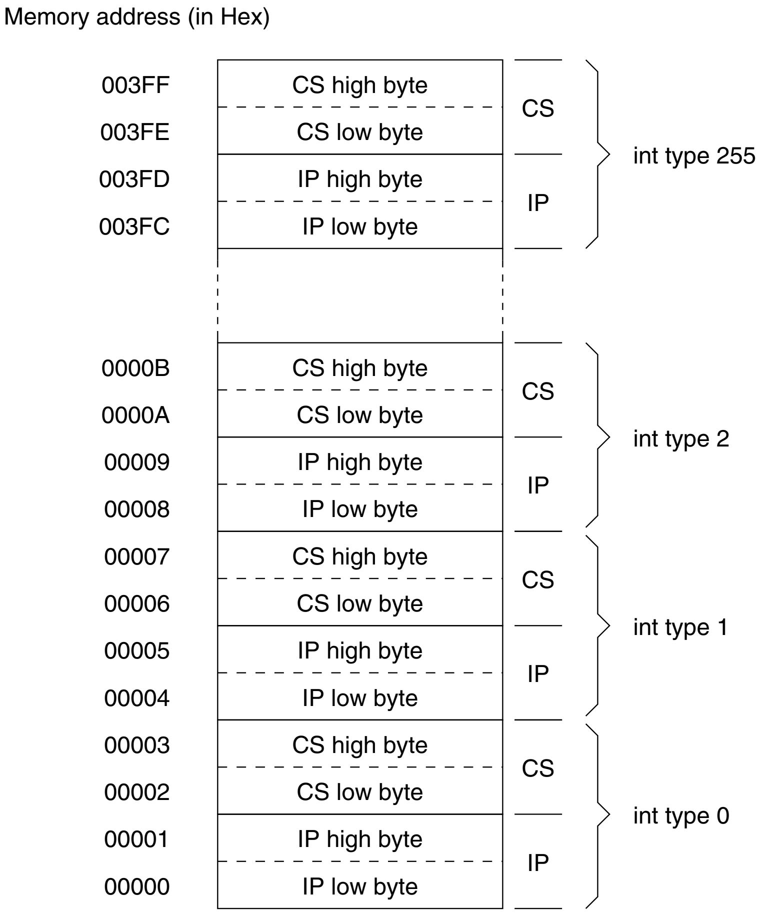


Figure 15.1 Real-mode interrupt vector table.

15.2 Software Interrupts

As discussed in Section 14.5, software interrupts are initiated by executing the interrupt instruction. All DOS services are provided by `int 21H`. DOS provides more than 80 different services (called functions). We discuss some of these services in the next couple of sections.

Both DOS and BIOS provide several interrupt service routines to access I/O devices. The following sections consider a select few of these services and explain by means of examples how they can be used. We organize our discussion around the two I/O devices: the keyboard and the display.

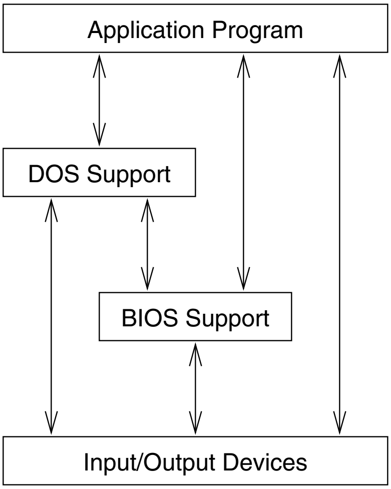


Figure 15.2 Various ways of interacting with I/O devices.

The interrupt services provided by DOS and BIOS are not mutually exclusive. Some services, such as reading a character from the keyboard, are provided by both DOS and BIOS. In fact, DOS uses BIOS-supported routines to provide some services that control the system hardware (see Figure 15.2). We can also access I/O devices directly using input and output instructions. We discuss this topic in Sections 15.6 through 15.8.

15.3 Keyboard Services

This section starts with a brief description of the keyboard interface. Then we discuss how the DOS and BIOS services can be used to read input from the keyboard.

15.3.1 Keyboard Description

Associated with each I/O device is a *device controller* or *I/O controller* that acts as hardware interface between the processor and the I/O device. The device controller performs many of the low-level tasks specific to the I/O device. This allows the processor to interact with the device at a higher level. For each device controller, there is a software interface that provides a clean interface to access the device. This interface is called the *device driver*.

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. This reporting is done via the 8259 interrupt controller, which in turn interrupts the processor to service the keyboard. We provide details on this interrupt controller later (see Section 15.7.1).

On your PC, every time a key is depressed or released, the 8259 interrupt controller generates a hardware interrupt (`int 9`). This interrupt is serviced by BIOS. The ISR for the keyboard interrupt reads the identity of the key and stores it in the type-ahead keyboard buffer. In addition, it also identifies special key combinations such as `ctrl-break`.

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 righthand 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 15.1 shows the scan codes for the PC keyboard.

The scan code of a key does not have any relation to the ASCII code of the corresponding character. The `int 9` ISR receives the scan code and generates the equivalent ASCII code, if there is one. The code is placed in the keyboard buffer. This buffer is organized as a queue, which is a first-in–first-out (FIFO) data structure. When a request is received to read a keyboard character, the oldest key in the buffer (the earliest key in the buffer) is supplied and it is removed from the buffer.

15.3.2 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. DOS provides the following seven functions to interact with the keyboard—reading a character or getting the status of the keyboard buffer.

Function 01H — Keyboard input with echo

Input:	AH	=	01H
Returns:	AL	=	ASCII code of the key

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, which aborts 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.

Table 15.1 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			

Subfunction — Keyboard input

Inputs: AH = 06H
DL = FFH
Returns: ZF = 0 if a character is available
In this case, the key ASCII code is in AL
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 Section 15.3.3).

Function 08H — Keyboard input without echo

Input: AH = 08H
Returns: AL = ASCII code of the key entered

This function provides similar services as 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.

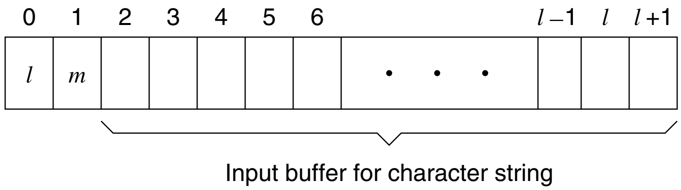
Function 0AH — Buffered keyboard input

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 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 are rejected, and this is indicated by a beep.



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 type-ahead 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.

15.3.3 Extended Keyboard Keys

The PC keyboard has several keys that are not part of the standard ASCII character set. These keys include the function keys, cursor arrows, Home, End, etc. 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 15.1 lists the keyboard scan codes for the extended keys. In contrast, when an ASCII key is pressed, the first byte in the keyboard buffer is the ASCII code of the key, and the second byte is the scan code of the key.

To read a character from the keyboard using the 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 $\neq$ 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 15.1 *A get string example.*

In this example, we look at the DOS version of the `GetStr` procedure to read a string from the keyboard. The program listing is given in Program 15.1. The main program prompts the user for maximum string length and reads this value into CX (lines 22 and 23). This value should be at least 1; if not, `read_string` procedure reports an error. After prompting for the input string, it calls the `read_string` procedure to read the string.

The `read_string` procedure is loosely based on the Linux `getstr` procedure discussed in the last chapter (see page 415). This procedure expects a pointer to a buffer to store the input string in BX and the buffer length in CX. This procedure reads a string from the keyboard using the buffered keyboard input function 0AH. The procedure, given in Program 15.1, follows the pseudocode shown below:

```
read_string()
    save registers used in the procedure
    if (CX < 2)
    then
        Display error message
        return
    end if
    if (CX > 81)
    then
        CX := 81
    end if
    use function 0AH to read input string into
        temporary buffer temp_buf
    copy input string from temp_buf to
        user buffer and append NULL
    restore registers
    return
end read_string
```

Note that the actual string buffer size is 81 bytes. The `temp_buf` uses two more bytes as we use the services of function 0AH. These two additional bytes are used for l and m values as shown on page 430.

Program 15.1 Procedure to read a string from the keyboard

```
;A string read program                        GETSTR.ASM
;        Objective: To demonstrate the use of DOS keyboard
;                   functions.
;            Input: Prompts for a string.
;           Output: Displays the input string.

STR_LENGTH  EQU  81
%include "io.mac"
.STACK   100H
.DATA
prompt_msg1  db  "Please enter maximum string length: ",0
prompt_msg2  db  "Please enter a string: ",0
string_msg   db  "The string entered is: ",0
error_msg    db  "No string read. Buffer size must be at least 1.",0
```

```

.UDATA
temp_buf     resb  STR_LENGTH+2
in_string    resb  STR_LENGTH

.CODE
        .STARTUP
        PutStr  prompt_msg1
        GetInt  CX              ; max. string length
        nwln
        PutStr  prompt_msg2
        mov     BX,in_string    ; BX = pinter to input buffer
        call    read_string     ; to call read_string procedure
        nwln
        PutStr  string_msg
        PutStr  in_string
        nwln
        .EXIT

;-----------------------------------------------------------
; Get string (of maximum length 80) from keyboard.
;     BX <-- pointer to a buffer to store the input string
;     CX <-- buffer size = string length + 1 for NULL
; If CX <2, reports error and terminates.
; If CX > 81, CX = 81 is used to read at most 80 characters.
;-----------------------------------------------------------
read_string:
        pusha
        ; ES = DS for use by the string instruction--movsb
        mov     DX,DS
        mov     ES,DX
        mov     DI,BX           ; DI = buffer pointer
        inc     CX              ; space for NULL
        ; check CX value
        cmp     CX,2
        jl      bailout
        cmp     CX,81
        jle     read_str
        mov     CX,81
read_str:
        ; use temporary buffer temp_buf to read the string
        ; using functin 0AH of int  21H
        mov     DX,temp_buf
        mov     SI,DX
        mov     [SI],CL         ; first byte = # chars. to read
        mov     AH,0AH
        int     21H
        inc     SI              ; second byte = # chars. read
        mov     CL,[SI]         ; CX = # bytes to copy
        inc     SI              ; SI = input string first char.
        cld                     ; forward direction for copy
        rep     movsb
        mov     byte[DI],0      ; append NULL
        jmp     done
bailout:
        nwln
        PutStr  error_msg
done:
        popa
        ret
```

15.3.4 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 `int 16H`. One difference between DOS and BIOS functions is that if you use the DOS services, the keyboard input can be redirected.

Function 00H — Read a character from the keyboard

Input:	AH	=	00H
Returns:	if AL	$\neq$	0 then
	AL	=	ASCII code of the key entered
	AH	=	Scan code of the key entered
	if AL	=	0
	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

Since 00H represents NULL in ASCII, returning the NULL ASCII code 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 `2` 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 431.

```
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
```

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. Returns ASCII in AL and scan code in AH. Does not remove them 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 as the DOS function 0BH (see page 430). Unlike the DOS function, the zero flag (ZF) is used to indicate whether or not the keyboard buffer is empty. If a character is available in the buffer, its ASCII and scan codes are copied to the AL and AH registers as if we performed the function 00H. One major difference is that it does not actually remove the key codes from the keyboard buffer. Thus, it allows you to look ahead at the next character without actually reading it from the buffer.

Function 02H — Check keyboard status

Input:	AH	=	02H
Returns:	AL	=	status of the shift and toggle keys

The bit assignment is shown in the following table. 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, Alt) and four toggle switches (Scroll lock, Number lock, Caps lock, and Ins).

Table 15.2 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

Example 15.2 *A special string read 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 could be terminated either by pressing both the 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

```
read_string()
    get maximum string length STRING_LENGTH 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 = STRING_LENGTH)
end_read:
```

```
          append NULL character to string input
          compute 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 almost never detects the condition that both shift keys are depressed (with the exception noted before).
To correct this problem, we have to modify the procedure as follows:

```
read_string()
    get maximum string length STRING_LENGTH 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 = STRING_LENGTH)
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 the 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 15.2 A program to read a string from the keyboard using the BIOS services

```
;A string read program       FUNNYSTR.ASM
;        Objective: To demonstrate the use of BIOS keyboard
;                   functions 0, 1, and 2.
;            Input: Prompts for a string.
;           Output: Displays the input string and its length.

STR_LENGTH  EQU  81
%include "io.mac"
.STACK   100H
.DATA
prompt_msg  db  "Please enter a string (< 81 chars): ",0
string_msg  db  "The string entered is ",0
length_msg  db  " with a length of ",0
end_msg     db  " characters.",0

.UDATA
string      resb  STR_LENGTH

.CODE
        .STARTUP
        PutStr  prompt_msg
        mov     AX,STR_LENGTH-1
        push    AX              ; push max. string length
        mov     AX,string
        push    AX              ; and string pointer parameters
        call    read_string     ; to call read_string procedure
        nwln
        PutStr  string_msg
        PutStr  string
        PutStr  length_msg
        PutInt  AX
        PutStr  end_msg
        nwln
        .EXIT

;-----------------------------------------------------------
; String read procedure using BIOS int 16H. Receives string
; pointer and the length via the stack. Length of the string
; is returned in AX.
;-----------------------------------------------------------
.CODE
read_string:
        push    BP
        mov     BP,SP
        push    BX
        push    CX
        mov     CX,[BP+6]    ; CX = length
        mov     BX,[BP+4]    ; BX = string pointer
read_loop:
        mov     AH,2         ; read keyboard status
        int     16H          ; status returned in AL
        and     AL,3         ; mask off most significant 6 bits
        cmp     AL,3         ; if equal both shift keys depressed
        jz      end_read
        mov     AH,1         ; otherwise, see if a key has been
        int     16H          ; struck
        jnz     read_key     ; if so, read the key
        jmp     read_loop
read_key:
        mov     AH,0         ; read the next key from keyboard
        int     16H          ; key returned in AL
        mov     [BX],AL      ; copy to buffer and increment
        inc     BX           ;  buffer pointer
        PutCh   AL           ; display the character
        loop    read_loop
end_read:
        mov     [BX],byte 0  ; append NULL
        sub     BX,[BP+4]    ; find the input string length
        mov     AX,BX        ; return string length in AX
        pop     CX
        pop     BX
        pop     BP
        ret     4
```

15.4 Text Output to Display Screen

DOS provides three functions to display characters on the screen. BIOS provides many more services to interact with the screen. Here we discuss the three DOS functions to display text on the screen. Two of these functions display a single character, while the third function displays a string terminated by $.

Function 02H — Display a character on the screen

Inputs:	AH	=	02H
	DL	=	ASCII code of the character to be displayed
Returns:	Nothing		

This service displays the character in DL on the screen at the current cursor position and advances the cursor. Special ASCII characters such as Backspace (08H), Carriage return (0DH), Line feed (0AH), Bell (07H), etc. are recognized as control characters and properly processed. A pending `ctrl-break` will be processed after the character is displayed.

Function 06H — Direct console I/O
This function, discussed on page 427, provides both keyboard input and display output services. A character code other than FFH in DL causes the character in DL to be displayed.

Function 09H — Display a string of characters

Inputs:	AH	=	09H
	DS:DX	=	pointer to a character string to be displayed. The string must be terminated by `$`.

This function is useful in displaying a $-terminated character string. The dollar sign is used to indicate the end of the string and is not displayed.

Example 15.3 *A procedure to display newline.*
The `nwln` macro defined in `io.mac` can be used to send a carriage-return (CR) and line feed (LF) pair to the screen. The macro simply calls the `proc_nwln` procedure, which uses DOS function 2 to display CR and LF. The code for this procedure is shown in Program 15.3. It uses the following constants:

```
CR     EQU   13   ; carriage return
LF     EQU   10   ; linefeed
```

It uses the `DOScall` macro, which is defined as follows:

```
%macro  DOScall 1
      mov    AH,%1
      int    0x21
%endmacro
```

Program 15.3 A procedure to send a newline to the screen

```
;-----------------------------------------------------------
; Sends CR and LF to the screen. Uses display function 2.
;-----------------------------------------------------------
proc_nwln:
    push     AX
    push     DX
    mov      DL,CR        ; carriage return
    DOScall  2
    mov      DL,LF        ; line feed
    DOScall  2
    pop      DX
    pop      AX
    ret
```

15.5 Exceptions: An Example

We discussed exceptions in Section 14.4 on page 407. As an example of an exception, we write an ISR to single-step a piece of code (let us call it *single-step code*). Recall that we enter the single-step mode when the trap flag is set to 1. In this example, during single-stepping, we display the contents of the AX and BX registers after the execution of each instruction of the single-step code. The objectives in writing this program are to demonstrate how ISRs can be defined and installed and to show how TF can be manipulated.

To put the processor in the single-step mode, we have to set TF. Since there are no instructions to manipulate TF directly, we have to use an indirect means: first 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 41–45 of Program 15.4 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
```

as shown on line 56. 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 ISR

Function 25H — Set interrupt vector

Inputs: AH = 25H
AL = interrupt type number
DS:DX = address of the ISR

Returns: Nothing

The remainder of the code is straightforward:

Lines 26–29: We use function 35H of DOS interrupt (`int 21H`) to get the current vector value of `int 1`. This vector value is restored before exiting the program.

Lines 32–38: The vector of our ISR is installed by using function 25H of `int 21H`.

Lines 61–67: The original `int 1` vector is restored using function 25H of `int 21H`.

Program 15.4 An example to illustrate the installation of a user-defined ISR

```
 1:  ;Single-step program      STEPINTR.ASM
 2:  ;
 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:  %include "io.mac"
10:  .STACK 100H
11:  .DATA
12:  start_msg   db  "Starts single-stepping process.",0
13:  AXequ       db  "AX = ",0
14:  BXequ       db  " BX = ",0
15:
16:  .UDATA
17:  old_offset  resw  1    ; for old ISR offset
18:  old_seg     resw  1    ;   and segment values
19:
20:  .CODE
21:          .STARTUP
22:          PutStr  start_msg
23:          nwln
24:
25:          ; get current interrupt vector for int 1H
26:          mov     AX,3501H         ; AH = 35H and AL = 01H
27:          int     21H              ; returns the offset in BX
28:          mov     [old_offset],BX  ;   and the segment in ES
29:          mov     [old_seg],ES
```

```

        ;set up interrupt vector to our ISR
        push    DS           ; DS is used by function 25H
        mov     AX,CS        ; copy current segment to DS
        mov     DS,AX
        mov     DX,sstep_ISR ; ISR offset in DX
        mov     AX,2501H     ; AH = 25H and AL = 1H
        int     21H
        pop     DS           ; restore DS

        ; set trap flag to start single-stepping
        pushf
        pop     AX           ; copy flags into AX
        or      AX,100H      ; set trap flag bit (TF = 1)
        push    AX           ; copy modified flag bits
        popf                 ;   back to flags register

        ; from now on int 1 is generated after executing
        ;  each instruction. Some test instructions follow.
        mov     AX,100
        mov     BX,20
        add     AX,BX

        ; clear trap flag to end single-stepping
        pushf
        pop     AX           ; copy flags into AX
        and     AX,0FEFFH    ; clear trap flag bit (TF = 0)
        push    AX           ; copy modified flag bits
        popf                 ;   back to flags register

        ; restore the original ISR
        mov     DX,[old_offset]
        push    DS
        mov     AX,[old_seg]
        mov     DS,AX
        mov     AX,2501H
        int     21H
        pop     DS

        .EXIT
;------------------------------------------------------------
;Single-step interrupt service routine replaces int 01H.
;------------------------------------------------------------
.CODE
```

```
sstep_ISR:
        sti                     ; enable interrupt
        PutStr  AXequ           ; display AX contents
        PutInt  AX
        PutStr  BXequ           ; display BX contents
        PutInt  BX
        nwln
        iret
```

15.6 Direct Control of I/O Devices

Figure 15.2 on page 426 shows three ways an application program can interact with I/O devices. Our emphasis thus far has been on using either DOS or BIOS support routines to access I/O devices. When we want to access an I/O device for which there is no such support available from either DOS or BIOS, or when we want a nonstandard access, we have to access these devices directly—the third method shown in Figure 15.2.

At this point, it is useful to review the material presented in Chapter 2. As described in Chapter 2, Pentium uses a separate I/O address space of 64K. This address space can be used for 8-bit, 16-bit, or 32-bit I/O ports. However, the combination cannot be more than the total I/O space. For example, we can have 64K 8-bit ports, 32K 16-bit ports, 16K 32-bit ports, or a combination of these that fits the 64K I/O address space. Devices that transfer data 8 bits at a time can use 8-bit ports. These devices are called 8-bit devices. An 8-bit device can be located anywhere in the I/O space without any restrictions. On the other hand, a 16-bit port should be aligned to an even address so that 16 bits can be simultaneously transferred in a single bus cycle. Similarly, 32-bit ports should be aligned at addresses that are multiples of four. Pentium, however, supports unaligned I/O ports, but there is a performance penalty (see Section 2.7 on page 41 for a related discussion).

15.6.1 Accessing I/O Ports

To facilitate access to the I/O ports, the instruction set provides two types of instructions: register I/O instructions and block I/O instructions. Register I/O instructions are used to transfer data 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

There are 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 are directly addressable—address is given as part of the instruction.

Both instructions can be used to operate on 8-, 16-, or 32-bit data. Each instruction can take one of two forms, depending on whether a port is directly addressable or not. The general format of the `in` instruction is

```
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 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 format for the `out` instruction is

```
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 instruction set has 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 10. For this reason, block I/O instructions are also called string I/O instructions. Like 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 pointed to by ES:(E)DI. The address size determines whether the DI or EDI register is used (see Chapter 2 for details). Block I/O instructions do not allow the direct addressing format.

For the `outs` instruction, the memory address should be pointed by 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.

You can use the `rep` prefix with `ins` and `outs` instructions. However, you cannot use the other two prefixes—`repe` and `repne`—with the block I/O instructions. The semantics of `rep` are the same as those in the string instructions. The directions flag (DF) determines whether the index register in the block I/O instruction is decremented (DF is 1) or incremented (DF is 0). The increment or 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. Use `insb` and `outsb` for byte transfers, `insw` and `outsw` for word transfers, and `insd` and `outsd` for doubleword transfers.

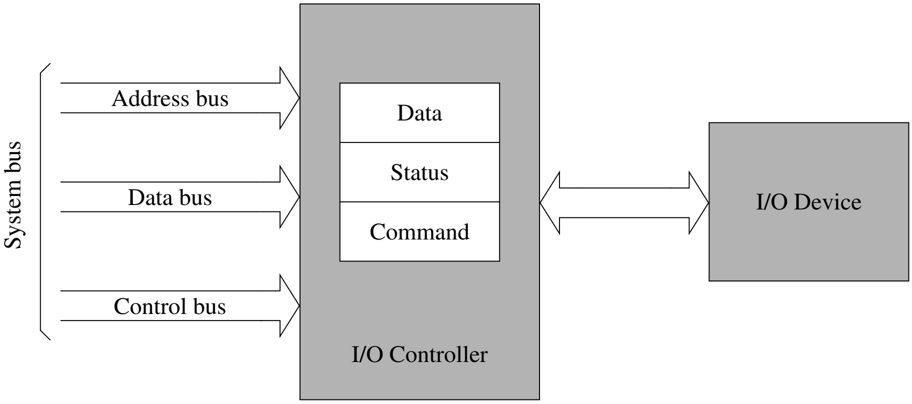


Figure 15.3 Input/output device interface to the system.

15.7 Peripheral Support Chips

Recall from Chapter 2 that I/O devices are not interfaced directly to the processor. Rather, each device has a peripheral controller that acts as an intermediary between the device and the processor, as shown in Figure 15.3.

In this section, we start our discussion by explaining how multiple devices can interrupt the processor using the 8259 programmable interrupt controller chip. Then, we proceed to describe the 8255 programmable peripheral interface chip.

15.7.1 8259 Programmable Interrupt Controller

We can use the 8259 programmable interrupt controller (PIC) to accommodate more than one interrupting device in the system. 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 15.4.

Internally, 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–IRQ7 requests in several ways. The BIOS initializes the 8259 to assign fixed priorities—the default mode called 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. Table 15.3 shows the mapping of the 8259 IRQ inputs to various devices in the system.

Also part of this initialization is the assignment of interrupt type numbers. To do this, only the lowest type number should be specified. BIOS uses 08H as the lowest interrupt type (for the request coming on the IRQ0 line). The 8259 automatically assigns the next

Table 15.3 Mapping of I/O Devices to External Interrupt Levels

IRQ #	Interrupt type	Device
0	08H	System timer
1	09H	Keyboard
2	0AH	Reserved
3	0BH	Serial port (COM1)
4	0CH	Serial port (COM2)
5	0DH	Hard disk
6	0EH	Floppy disk
7	0FH	Printer

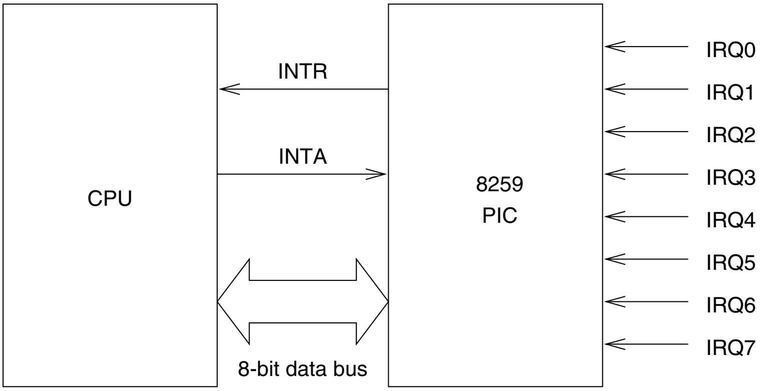


Figure 15.4 Intel 8259 programmable interrupt controller.

seven numbers to the remaining seven IRQ lines in increasing order, with IRQ7 generating an interrupt of type 0FH.

As discussed in Section 14.8, the interrupt controller raises the interrupt signal on the INTR input of the processor to cause a hardware interrupt. In response to this signal, the processor sends out an interrupt acknowledge signal on INTA line. When the 8259 receives this signal, it places the interrupt vector on the 8-bit data bus. The processor reads this value and uses it as the interrupt vector.

All communication between the processor and the 8259 occurs via the data bus. The 8259 PIC is an 8-bit device requiring two ports for ICR and IMR. These are mapped to the I/O address space, as shown in Table 15.4.

Table 15.4 8259 Port Address Mapping

8259 register	Port address
ICR	20H
IMR	21H

Note that the processor recognizes external interrupt requests generated by 8259 only if the IF flag is set. Thus, by clearing the IF flag, we can mask or disable all eight external interrupts as a group. However, to selectively disable external interrupts, we have to use IMR. Each bit in 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

```
mov     AL,0FEH
out     21H,AL
```

to disable all external interrupts except the system timer interrupt request 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 processor, as it has a higher priority than the keyboard interrupt. Once the timer ISR is completed, the 8259 forwards the keyboard interrupt to the processor for processing. To facilitate this, the 8259 should know when an ISR is completed. The end of an ISR execution is signalled 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 the 8259 PIC. This code fragment appears before the `iret` instruction of an ISR.

15.7.2 8255 Programmable Peripheral Interface Chip

The 8255 programmable peripheral interface (PPI) chip provides three 8-bit general-purpose registers that can be used to interface with I/O devices. These three registers—called PA, PB, and PC—are mapped to the I/O space as shown in Table 15.5.

The BIOS configures these three ports as shown in the above table. Here input and output are from the processor viewpoint. For our discussion, we need to know details only about PA and PB ports. These details are given in Table 15.6.

The keyboard interface is provided by port PA and PB7. The keyboard sends an interrupt to 8259 (on the IRQ1 line) whenever there is a change in the state of a key. The scan code of the key whose state has changed (i.e., depressed or released) is provided by the keyboard at PA. The keyboard then waits for an acknowledge signal to know that the scan code has been

Table 15.5 8255 Port Address Mapping

8255 register	port address
PA (input port)	60H
PB (output port)	61H
PC (input port)	62H
Command register	63H

Table 15.6 I/O Bit Map of Ports PA and PB of 8255

PA	Keyboard scan code if PB7 = 0 PA7 = 0 if a key is depressed PA7 = 1 if a key is released PA0–PA6 = key scan code Configuration switch 1 if PB7 = 1
PB	PB7 — selects source for PA input 0 — keyboard scan code 1 — configuration switch 1 Also, 1 is used as keyboard acknowledge

read by the processor. This acknowledgment can be signalled by setting and clearing PB7 momentarily. The normal state of PB7 is 0.

The scan code of the key can be read from PA. Bits PA0–PA6 give the scan code of the key whose state has changed. PA7 is used to indicate the current state of the key.

PA7 = 0 — key is depressed
PA7 = 1 — key is released

For example, if the `Esc` is pressed, PA supplies 01H as 1 is the scan code for the `Esc` key. When Esc is released, PA supplies 81H. In the next section, we write our own keyboard driver to illustrate the keyboard interface.

15.8 I/O Data Transfer

We discussed various ways of accessing the I/O devices. Let's now look at how we can transfer data between the system and an I/O device. There are three basic techniques: programmed I/O, interrupt-driven I/O, and direct memory access (DMA).

Programmed I/O involves the processor in the I/O data transfer. The processor repeatedly checks to see if a particular condition is true. Typically, it busy-waits until the condition is true. For example, if we are interested in reading a key, the processor repeatedly checks to see if a key pressed. Once a key is pressed, it gets the ASCII value of the key and busy-waits for another key. This process is called *polling*. From this brief description, it should be clear that the programmed I/O mechanism wastes processor time.

In interrupt-driven I/O, the processor will be interrupted when the specific event (key depression in our example) occurs. Obviously, this is a better way of using the processor. However, 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 transfers. Typically, a DMA controller oversees the data transfers. When the specified transfer is complete, the processor is notified by an interrupt signal. More details on this technique are available in computer architecture books. In the remainder of this section, we give examples for the other two techniques.

15.8.1 Programmed I/O

The heart of programmed I/O is a busy-wait loop. 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 15.5 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. The `and` statement on line 38 masks off 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 translation table (`lcase_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 `0CH` (lines 56 to 57). The rest of the program is straightforward to follow.

Program 15.5 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:
```

```
ESC_KEY      EQU  1BH   ; ASCII code for ESC key
KB_DATA      EQU  60H   ; 8255 port PA

.MODEL SMALL
.STACK 100H
.DATA
prompt_msg     db  ’Press a key. ESC key terminates the program.’,0
; lowercase scan code to ASCII conversion table.
; ASCII code 0 is used for scan codes in which we are not interested.
lcase_table    db  01BH,’1234567890-=’,08H,09H
               db  ’qwertyuiop[]’,0DH,0
               db  ’asdfghjkl;’,27H,60H,0,’\’
               db  ’zxcvbnm,./’,0,’*’,0,’ ’,0
               db  0,0,0,0,0,0,0,0,0,0
               db  0,0,0,0,0,0,0,0,0,0
               db  0,0,0,0,0,0,0,0,0,0
.CODE
INCLUDE io.mac

main    PROC
        .STARTUP
        PutStr  prompt_msg
        nwln
key_up_loop:
        ; Loops until a key is pressed i.e., until PA7 = 0.
        ; PA7 = 1 if a key is up.
        in      AL,KB_DATA   ; read keyboard status & scan code
        test    AL,80H       ; PA7 = 0?
        jnz     key_up_loop  ; if not, loop back

        and     AL,7FH       ; isolate the scan code
        mov     BX,OFFSET lcase_table
        dec     AL           ; index is one less than scan code
        xlat
        cmp     AL,0         ; ASCII code of 0 => uninterested key
        je      key_down_loop
        cmp     AL,ESC_KEY   ; ESC key---terminate program
        je      done
display_ch:
        putch   AL
        putch   ’ ’

key_down_loop:
        in      AL,KB_DATA    ; read keyboard status & scan code
        test    AL,80H        ; PA7 = 1?
        jz      key_down_loop ; if not, loop back

        ; clear keyboard buffer
        mov     AX,0C00H
        int     21H

        jmp     key_up_loop
Done:
        ; clear keyboard buffer
        mov     AX,0C00H
        int     21H

        .EXIT
main    ENDP
        END   main
```

15.8.2 Interrupt-driven I/O

We discussed hardware interrupts in Chapter 14. In this section, we illustrate how a hardware interrupt mechanism can be used to perform interrupt-driven I/O. For our example, we write a type 9 interrupt routine that replaces the BIOS supplied routine. Recall that a type 9 interrupt is generated via the IRQ1 line 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 “ISR 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 = 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

```
read_kb_key()
    while (keyboard_data = –1)
    end while
    AL := keyboard_data
    keyboard_data := –1
    return
end read_kb_key
```

The keyboard ISR `kbrd_ISR` is invoked whenever a key is pressed or released. The scan code of the key can be read from PA0–PA6, while the key state can be read from PA7. PA7 is 0 if the key is depressed; PA7 is 1 if the key is released. The program listing is given in Program 15.6.

After reading the key scan code (lines 107 and 108), the keyboard should be acknowledged. This is done by momentarily setting and clearing the PB7 bit (lines 111–116). 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 sends an end-of-interrupt (EOI) notification 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 15.6 A keyboard ISR to replace the BIOS keyboard handler

```
;Keyboard interrupt service program             KEYBOARD.ASM
;
;        Objective: To demonstrate how the keyboard works.
;            Input: Key strokes from the keyboard. Only left- and
;                   right-shift keys are recognized.
;                   ESC key restores the original keyboard ISR
;                   and terminates the program.
;           Output: Displays the key on the screen.

ESC_KEY      EQU  1BH   ; ASCII code for ESC key
CR           EQU  0DH   ; ASCII code for carriage return
KB_DATA      EQU  60H   ; 8255 port PA
KB_CTRL      EQU  61H   ; 8255 port PB
LEFT_SHIFT   EQU  2AH   ; left-shift scan code
RIGHT_SHIFT  EQU  36H   ; right-shift scan code
EOI          EQU  20H   ; end-of-interrupt byte for 8259 PIC
PIC_CMD_PORT EQU  20H   ; 8259 PIC command port

%include "io.mac"
.STACK 100H
.DATA
install_msg    db  "New keyboard ISR installed.",0
keyboard_data  db  -1   ; keyboard buffer
keyboard_flag  db  0    ; keyboard shift status
; lowercase scan code to ASCII conversion table.
; ASCII code 0 is used for unnecessary scan codes.
lcase_table    db  01BH,"1234567890-=",08H,09H
               db  "qwertyuiop[]",CR,0
               db  "asdfghjkl;",27H,60H,0,’\’
               db  "zxcvbnm,./",0,’*’,0,’ ’,0
               db  0,0,0,0,0,0,0,0,0,0
               db  0,0,0,0,0,0,0,0,0,0
               db  0,0,0,0,0,0,0,0,0,0
; uppercase scan code to ASCII conversion table.
ucase_table    db  01BH,"!@#$%^&*()_+",08H,09H
               db  "QWERTYUIOP{}",0DH,0
               db  "ASDFGHJKL:",’"’,’~’,0,’|’
               db  "ZXCVBNM<>?",0,’*’,0,’ ’
               db  0,0,0,0,0,0,0,0,0,0
               db  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

.UDATA
old_offset     resw  1    ; storage for old int 09H vector
```

```
old_segment    resw  1

.CODE
        .STARTUP
        PutStr  install_msg
        nwln

        ; save int 09H vector for later restoration
        mov     AX,3509H     ; AH = 35H and AL = 09H
        int     21H          ; DOS function 35H returns
        mov     [old_offset],BX   ; offset in BX and
        mov     [old_segment],ES  ; segment in ES

        ;set up interrupt vector to our keyboard ISR
        push    DS           ; DS is used by function 25H
        mov     AX,CS        ; copy current segment to DS
        mov     DS,AX
        mov     DX,kbrd_ISR  ; ISR offset in DX
        mov     AX,2509H     ; AH = 25H and AL = 09H
        int     21H
        pop     DS           ; restore DS

repeat1:
        call    read_kb_key  ; read a key
        cmp     AL,ESC_KEY   ; if ESC key
        je      done         ; then done
        cmp     AL,CR        ; if carriage return
        je      newline      ; then display new line
        PutCh   AL           ; else display character
        jmp     repeat1
newline:
        nwln
        jmp     repeat1
done:
        ; restore original keyboard interrupt int 09H vector
        mov     DX,old_offset
        push    DS
        mov     AX,old_segment
        mov     DS,AX
        mov     AX,2509H
        int     21H
        pop     DS

        .EXIT
```

```
;-----------------------------------------------------------
;This procedure waits until a valid key is entered at the
; keyboard. The ASCII value of the key is returned in AL.
;-----------------------------------------------------------
.CODE
read_kb_key:
        cmp     byte [keyboard_data],-1  ; -1 is an invalid entry
        je      read_kb_key
        mov     AL,[keyboard_data]
        mov     byte [keyboard_data],-1
        ret
;-----------------------------------------------------------
;This keyboard ISR replaces the original int 09H ISR.
;-----------------------------------------------------------
.CODE
kbrd_ISR:
        sti                  ; enable interrupt
        push    AX           ; save registers used by ISR
        push    BX
        in      AL,KB_DATA   ; read keyboard scan code and the
        mov     BL,AL        ;  key status (down or released)
        ; send keyboard acknowledge signal by momentarily
        ;  setting and clearing PB7 bit
        in      AL,KB_CTRL
        mov     AH,AL
        or      AL,80H
        out     KB_CTRL,AL   ; set PB7 bit
        xchg    AL,AH
        out     KB_CTRL,AL   ; clear PB7 bit

        mov     AL,BL        ; AL = scan code + key status
        and     BL,7FH       ; isolate scan code
        cmp     BL,LEFT_SHIFT   ; left- or right-shift key
        je      left_shift_key  ;  changed status?
        cmp     BL,RIGHT_SHIFT
        je      right_shift_key
        test    AL,80H          ; if not, check status bit
        jnz     EOI_to_8259     ; if key released, do nothing
        mov     AH,[keyboard_flag] ; AH = shift key status
        and     AH,1         ; AH = 1 if left/right shift is ON
        jnz     shift_key_on
        ; no shift key is pressed
        mov     BX,lcase_table ; shift OFF, use lowercase
        jmp     SHORT get_ASCII       ;  conversion table
```

```
shift_key_on:
        mov     BX,ucase_table ; shift key ON, use uppercase
get_ASCII:                            ;  conversion table
        dec     AL           ; index is one less than scan code
        xlat
        cmp     AL,0         ; ASCII code of 0 => uninterested key
        je      EOI_to_8259
        mov     [keyboard_data],AL  ; save ASCII code in keyboard buffer
        jmp     SHORT EOI_to_8259

left_shift_key:
right_shift_key:
        test    AL,80H       ; test key status bit (0=down, 1=up)
        jnz     shift_off
shift_on:
        or      byte [keyboard_flag],1     ; shift bit (i.e., LSB) := 1
        jmp     SHORT EOI_to_8259
shift_off:
        and     byte [keyboard_flag],0FEH  ; shift bit (i.e., LSB) := 0
        jmp     SHORT EOI_to_8259

EOI_to_8259:
        mov     AL,EOI              ; send EOI to 8259 PIC
        out     PIC_CMD_PORT,AL     ; indicating end of ISR
        pop     BX                  ; restore registers
        pop     AX
        iret
```

15.9 Summary

We presented details about the interrupt processing mechanism in the real mode. It is similar to the protected-mode mechanism discussed in the last chapter. The operational differences between the two mechanisms are due to the fact that the real mode works with 16-bit segments.

Software interrupts are often used to support access to the system I/O devices. Both BIOS and DOS provide a high-level interface to the hardware with software interrupts. Hardware interrupts are used by I/O devices to interrupt the processor to service their requests.

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 I/O device. We used several examples to illustrate how this can be done. Specifically, we looked at programmed I/O and interrupt-driven I/O using the keyboard as our example device.

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:

- 8255 PPI chip
- BIOS services
- Block I/O instructions
- DOS services
- Exceptions
- Hardware interrupts
- I/O data transfer
- Interrupt-driven I/O
- Programmable interrupt controller
- Programmed I/O
- Real-mode interrupt processing
- Register I/O instructions
- Scan code
- Software interrupts

15.10 Exercises

15–1 Describe how extended keyboard keys are handled.

15–2 Explain how one can disable all maskable hardware interrupts efficiently. Efficiency here refers to both time and space efficiency of the code.

15–3 In the last question, you looked at a solution to disable all the hardware interrupts. Describe another way to disable all maskable hardware interrupts.

15–4 Write a piece of code to disable all maskable hardware interrupts except the timer and keyboard interrupts. Refer to the interrupt table on page 447.

15–5 Discuss the advantages and disadvantages of the three ways an application program can interact with I/O devices (see Figure 15.2).

15–6 Describe the actions taken (until the beginning of the execution of ISR) by the processor in response to an interrupt `int 10H`. You can assume real mode of operation.

15–7 What is the difference between the DOS keyboard function 0BH and the BIOS keyboard function 01H?

15–8 Discuss the tradeoffs between programmed I/O and interrupt-driven I/O.

15.11 Programming Exercises

15–P1 Write a divide error exception handler to replace the system supplied one. This handler should display a message “A divide error has occurred” and then replace the result with the dividend. You can use registers for the dividend and divisor of the `div` instruction. Test your divide error ISR by making the divisor zero. Also, experiment with the ISR code so that you see that the `div` instruction is restarted because divide error is considered a fault. For example, if your ISR 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 ISR to change the divisor to a value other than 0 in order to proceed with your test program.

15–P2 The `into` instruction generates 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 ISR to replace the system supplied one. Your ISR should display a 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.

15–P3 Convert `toupper.asm` given in Chapter 4 into an ISR for interrupt 100. You can assume that DS:BX points to a null-terminated string. Write a simple program to test your ISR.

15–P4 Write a program to display the date in the format `dd-mmm-yyyy`, where `mmm` is the three-letter abbreviation for the month (e.g., JAN, FEB, etc.). To get the current date, you can use the function 2AH of interrupt 21H. Details are given below:

Function 2AH — Get date

Input:	AH	=	2AH
Returns:	AL	=	day of the week (0 = Sun, 1 = Mon, etc.)
	CX	=	year (1980–2099)
	DH	=	month (1= Jan, 2 = Feb, etc.)
	DL	=	day of the month (1–31)

15–P5 Write a program to display the time in the format `hh:mm:ss`. To get the current time, you can use the function 2CH of interrupt 21H. Details are given below:

Function 2CH — Get time

Input:	AH	=	2CH
Returns:	CH	=	hours (0–23)
	CL	=	minutes (0–59)
	DH	=	seconds (0–59)
	DL	=	hundredths of a second (0–99)

PART V

Advanced Topics

This part consists of the last three chapters: Chapters 16 to 18. These chapters deal with advanced topics such as recursion, high-level language interface, and floating-point operations.

Chapter 16 discusses how recursive procedures are implemented in Pentium and MIPS assembly languages. The next chapter deals with high-level language interface, which allows mixed-mode programming. We use C and Pentium assembly language to cover the principles involved in mixed-mode programming.

The last chapter discusses Pentium's floating-point instructions. To follow the programming examples of this chapter, you need to understand the high-level language interface details presented in Chapter 17.

Chapter 16

Recursion

Objectives

- To introduce principles of recursion
- To show how recursive procedures are written in the Pentium and MIPS assembly languages
- To discuss pros and cons of recursion over iteration

We can use recursion as an alternative to iteration. We introduce the basics of recursion in Section 16.1. The next two sections give some example recursive procedures in the Pentium and MIPS assembly languages. The advantages and pitfalls associated with a recursive solution as opposed to an iterative solution are discussed in Section 16.4. The last section gives a summary.

16.1 Introduction

A recursive procedure calls itself, either directly or indirectly. In direct recursion, a procedure calls itself directly. In indirect recursion, procedure P makes a call to procedure Q, which in turn calls procedure P. The sequence 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 solutions 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

$$\text{factorial}(0) = 1$$
$$\text{factorial}(n) = n * \text{factorial}(n-1) \text{ for } n > 0.$$

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 the 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 151, 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.

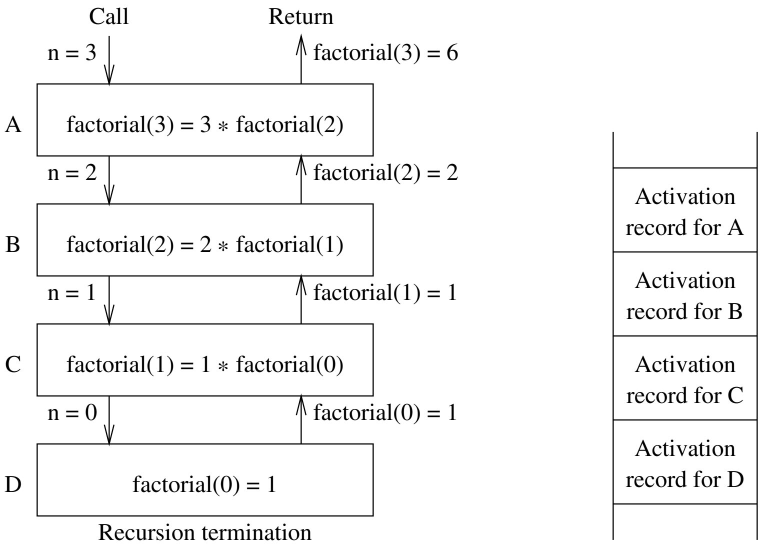
Figure 16.1 shows the stack activation records for factorial(3). As you can see from this figure, each call to the factorial function creates an activation record. Stack is used to keep these activation records. In the next two sections we look at some example recursive procedures in the Pentium and MIPS assembly languages.

16.2 Recursion in Pentium Assembly Language

To illustrate the principles of recursion, we give two examples in the Pentium assembly language. The first example computes the factorial function while the second one implements the popular quicksort algorithm. In the next section we redo these two examples in the MIPS assembly language.

Example 16.1 *Recursive procedure to compute the factorial function.*

An implementation of the factorial function is shown in Program 16.1. The main function provides the user interface. It requests a positive number from the user. If a negative number



(a)

(b)

Figure 16.1 Recursive computation of factorial(3).

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 us one recursive call. When the value in 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 to pass n, it is decremented before the call (line 48) and restored after the call (line 50). The multiply instruction

```
mul    BL
```

multiplies the contents of the BL and AL registers and places the 16-bit result in the AX register.

Program 16.1 Recursive computation of factorial(N)

```
1:  ;Factorial - Recursive version            FACT_PENTIUM.ASM
2:  ;
3:  ;         Objective: To demonstrate principles of recursion.
4:  ;             Input: Requests an integer N from the user.
```

```
;           Output: Outputs N!

%include "io.mac"

.DATA
prompt_msg  db  "Please enter a positive integer: ",0
output_msg  db  "The factorial is: ",0
error_msg   db  "Sorry! Not a positive number. Try again.",0

.CODE
      .STARTUP
      PutStr  prompt_msg     ; request the number

try_again:
      GetInt  BX             ; read number into BX
      cmp     BX,0           ; test for positive number
      jge     num_ok
      PutStr  error_msg
      nwln
      jmp     try_again

num_ok:
      call    fact

      PutStr  output_msg     ; output result
      PutInt  AX
      nwln

done:
      .EXIT

;-----------------------------------------------------------
;Procedure fact receives a positive integer N in BX register.
;It returns N! in AX register.
;-----------------------------------------------------------
.CODE
fact:
      cmp     BL,1           ; if N > 1, recurse
      jg      one_up
      mov     AX,1           ; return 1 for N < 2
      ret                    ; terminate recursion

one_up:
      dec     BL             ; recurse with (N-1)
```

```
        call    fact
        inc     BL
        mul     BL              ; AX = AL * BL

        ret
```

Example 16.2 *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 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 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-the-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 16.2. The input values are read by the read loop (lines 25 to 31). 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 ECX with 200 and using the `loop` instruction on line 31. The three arguments are placed in the EBX (array pointer), ESI (lo), and EDI (hi) registers (lines 35 to 37). After the quicksort call on line 38, the program outputs the sorted array (lines 41 to 50).

The quicksort procedure follows the pseudocode. Since we are not returning any values, we use `pushad` to preserve all registers (line 62). The two inner **while** loops are implemented by the `LO_LOOP` and `HI_LOOP`. The exchange of elements is done by using three `xchg` instructions (lines 89 to 91 and 95 to 97). The rest of the program follows the pseudocode in a straightforward manner.

Program 16.2 Sorting integers using the recursive quicksort algorithm

```
 1:  ;Sorting integers using quicksort        QSORT_PENTIUM.ASM
 2:  ;
 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:  %include "io.mac"
10:
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:  .UDATA
17:  array1      resw  200
18:
19:  .CODE
20:        .STARTUP
21:        PutStr  prompt_msg     ; request the number
22:        nwln
23:        mov     EBX,array1
24:        xor     EDI,EDI        ; EDI keeps a count of input numbers
25:  read_more:
26:        GetInt  AX             ; read a number
27:        mov     [EBX+EDI*2],AX ; store it in array
28:        cmp     AX,0           ; test if it is zero
29:        je      exit_read
30:        inc     EDI
31:        jmp     read_more
32:
33:  exit_read:
34:        ; prepare arguments for procedure call
35:        mov     EBX,array1
36:        xor     ESI,ESI        ; ESI = lo index
37:        dec     EDI            ; EDI = hi index
38:        call    qsort
39:
      PutStr  output_msg     ; output sorted array
write_more:
      ; since qsort preserves all registers, we will
      ; have valid EBX and ESI values.
      mov     AX,[EBX+ESI*2]
      cmp     AX,0
      je      done
      PutInt  AX
      nwln
      inc     ESI
      jmp     write_more

done:
      .EXIT

;------------------------------------------------------------
;Procedure qsort receives a pointer to the array in BX.
;LO and HI are received in ESI and EDI, respectively.
;It preserves all the registers.
;------------------------------------------------------------
.CODE
qsort:
      pushad
      cmp     EDI,ESI
      jle     qsort_done        ; end recursion if hi <= lo

      ; save hi and lo for later use
      mov     ECX,ESI
      mov     EDX,EDI

      mov     AX,[EBX+EDI*2] ; AX = xsep

lo_loop:                        ;
      cmp     [EBX+ESI*2],AX    ;
      jge     lo_loop_done      ; LO while loop
      inc     ESI               ;
      jmp     lo_loop           ;
lo_loop_done:

      dec     EDI            ; hi = hi-1
hi_loop:
      cmp     EDI,ESI           ;
      jle     sep_done          ;
      cmp     [EBX+EDI*2],AX    ; HI while loop
      jle     hi_loop_done      ;
      dec     EDI               ;
      jmp     hi_loop           ;
hi_loop_done:

      xchg    AX,[EBX+ESI*2]    ;
      xchg    AX,[EBX+EDI*2]    ; x[i] <=> x[j]
      xchg    AX,[EBX+ESI*2]    ;
      jmp     lo_loop

sep_done:
      xchg    AX,[EBX+ESI*2]    ;
      xchg    AX,[EBX+EDX*2]    ; x[i] <=> x[hi]
      xchg    AX,[EBX+ESI*2]    ;

      dec     ESI
      mov     EDI,ESI           ; hi = i-1
      ; We will modify the ESI value in the next statement.
      ; Since the original ESI value is in EDI, we will use
      ; EDI value to get i+1 value for the second qsort call.
      mov     ESI,ECX
      call    qsort

      ; EDI has the i value
      inc     EDI
      inc     EDI
      mov     ESI,EDI           ; lo = i+1
      mov     EDI,EDX
      call    qsort

qsort_done:
      popad
      ret
```

16.3 Recursion in MIPS Assembly Language

In MIPS, we could write procedures without using the stack. 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 16.1. In the last section, we presented two example programs, factorial and quicksort, in the Pentium assembly language. Now, we do these two examples in the MIPS assembly language to illustrate how recursion is implemented in MIPS. It also gives you an opportunity to compare the two assembly language implementations.

Example 16.3 *A recursion example—factorial.*
Recall that the factorial function is defined as

0! = 1
N! = N * (N−1)!

Program 16.3 requests an integer N from the input and prints $N!$. This value is passed onto 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 previous examples, 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 back 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 69–71).

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 16.3 Computing factorial—an example recursive function

```
 1:  # Finds factorial of a number                     FACT_MIPS.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:
      .asciiz     "\n"

###################### Code segment ##########################

      .text
      .globl main
main:
      la    $a0,prompt          # prompt user for input
      li    $v0,4
      syscall

try_again:
      li    $v0,5               # read the input number into $a0
      syscall
      move  $a0,$v0

      bgez  $a0,num_OK
      la    $a0,error_msg       # write error message
      li    $v0,4
      syscall
      b     try_again

num_OK:
      jal   fact
      move  $s0,$v0

      la    $a0,out_msg         # write output message
      li    $v0,4
      syscall

      move  $a0,$s0             # output factorial
      li    $v0,1
      syscall

      la    $a0,newline         # write newline
      li    $v0,4
      syscall

      li    $v0,10              # exit
      syscall

#-----------------------------------------------------------
# FACT receives N in $a0 and returns the result in $v0
# It uses recursion to find N!
```

```
#------------------------------------------------------------
fact:
      subu  $sp,$sp,4         # allocate stack space
      sw    $ra,0($sp)        # save return address

      bgt   $a0,1,one_up      # recursion termination
      li    $v0,1
      b     return

one_up:
      subu  $a0,$a0,1         # recurse with (N-1)
      jal   fact
      addu  $a0,$a0,1
      mulou $v0,$a0,$v0       # $v0 = $a0*$v0

return:
      lw    $ra,0($sp)        # restore return address
      addu  $sp,$sp,4         # clear stack space
      jr    $ra
```

Example 16.4 *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 467. Program 16.4 gives an implementation of the quicksort algorithm in the MIPS assembly language. The corresponding Pentium assembly language implementation is given on page 469. One main difference you will notice between these two programs is the addressing modes used to access the array elements. Since MIPS does not support based-indexed addressing, the `qsort` procedure receives two pointers (as opposed to array indexes). Furthermore, 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 43–46 prepare the two arguments for the `qsort` procedure.

The `qsort` recursive procedure stores `a3` in addition to `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 122). 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 index involves adding or subtracting 4 (see lines 94, 99, and 116). The rest of the procedure follows the quicksort algorithm described on page 468. You may find it interesting to compare this program with the Pentium version presented in Example 16.2 to see the similarities and differences between the two assembly languages in implementing recursion.

Program 16.4 Quicksort—another example recursive program

```
# Sorting numbers using quick sort              QSORT_MIPS.ASM
#
# Objective: Sorts an array of integers using quick sort.
#            Uses recursion.
#     Input: Requests integers from the user;
#            terminated by entering a zero.
#    Output: Outputs the sorted integer array.
#
#    a0 - start of array
#    a1 - beginning of (sub)array
#    a2 - end of (sub)array
#
###################### Data segment ##########################
      .data
prompt:
      .ascii     "Please enter integers. \n"
      .asciiz    "Entering zero terminates the input. \n"
output_msg:
      .asciiz     "The sorted array is: \n"
newline:
      .asciiz     "\n"
array:
      .word       200

###################### Code segment ##########################

      .text
      .globl main
main:
      la    $a0,prompt         # prompt user for input
      li    $v0,4
      syscall

      la    $t0,array
read_more:
      li    $v0,5              # read a number
      syscall
      sw    $v0,($t0)          # store it in the array
      beqz  $v0,exit_read
      addu  $t0,$t0,4
      b     read_more
exit_read:
      # prepare arguments for procedure call
      la    $a1,array          # a1 = lo pointer
      move  $a2,$t0
      subu  $a2,$a2,4          # a2 = hi pointer
      jal   qsort

      la    $a0,output_msg     # write output message
      li    $v0,4
      syscall

      la    $t0,array
write_more:
      lw    $a0,($t0)          # output sorted array
      beqz  $a0,exit_write
      li    $v0,1
      syscall
      la    $a0,newline        # write newline message
      li    $v0,4
      syscall
      addu  $t0,$t0,4
      b     write_more
exit_write:

      li    $v0,10             # exit
      syscall

#------------------------------------------------------------
# QSORT receives pointer to the start of (sub)array in a1 and
# end of (sub)array in a2.
#------------------------------------------------------------
qsort:
      subu  $sp,$sp,16         # save registers
      sw    $a1,0($sp)
      sw    $a2,4($sp)
      sw    $a3,8($sp)
      sw    $ra,12($sp)

      ble   $a2,$a1,done       # end recursion if hi <= lo

      move  $t0,$a1
      move  $t1,$a2

      lw    $t5,($t1)          # t5 = xsep

lo_loop:                            #
```

```
      lw    $t2,($t0)               #
      bge   $t2,$t5,lo_loop_done    # LO while loop
      addu  $t0,$t0,4               #
      b     lo_loop                 #
lo_loop_done:

      subu  $t1,$t1,4          # hi = hi-1
hi_loop:
      ble   $t1,$t0,sep_done        #
      lw    $t3,($t1)               #
      blt   $t3,$t5,hi_loop_done    # HI while loop
      subu  $t1,$t1,4               #
      b     hi_loop                 #
hi_loop_done:

      sw     $t2,($t1)              #
      sw     $t3,($t0)              # x[i]<=>x[j]
      b      lo_loop                #

sep_done:
      move  $t1,$a2                 #
      lw    $t4,($t0)               #
      lw    $t5,($t1)               # x[i] <=>x[hi]
      sw    $t5,($t0)               #
      sw    $t4,($t1)               #

      move  $a3,$a2            # save HI for the second call
      move  $a2,$t0                 #
      subu  $a2,$a2,4               # set hi as i-1
      jal   qsort

      move  $a1,$a2                 #
      addu  $a1,$a1,8               # set lo as i+1
      move  $a2,$a3
      jal   qsort
done:
      lw    $a1,0($sp)         # restore registers
      lw    $a2,4($sp)
      lw    $a3,8($sp)
      lw    $ra,12($sp)
      addu  $sp,$sp,16

      jr    $ra
```

16.4 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 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 (see Programming Exercise 1 for the definition of this function)

  ```
  fib(5) = fib(4) + fib(3)
  ```

  a recursive procedure computes `fib(3)` two times, `fib(2)` two times, and so on.
- *Increased memory requirement:* Recursion tends to demand 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 aids in program maintenance.

16.5 Summary

We can use recursive procedures as an alternative to iterative ones. A procedure that calls itself, whether directly or indirectly, is called a recursive procedure. In direct recursion, a procedure calls itself, as in our factorial example. In indirect recursion, a procedure may initiate a sequence of calls that eventually results in calling the procedure itself.

For some applications, we can write an elegant solution because recursion is a natural fit. We illustrated the principles of recursion using two examples: factorial and quicksort. We presented recursive versions of these functions in the Pentium and MIPS assembly languages. In the last section we identified the tradeoffs associated with recursion as opposed to iteration.

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
- Direct recurion
- Factorial function
- Indirect recursion
- Quicksort
- Recursion

16.6 Exercises

16–1 What is direct recursion?

16–2 What is indirect recursion?

16–3 We know that MIPS procedures can be written without using the stack. However, we need to use the stack with recursion. Explain why.

16–4 What are the differences between iteration and recursion?

16–5 Explain why the stack size imposes a restriction on recursion depth.

16–6 What are the potential problems with recursion compared to iteration?

16–7 In Section 16.4 we talked about duplicate computation as one of the problems with recursion. Do we have this problem with the factorial and quicksort solutions presented in this chapter?

16.7 Programming Exercises

16–P1 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 5 (see page 152). 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.

16–P2 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.

16–P3 Write an assembly language 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 years on whatever PC you have!

Chapter 17

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 language programs. This chapter considers mixed-mode programming, which refers to writing parts of a program in different programming languages. 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 17.1. Section 17.2 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 17.5. Other sections focus on the separate assembly module method.

Section 17.3 describes the mechanics involved in calling assembly language procedures from a C program. This section presents details about parameter passing, returning values of C functions, and so on. Section 17.4 shows how a C function can be called from an assembly language procedure. The last section summarizes the chapter.

17.1 Why Program in Mixed Mode?

In this chapter we focus on mixed-mode programming that involves C and assembly languages. Thus, we write part of the program in C and the other part in the Pentium assembly language. We use the gcc compiler and NASM 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 1 we discussed several reasons why one would want to program in the assembly language. Although it is possible to write a program entirely in the assembly language, there are several disadvantages in doing so. These include

- Low productivity
- High maintenance cost
- 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 the assembly language are difficult to maintain. This is a direct consequence of it's being a low-level language. In addition, assembly language programs are not portable. On the other hand, the assembly language provides low-level access to system hardware. In addition, the assembly language may help us reduce the execution time.

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.

17.2 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 contains assembly language instructions. Most C compilers including gcc 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 embed. Otherwise, separate assembly modules are preferred. We discuss the inline assembly method in Section 17.5.

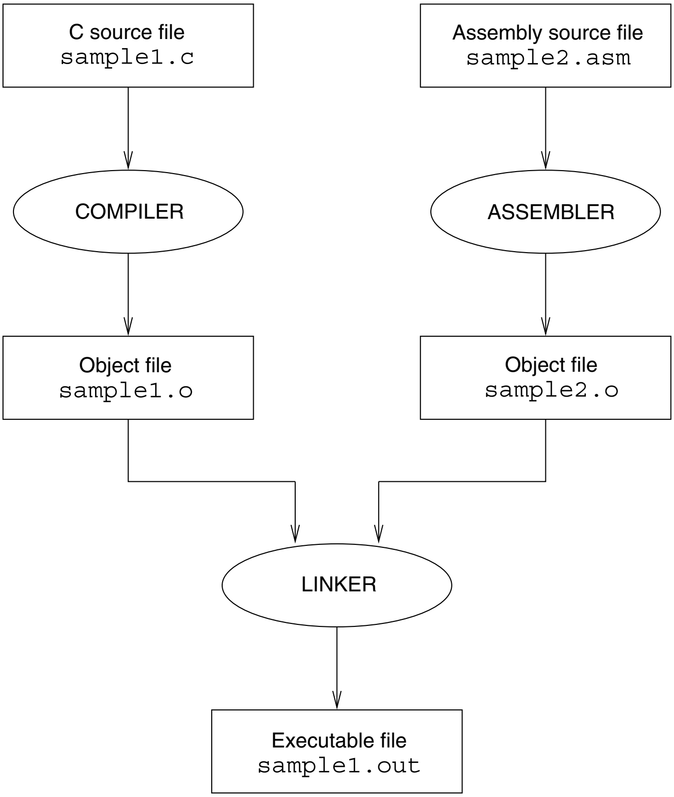


Figure 17.1 Steps involved in compiling mixed-mode programs.

When separate modules are used for C and assembly languages, each module can be translated into the corresponding object file. To do this translation, we use a C compiler for the C modules and an assembler for the assembly modules, as shown in Figure 17.1. Then the linker can be used to produce the executable 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 17.1. We can invoke the NASM assembler as

```
nasm -f elf sample2.asm
```

This creates the `sample2.o` object file. We can compile and link the files with the following command:

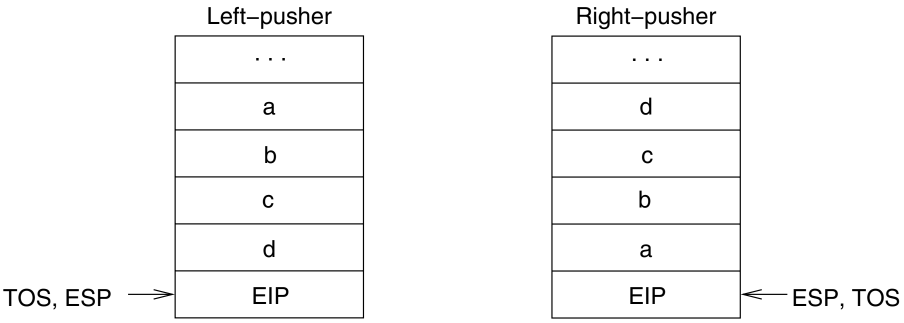


Figure 17.2 Two ways of pushing arguments onto the stack.

```
gcc -o sample1.out sample1.c sample2.o
```

This command instructs the compiler to first compile `sample1.c` to `sample1.o`. The linker is automatically invoked to link `sample1.o` and `sample2.o` to produce the executable file `sample1.out`.

17.3 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 a few more details as to 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.

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 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 17.2. From now on, we consider only right-pushing of arguments, as we focus on the C language.

To see how `gcc` pushes arguments onto the stack, take a look at the following C program (this is a partial listing of Example 17.1):

```
int main(void)
{
    int     x=25, y=70;
    int     value;
    extern  int test(int, int, int);

    value = test (x, y, 5);
    . . .
}
```

The assembly language translation of the procedure call (use `-S` option to generate the assembly source code) is shown below:[1]

```
push    5
push    70
push    25
call    test
add     ESP,12
mov     [EBP-12],EAX
```

This program is compiled with `-O2` optimization. This optimization is the reason for pushing constants 70 and 25 instead of variables `x` and `y`. If you don't use this optimization, `gcc` produces the following code:

```
push    5
push    [EBP-8]
push    [EBP-4]
call    test
add     ESP,12
mov     [EBP-12],EAX
```

It is obvious from this code fragment that the compiler assigns space for variables `x`, `y`, and `value` on the stack at EBP−4, EBP−8, and EBP−12, 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 the following statement:

```
add   ESP,12
```

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 (see our discussion in Section 5.8 on page 146).

[1] Note that `gcc` uses AT&T syntax for the assembly language—not the Intel syntax we have been using in this book. To avoid any confusion, the contents are reported in our syntax. The AT&T syntax is introduced in Section 17.5.

Returning Values

We can see from the assembly language code given in the last subsection that the EAX register is used to return the function value. In fact, the EAX is used to return 8-, 16-, and 32-bit values. To return a 64-bit value, use the EDX:EAX pair with the EDX holding the upper 32 bits.

We have not discussed how floating-point values are returned. For example, if a C function returns a `double` value, how do we return this value? We discuss this issue in Chapter 18.

Preserving Registers

In general, the called assembly language procedure can use the registers as needed, except that the following registers should be preserved:

```
EBP, EBX, ESI, EDI
```

The other registers, if needed, must be preserved by the calling function.

Globals 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 global, as discussed in Chapter 5. Before proceeding further, you may want to review the material on multimodule programs presented in Chapter 5 (see Section 5.10 on page 156). Here we mention only those details that are specific to the mixed-mode programming involving C and assembly language.

In most C compilers, 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. However, `gcc` does not follow this convention by default. Thus, we don't have to worry about the underscore.

17.3.1 Illustrative Examples

We now look at three examples to illustrate the interface between C and assembly language programs. We start with a simple example, whose C part has been dissected before.

Example 17.1 *Our first mixed-mode example.*

This example passes three parameters to the assembly language function `test1`. The C code is shown in Program 17.1 and the assembly code in Program 17.2. The function `test1` is declared as external in the C program (line 12) and global in the assembly program (line 8). 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 17.1 An example illustrating assembly calls from C: C code (in file `hll_ex1c.c`)

```
/*****************************************************************
 * A simple program to illustrate how mixed-mode programs are
 * written in C and assembly languages. The main C program calls
 * the assembly language procedure test1.
 *****************************************************************/
#include        <stdio.h>

int main(void)
{
    int    x = 25, y = 70;
    int    value;
    extern int test1 (int, int, int);

    value = test1(x, y, 5);
    printf("Result = %d\n", value);

    return 0;
}
```

Program 17.2 An example illustrating assembly calls from C: assembly language code (in file `hll_test.asm`)

```
;-----------------------------------------------------------
; This procedure receives three integers via the stack.
; It adds the first two arguments and subtracts the third one.
; It is called from the C program.
;-----------------------------------------------------------
segment .text

global  test1

test1:
      enter   0,0
      mov     EAX,[EBP+8]        ;  get argument1 (x)
      add     EAX,[EBP+12]       ;  add argument 2 (y)
      sub     EAX,[EBP+16]       ;  subtract argument3 (5)
      leave
      ret
```

Example 17.2 *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 17.3. The assembly procedure `min_max`, shown in Program 17.4, 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 these two pointers. The minimum value is kept in EAX and the maximum in EDX. The code given on lines 27 to 30 in Program 17.4 stores the return values by using the EBX register in the indirect addressing mode.

Program 17.3 An example with the C program passing pointers to the assembly program: C code (in file `hll_minmaxc.c`)

```c
/***********************************************************
 * An example to illustrate call-by-value and              *
 * call-by-reference parameter passing between C and       *
 * assembly language modules. The min_max function is      *
 * written in assembly language (in the file hll_minmaxa.asm).  *
 ***********************************************************/
#include <stdio.h>
int main(void)
{
      int     value1, value2, value3;
      int     minimum, maximum;
      extern  void min_max (int, int, int, int*, int*);

      printf("Enter number 1 = ");
      scanf("%d", &value1);
      printf("Enter number 2 = ");
      scanf("%d", &value2);
      printf("Enter number 3 = ");
      scanf("%d", &value3);

      min_max(value1, value2, value3, &minimum, &maximum);
      printf("Minimum = %d, Maximum = %d\n", minimum, maximum);
      return 0;
}
```

Program 17.4 An example with the C program passing pointers to the assembly program: assembly language code (in file `hll_minmax_a.asm`)

```
;-------------------------------------------------------------
; Assembly program for the min_max function - called from the
; C program in the file hll_minmaxc.c. This function finds
; the minimum and maximum of the three integers it receives.
;-------------------------------------------------------------
global  min_max

min_max:
      enter   0,0
      ; EAX keeps minimum number and EDX maximum
      mov     EAX,[EBP+8]      ; get value 1
      mov     EDX,[EBP+12]     ; get value 2
      cmp     EAX,EDX          ; value 1 < value 2?
      jl      skip1            ; if so, do nothing
      xchg    EAX,EDX          ; else, exchange
skip1:
      mov     ECX,[EBP+16]     ; get value 3
      cmp     ECX,EAX          ; value 3 < min in EAX?
      jl      new_min
      cmp     ECX,EDX          ; value 3 < max in EDX?
      jl      store_result
      mov     EDX,ECX
      jmp     store_result
new_min:
      mov     EAX,ECX
store_result:
      mov     EBX,[EBP+20]     ; EBX = &minimum
      mov     [EBX],EAX
      mov     EBX,[EBP+24]     ; EBX = &maximum
      mov     [EBX],EDX
      leave
      ret
```

Example 17.3 *Array sum example.*

This example illustrates how arrays, declared in C, are accessed by assembly language procedures. The array `value` is declared in the C program, as shown in Program 17.5 (line 12). The assembly language procedure computes the sum as shown in Program 17.6. As in the other programs in this chapter, the C program clears the parameters off the stack. We will redo this example using inline assembly in Section 17.5.

Program 17.5 An array sum example: C code (in file hll_arraysumc.c)

```
/************************************************************
 * This program reads 10 integers into an array and calls an
 * assembly language program to compute the array sum.
 * The assembly program is in the file "hll_arraysuma.asm".
 ************************************************************/
#include        <stdio.h>

#define  SIZE  10

int main(void)
{
    int    value[SIZE], sum, i;
    extern int array_sum(int*, int);

    printf("Input %d array values:\n", SIZE);
    for (i = 0; i < SIZE; i++)
        scanf("%d",&value[i]);

    sum = array_sum(value,SIZE);
    printf("Array sum = %d\n", sum);

    return 0;
}
```

Program 17.6 An array sum example: assembly language code (in file hll_arraysuma.asm)

```
;------------------------------------------------------------
; This procedure receives an array pointer and its size via
; the stack. It computes the array sum and returns it.
;------------------------------------------------------------
segment .text

global  array_sum

array_sum:
      enter   0,0
      mov     EDX,[EBP+8]    ; copy array pointer to EDX
      mov     ECX,[EBP+12]   ; copy array size to ECX
      sub     EBX,EBX        ; array index = 0
      sub     EAX,EAX        ; sum = 0 (EAX keeps the sum)
```

```
add_loop:
      add     EAX,[EDX+EBX*4]
      inc     EBX            ; increment array index
      cmp     EBX,ECX
      jl      add_loop
      leave
      ret
```

17.4 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 language code for a complex task. 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 17.4 *An example to illustrate a C function call from an assembly procedure.*
In previous chapters, we used simple I/O routines to facilitate input and output in our assembly language programs. If we want to use the C functions like `printf()` and `scanf()`, we have to pass the arguments as required by the function. In this example, we show how we can use these two C functions to facilitate input and output of integers. This discussion can be generalized to other types of data.

Here we compute the sum of an array passed onto the assembly language procedure `array_sum`. This example is similar to Example 17.3, except that the C program does not read the array values; instead, the assembly program does this by calling the `printf()` and `scanf()` functions as shown in Program 17.8. In this program, the prompt message is declared as a string on line 9 (including the newline). The assembly language version implements the equivalent of the following `printf` statement we used in Program 17.5:

```
printf("Input %d array values:\n", SIZE);
```

Before calling the `printf` function on line 21, we push the array size (which is in ECX) and the string onto the stack. The stack is cleared on line 22.

The array values are read using the read loop on lines 26 to 36. It uses the `scanf` function, the equivalent of the following statement:

```
scanf("%d",&value[i]);
```

The required arguments (array and format string pointers) are pushed onto the stack on lines 28 and 29 before calling the `scanf` function on line 30. The array sum is computed using the add loop on lines 41 to 45 as in Program 17.6.

Program 17.7 An example to illustrate C calls from assembly programs: C code (in file hll_arraysum2c.c)

```
/************************************************************
 * This program calls an assembly program to read the array
 * input and compute its sum. This program prints the sum.
 * The assembly program is in the file "hll_arraysum2a.asm".
 ************************************************************/
#include        <stdio.h>

#define  SIZE  10

int main(void)
{
    int    value[SIZE];
    extern int array_sum(int*, int);

    printf("sum = %d\n",array_sum(value,SIZE));

    return 0;
}
```

Program 17.8 An example to illustrate C calls from assembly programs: assembly language code (in file hll_arraysum2a.asm)

```
;-----------------------------------------------------------
; This procedure receives an array pointer and its size
; via the stack. It first reads the array input from the
; user and then computes the array sum.
; The sum is returned to the C program.
;-----------------------------------------------------------
segment .data
scan_format    db   "%d",0
printf_format  db   "Input %d array values:",10,13,0

segment .text

global  array_sum
extern  printf,scanf

array_sum:
      enter   0,0
      mov     ECX,[EBP+12]   ; copy array size to ECX
      push    ECX            ; push array size
      push    dword printf_format
      call    printf
      add     ESP,8          ; clear the stack

      mov     EDX,[EBP+8]    ; copy array pointer to EDX
      mov     ECX,[EBP+12]   ; copy array size to ECX
read_loop:
      push    ECX            ; save loop count
      push    EDX            ; push array pointer
      push    dword scan_format
      call    scanf
      add     ESP,4          ; clear stack of one argument
      pop     EDX            ; restore array pointer in EDX
      pop     ECX            ; restore loop count
      add     EDX,4          ; update array pointer
      dec     ECX
      jnz     read_loop

      mov     EDX,[EBP+8]    ; copy array pointer to EDX
      mov     ECX,[EBP+12]   ; copy array size to ECX
      sub     EAX,EAX        ; EAX = 0 (EAX keeps the sum)
add_loop:
      add     EAX,[EDX]
      add     EDX,4          ; update array pointer
      dec     ECX
      jnz     add_loop
      leave
      ret
```

17.5 Inline Assembly

In this section we look at writing inline assembly code. In this method, we embed assembly language statements within the C code. We identify assembly language statements by using the asm construct. (You can use __asm__ if asm causes a conflict, e.g., for ANSI C compatibility.)

We now have a serious problem: the syntax that the gcc compiler uses for assembly language statements is different from the syntax we have been using so far. We have been using the Intel syntax (NASM, TASM, and MASM use this syntax). The gcc compiler uses the AT&T syntax, which is used by GNU assemblers. It is different in several aspects from the Intel syntax. But don't worry! We give an executive summary of the differences so that you can understand the syntactical differences without spending too much time.

17.5.1 The AT&T Syntax

This section gives a summary of some of the key differences from the Intel syntax.

Register Naming In the AT&T syntax, we have to prefix register names with %. For example, the EAX register is specified as %eax.

Source and Destination Order The source and destination operands order is reversed in the AT&T syntax. In this format, source operand is on the left-hand side. For example, the instruction

```
mov     eax,ebx
```

is written as

```
movl     %ebx,%eax
```

Operand Size As demonstrated by the last example, the instructions specify the operand size. The instructions are suffixed with b, w, and l for byte, word, and longword operands, respectively. With this specification, we don't have to use byte, word, and dword to clarify the operand size (see our discussion in Section 4.4.2 on page 79).

The operand size specification is not strictly necessary. You can let the compiler guess the size of the operand. However, if you specify, it takes the guesswork out and we don't have to worry about the compiler making an incorrect guess. Here are some examples:

```
movb    %bl,%al       ; moves contents of bl to al
movw    %bx,%ax       ; moves contents of bx to ax
movl    %ebx,%eax     ; moves contents of ebx to eax
```

Immediate and Constant Operands In the AT&T syntax, immediate and constant operands are specified by prefixing with $. Here are some examples:

```
movb     $255,%al
movl     $0xFFFFFFFF,%eax
```

The following statement loads the address of the C global variable total into the EAX register:

```
movl     $total,%eax
```

This works only if total is declared as a global variable. Otherwise, we have to use the extended asm construct that we discuss later.

Addressing To specify indirect addressing, the AT&T syntax uses brackets (not square brackets). For example, the instruction

```
mov     eax,[ebx]
```

is written in AT&T syntax as

```
movl    (%ebx),%eax
```

The full 32-bit protected-mode addressing format is shown below:

```
imm32(base,index,scale)
```

The address is computed as

```
imm32 + base + index * scale
```

If we declared `marks` as a global array of integers, we can load `marks[5]` into EAX register using

```
movl  $5,%ebx
movl  marks(,%ebx,4),%eax
```

For example, if the pointer to `marks` is in the EAX register, we can load `marks[5]` into the EAX register using

```
movl  $5,%ebx
movl  (%eax,%ebx,4),%eax
```

We use a similar technique in the array sum example discussed later. We have covered enough details to work with the AT&T syntax.

17.5.2 Simple Inline Statements

At the basic level, introducing assembly statements is not difficult. Here is an example that increments the EAX register contents:

```
 asm("incl %eax");
```

Multiple assembly statements like these

```
asm("pushl   %eax");
asm("incl    %eax");
asm("popl    %eax");
```

can be grouped into a single compound `asm` statement as shown below:

```
asm("pushl   %eax; incl    %eax; popl    %eax");
```

If you want to add structure to this compound statement, you can write the above statement as follows:

```
asm("pushl   %eax;"
    "incl    %eax;"
    "popl    %eax");
```

We have one major problem in accessing the registers as we did here: How do we know if `gcc` is not keeping something useful in the register that we are using? More importantly, how do we get access to C variables that are not global to manipulate in our inline assembly code? The answers are provided by the extended `asm` statement. This is where we are going next.

17.5.3 Extended Inline Statements

The format of the `asm` statement consists of four components as shown below:

```
asm(assembly code
    :outputs
    :inputs
    :clobber list);
```

Each component is separated by a colon (`:`). The last three components are optional. These four components are described next.

Assembly Code This component consists of the assembly language statements to be inserted into the C code. This may have a single instruction or a sequence of instructions, as discussed in the last subsection. If no compiler optimization should be done to this code, add the keyword `volatile` after `asm` (i.e., use `asm volatile`). The instructions typically use the operands specified in the next two components.

Outputs This component specifies the output operands for the assembly code. The format for specifying each operand is shown below:

```
"=op-constraint" (C-expression)
```

The first part specifies an operand constraint, and the part in brackets is a C expression. The = identifies that it is an output constraint. For some strange reason we have to specify = even though we separate inputs and outputs with a colon. The following example

```
"=r" (sum)
```

specifies that the C variable `sum` should be mapped to a register as indicated by `r` in the constraint. Multiple operands can be specified by separating them with commas. We give some examples later.

Depending on the processor, several other choices are allowed including `m` (memory), `i` (immediate), `rm` (register or memory), `ri` (register or immediate), or `g` (general). The last one is typically equivalent to `rim`. You can also specify a particular register by using `a`, `b`, and so on. The following table summarizes the register letters used to specify which registers that `gcc` may use:

Letter	Register set
a	EAX register
b	EBX register
c	ECX register
d	EDX register
S	ESI register
D	EDI register
r	Any of the eight general registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	Any of the four data registers (EAX, EBX, ECX, EDX)
A	A 64-bit value in EAX and EDX
f	Floating-point registers
t	Top floating-point register
u	Second top floating-point register

The last three letters are used to specify floating-point registers. We discuss floating-point operations in Chapter 18.

Inputs The inputs are also specified in the same way (except for the = sign). The operands specified in the output and input parts are assigned sequence numbers 0, 1, 2, ... starting with the leftmost output operand. There can be a total of 10 operands, inputs and outputs combined. Thus, 9 is the maximum sequence number allowed.

In the assembly code, we can refer to the output and input operands by their sequence number prefixed with `%`. In the following example

```
asm("movl %1,%0"
     :"=r"(sum)      /* output */
     :"r"(number1)   /* input  */
    );
```

the C variables `sum` and `number1` are both mapped to registers. In the assembly code statement, `sum` is identified by `%0` and `number1` by `%1`. Thus, this statement copies the value of `number1` to `sum`.

Sometimes, an operand provides input and receives the result as well (e.g., x in x = x + y). In this case, the operand should be in both lists. In addition, you should use its output sequence number as its input constraint specifier. The following example clarifies what we mean.

```
asm("addl %1,%0"
     :"=r"(sum)                /* output */
     :"r"(number1), "0"(sum)   /* inputs */
    );
```

In this example, we want to perform `sum = sum + number1`. In this expression, the variable `sum` provides one of the inputs and also receives the result. Thus, `sum` is in both lists. However, note that the constraint specifier for it in the input list is `"0"`, not `"r"`.

The assembly code can use specific registers prefixing the register with `%`. Since the AT&T syntax prefixes registers with `%`, we end up using `%%` as in `%%eax` to refer to the EAX register.

Clobber List This last component specifies the list of registers modified by the assembly instructions in the `asm` statement. This lets `gcc` know that it cannot assume that the contents of these registers are valid after the `asm` statement. The compiler may use this information to reload their values after executing the `asm` statement.

In case the assembly code modifies the memory, use the keyword `"memory"` to indicate this fact. Even though it may not be needed, you may want to specify `"cc"` in the clobber list if the flags register is modified (e.g., by an arithmetic instruction). Here is an example that includes the clobber list:

```
asm("movl %0,%%eax"
     : /* no output */
     :"r"(number1)  /* inputs */
     :"%eax"        /* clobber list */
   );
```

In this example, there is no output list; thus, the input operand (`number1`) is referred by `%0`. Since we copy the value of `number1` into EAX register, we specify EAX in the clobber list so that `gcc` knows that our `asm` statement modifies this register.

17.5.4 Inline Examples

We now give some examples to illustrate how we can write mixed-mode programs using the inline assembly method.

Example 17.5 *Our first inline assembly example.*

As our first example, we rewrite the code for Example 17.1 using inline assembly (see Program 17.9). The procedure `test1` is written using inline assembly code. We use the EAX register to compute the sum as in Program 17.2 (see lines 22–24). Since there are no output operands, we explicitly state this by the comment on line 25. The three input operands x, y, and z, specified on line 26, are referred in the assembly code as %0, %1, and %2, respectively. The clobbered list consists of EAX register and the flags register ("cc") as the add and sub instructions modify the flags register. Since the result is available in the EAX register, we simply return from the function.

Program 17.9 Our first inline assembly code example (in file hll_ex1_inline.c)

```
 1: /****************************************************************
 2:  * A simple program to illustrate how mixed-mode programs are
 3:  * written in C and assembly languages. This program uses inline
 4:  * assembly code in the test1 function.
 5:  ****************************************************************/
 6: #include        <stdio.h>
 7:
 8: int main(void)
 9: {
10:         int    x = 25, y = 70;
11:         int    value;
12:         extern int test1 (int, int, int);
13:
14:         value = test1(x, y, 5);
15:         printf("Result = %d\n", value);
16:
17:         return 0;
18: }
19:
20: int test1(int x, int y, int z)
21: {
22:        asm("movl  %0,%%eax;"
23:            "addl  %1,%%eax;"
24:            "subl  %2,%%eax;"
25:             :/* no outputs */        /* outputs */
26:             : "r"(x), "r"(y), "r"(z) /* inputs */
27:             :"cc","%eax");           /* clobber list */
28: }
```

Example 17.6 *Array sum example—inline version.*

This is the inline assembly version of the array sum example we did in Example 17.3. The program is given in Program 17.10. In the array_sum procedure, we replace the C statement

```
      sum += value[i];
```

by the inline assembly code. The output operand specifies `sum`. The input operand list consists of the array `value`, array index variable `i`, and `sum`. Since `sum` is also in the output list, we use `"0"` as explained before. Since we use the `add` instruction, we specify `"cc"` in the clobber list as in the last example.

The assembly code consists of a single `addl` instruction. The source operand of this add instruction is given as `(%1,%2,4)`. From our discussion on page 497 it is clear that this operand refers to `value[i]`. The rest of the code is straightforward to follow.

Program 17.10 Inline assembly version of the array sum example (in file `hll_arraysum_inline.c`)

```
/************************************************************
 * This program reads 10 integers into an array and calls an
 * assembly language program to compute the array sum.
 * It uses inline assembly code in array_sum function.
 ************************************************************/
#include        <stdio.h>

#define  SIZE  10

int main(void)
{
    int    value[SIZE], sum, i;
    int    array_sum(int*, int);

    printf("Input %d array values:\n", SIZE);
    for (i = 0; i < SIZE; i++)
        scanf("%d",&value[i]);

    sum = array_sum(value,SIZE);
    printf("Array sum = %d\n", sum);

    return 0;
}

int array_sum(int* value, int size)
{
    int  i, sum=0;
    for (i = 0; i < size; i++)
         asm("addl (%1,%2,4),%0"
              :"=r"(sum)                   /* output */
              :"r"(value),"r"(i),"0"(sum)  /* inputs */
              :"cc");                      /* clobber list */
        return(sum);
}
```

Example 17.7 *Array sum example—inline version 2.*
In the last example, we just replaced the statement

```
sum += value[i];
```

of the `array_sum` function by the assembly language statement. In this example, we rewrite the `array_sum` function completely in the assembly language. The rewritten function is shown in Program 17.11. This code illustrates some of the features we have not used in the previous examples.

As you can see from line 10, we receive the two input parameters (`value` and `size`) in specific registers (`value` in EBX and `size` in ECX). We compute the sum directly in the EAX register, so there are no outputs in the `asm` statement (see line 9). We don't use `"%0"` and `"%1"` to refer to the input operands. Since these are mapped to specific registers, we can use the register names in our assembly language code (see lines 5 and 6).

We use the EAX register to keep the sum. This register is initialized to zero on line 3. We use `jecxz` to test if ECX is zero. This is the termination condition for the loop. This code also shows how we can use jump instructions and labels.

Program 17.11 Another inline assembly version of the array_sum function (This function is in file `hll_arraysum_inline2.c`)

```
 1:  int array_sum(int* value, int size)
 2:  {
 3:          asm("      xorl  %%eax,%%eax;"   /* sum = 0 */
 4:              "rep1: jecxz done;       "
 5:              "      decl  %%ecx;      "
 6:              "      addl  (%%ebx,%%ecx,4),%%eax;"
 7:              "      jmp   rep1;       "
 8:              "done:                   "
 9:                : /* no outputs */
10:                :"b"(value),"c"(size)      /* inputs */
11:                :"%eax","cc");             /* clobber list */
12:  }
```

17.6 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 `gcc` compiler and NASM assembler, 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:

- `asm` directive
- Inline assembly
- Left-pusher language
- Mixed-mode programs
- Parameter passing
- Right-pusher language

17.7 Exercises

17–1 Why do we need to write mixed-mode programs?

17–2 Describe how parameters are passed from a C function to an assembly language procedure.

17–3 What is the difference between right-pusher and left-pusher languages (as far as parameter passing is concerned)?

17–4 Explain why, in C, the calling function is responsible for clearing the stack.

17–5 What are the pros and cons of inline assembly as opposed to separate assembly modules?

17–6 What is the purpose of the clobber list in the extended `asm` statement?

17–7 Describe how we can access C variables (that are not global) in inline assembly code.

17.8 Programming Exercises

17–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`.

17–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.

17–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.

17–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-hand 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 18

Floating-Point Operations

Objectives

- To introduce the floating-point unit details
- To describe the floating-point instructions
- To give example programs that use the floating-point instructions

This chapter gives an introduction to the floating-point instructions of the Pentium processor. After giving a brief introduction to floating-point numbers, we describe the floating-point registers in Section 18.2. The Pentium's floating-point unit supports several floating-point instructions. Some of these instructions are described in the next section. We give some examples illustrating the application of the floating-point instructions in Section 18.4. We conclude the chapter with a summary.

18.1 Introduction

In the previous chapters, we represented numbers using integers. As you know, these numbers cannot be used to represent fractions. We use floating-point numbers to represent fractions. For example, in C, we use the `float` and `double` data types for the floating-point numbers.

One key characteristic of integers is that operations on these numbers are always precise. For example, when we add two integers, we always get the exact result. In contrast, operations on floating-point numbers are subjected to rounding-off errors. This tends to make the result approximate, rather than precise. However, floating-point numbers have several advantages.

Floating-point numbers can used to represent both very small numbers and very large numbers. To achieve this, these numbers use the scientific notation to represent numbers. The number is divided into three parts: the *sign*, the *mantissa*, and the *exponent*. The sign bit identifies whether the number is positive (0) or negative (1). The magnitude is given by

$$\text{magnitude} = \text{mantissa} \times 2^{\text{exponent}}$$

Implementation of floating-point numbers on computer systems vary from this generic format—usually for efficiency reasons or to conform to a standard. Pentium, like most other processors, follows 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 floating-point unit (FPU) supports three formats for floating-point numbers. Two of these are for external use and one for internal use. The external format defines two precision types: single-precision format uses 32 bits while the double-precision format uses 64 bits. In C, we use `float` for single-precision and `double` for double-precision floating-point numbers. The internal format uses 80 bits and is referred to as the extended format. As we see in the next section, all internal registers of the floating-point unit are 80 bits so that they can store floating-point numbers in the extended format. More details on the floating-point numbers are given in Appendix A.

The number-crunching capability of a processor can be enhanced by using a special hardware to perform floating-point operations. The 80X87 numeric coprocessors were designed to work with the 80X86 family of processors. The 8087 coprocessor was designed for the 8086 and 8088 processors to provide extensive high-speed numeric processing capabilities. The 8087, for example, provided about a hundredfold improvement in execution time compared to that of an equivalent software function on the 8086 processor. The 80287 and 80387 coprocessors were designed for use with the 80286 and 80386 processors, respectively. Starting with the 80486 processor, the floating-point unit has been integrated into the processor itself, avoiding the need for external numeric processors.

In the remainder of this chapter, we discuss the floating-point unit organization and its instructions. Toward the end of the chapter, we give several example programs that use the floating-point instructions.

18.2 Floating-Point Unit Organization

The floating-point unit provides several registers, as shown in Figure 18.1. These registers are divided into three groups: data registers, control and status registers, and pointer registers. The last group consists of the instruction and data pointer registers, as shown in Figure 18.1. These pointers provide support for programmed exception handlers. Since this topic is beyond the scope of this book, we do not discuss details of these registers.

18.2.1 Data Registers

The FPU has eight floating-point registers to hold the floating-point operands. These registers supply the necessary operands to the floating-point instructions. Unlike the processor's general-purpose registers like the EAX and EBX registers, these registers are organized as a

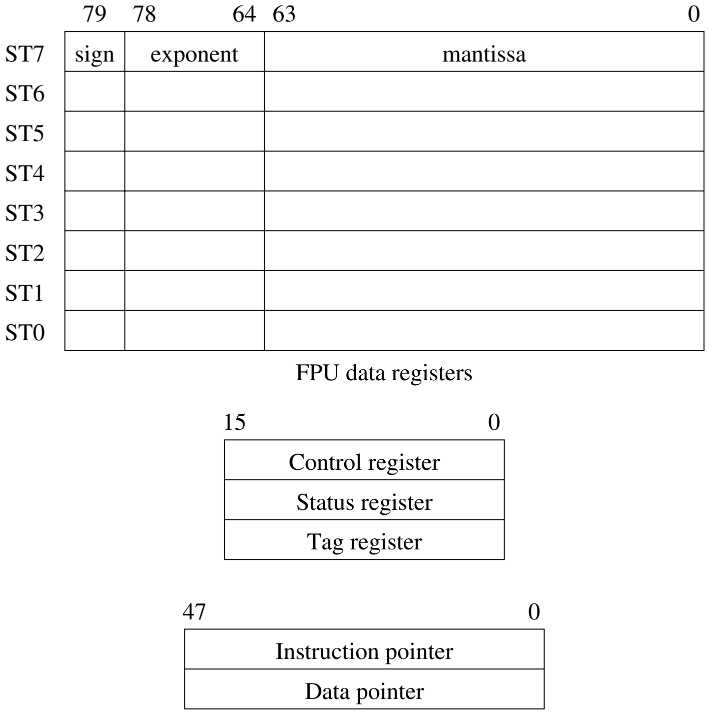


Figure 18.1 FPU registers.

register stack. In addition, we can access these registers individually using `ST0`, `ST1`, and so on.

Since these registers are organized as a register stack, these names are not statically assigned. That is, ST0 does not refer to a specific register. It refers to whichever register is acting as the top-of-stack (TOS) register. The next register is referred to as `ST1`, and so on; the last register as `ST7`. There is a 3-bit top-of-stack pointer in the status register to identify the TOS register.

Each data register can hold an extended-precision floating-point number. This format uses 80 bits as opposed single-precision (32 bits) or double-precision (64 bits) formats. The rationale is that these registers typically hold intermediate results and using the extended format improves the accuracy of the final result.

The status and contents of each register is indicated by a 2-bit tag field. Since we have eight registers, we need a total of 16 tag bits. These 16 bits are stored in the tag register (see Figure 18.1). We discuss the tag register details a little later.

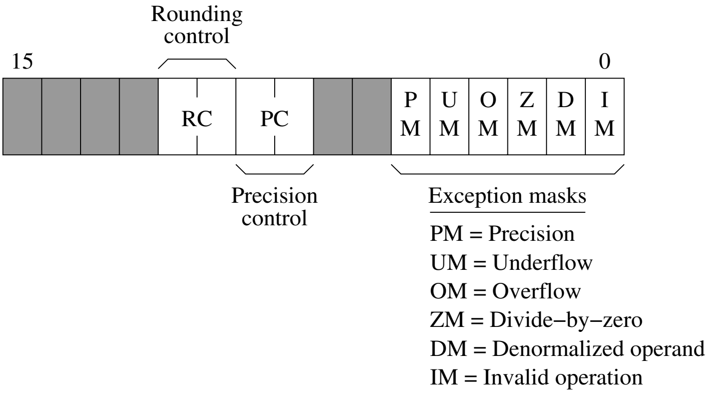


Figure 18.2 FPU control register details (the shaded bits are not used).

18.2.2 Control and Status Registers

This group consists of three 16-bit registers: the control register, the status register, and the tag register, as shown in Figure 18.1.

FPU Control Register This register is used to provide control to the programmer on several processing options. Details about the control word are given in Figure 18.2. The least significant six bits contain masks for the six floating-point exceptions. The PC and RC bits control precision and rounding. Each uses two bits to specify four possible controls. The options for the rounding control are

- 00 — Round to nearest
- 01 — Round down
- 10 — Round up
- 11 — Truncate

The precision control can used to set the internal operating precision to less than the default precision. These bits are provided for compatibility to earlier FPUs with less precision. The options for precision are

- 00 — 24 bits (single precision)
- 01 — Not used
- 10 — 53 bits (double precision)
- 11 — 64 bits (extended precision)

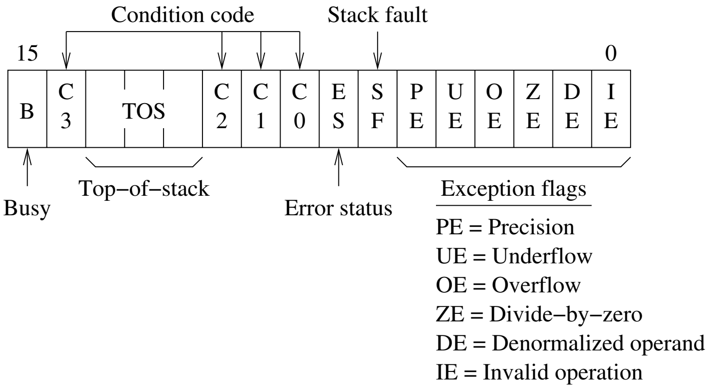


Figure 18.3 FPU status register details. The busy bit is included for 8087 compatibility only.

FPU Status Register This 16-bit register keeps the status of the FPU (see Figure 18.3). The four condition code bits (C0 − C3) are updated to reflect the result of the floating-point arithmetic operations. These bits are similar to the flags register. The correspondence between three of these four bits and the flag register is shown below:

FPU flag	**CPU flag**
C0	CF
C2	PF
C3	ZF

The missing C1 bit is used to indicate stack underflow/overflow (discussed below). These bits are used for conditional branching just like the corresponding CPU flag bits.

To facilitate this branching, the status word should be copied into the CPU flags register. This copying is a two-step process. First, we use the `fstsw` instruction to store the status word in the AX register. We can then load these values into the flags register by using the `sahf` instruction. Once loaded, we can use conditional jump instructions. We demonstrate an application of this in Example 18.2.

The status register uses three bits to maintain the top-of-stack (TOS) information. The eight floating-point registers are organized as a circular buffer. The TOS identifies the register that is at the top. Like the CPU stack, this value is updated as we push and pop from the stack.

The least significant six bits give the status of the six exceptions shown in Figure 18.3. The invalid operation exception may occur due to either a stack operation or an arithmetic operation. The stack fault bit gives information as to the cause of the invalid operation. If this bit is 1, the stack fault is caused by a stack operation that resulted in a stack overflow or underflow condition; otherwise, the stack fault is due to an arithmetic instruction encountering

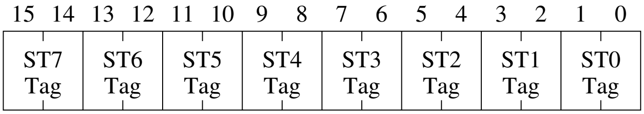


Figure 18.4 FPU tag register details.

an invalid operand. We can use the C1 bit to further distinguish between the stack underflow (C1 = 0) and overflow (C1 = 1).

The overflow and underflow exceptions occur if the number is too big or too small. These exceptions usually occur when we execute floating-point arithmetic instructions.

The precision exception indicates that the result of an operation could not be represented exactly. This, for example, would be the case when we want to represent a fraction like 1/3. This exception indicates that we lost some accuracy in representing the result. In most cases, this loss of accuracy is acceptable.

The divide-by-zero exception is similar to the divide error exception generated by the processor (see our discussion on page 408). The denormal exception is generated when an arithmetic instruction attempts to operate on a denormal operand (denormals are explained later—see page 518).

Tag Register This register stores information on the status and content of the data registers. The tag register details are shown in Figure 18.4. For each register, two bits are used to give the following information:

- 00 — valid
- 01 — zero
- 10 — special (invalid, infinity, or denormal)
- 11 — empty

The least significant two bits are used for the ST0 register, and the next two bits for the ST1 register, and so on. This tag field identifies whether the associated register is empty or not. If not empty, it identifies the contents: valid number, zero, or some special value like infinity.

18.3 Floating-Point Instructions

The FPU provides several floating-point instructions for data movement, arithmetic, comparison, and transcendental operations. In addition, there are instructions for loading frequently used constants like π as well as processor control words. In this section we look at some of these instructions.

Unless otherwise specified, these instructions affect the four FPU flag bits as follows: the flag bits C0, C2, and C3 are undefined; the C1 flag is updated as described before to indicate the stack overflow/underflow condition. Most instructions we discuss next, except the compare instructions, affect the flags this way.

18.3.1 Data Movement

Data movement is supported by two types of instructions: load and store. We start our discussion with the load instructions. The general load instruction has the following format:

```
fld    src
```

This instruction pushes `src` onto the FPU stack. That is, it decrements the TOS pointer and stores `src` at ST0. The `src` operand can be in a register or in memory. If the source operand is in memory, it can be a single-precision (32-bit), double-precision (64-bit), or extended (80-bit) floating-point number. Since the registers hold the numbers in the extended format, a single- or double-precision number is converted to the extended format before storing it in ST0.

There are also instructions to push constants onto the stack. These instructions do not take any operands. Here is a list of these instructions:

Instruction	**Description**
`fldz`	Push +0.0 onto the stack
`fld1`	Push +1.0 onto the stack
`fldpi`	Push π onto the stack
`fldl2t`	Push $\log_2 10$ onto the stack
`fldl2e`	Push $\log_2 e$ onto the stack
`fldlg2`	Push $\log_{10} 2$ onto the stack
`fldln2`	Push $\log_e 2$ onto the stack

To load an integer, we can use

```
fild    src
```

The `src` operand must be a 16- or 32-bit integer located in memory. The instruction converts the integer to the extended format and pushes onto the stack (i.e., loads in ST0).

The store instruction has the following format:

```
fst    dest
```

It stores the top-of-stack values at `dest`. The destination can be one of the FPU registers or memory. Like the load instruction, the memory operand can be single-precision, double-precision, or extended floating-point number. As usual, if the destination is a single- or double-precision operand, the register value is converted to the destination format. It is important to note this instruction does not remove the value from the stack; it simply copies its value. If you want the value to be copied as well as pop it off the stack, use the following instruction (i.e., use the suffix p):

```
fstp    dest
```

There is an integer version of the store instruction. The instruction

```
fist    dest
```

converts the value in ST0 to a signed integer and stores it at dest in memory. It uses the RC (rounding control) field in the conversion (see the available rounding options on page 510).

The pop version of this instruction

```
fistp    dest
```

performs similar conversion as the fist instruction; the difference is that it also pops the value from the stack.

18.3.2 Addition

The basic add instruction has the following format:

```
fadd    src
```

It adds the floating-point number in memory (at src) to that in ST0 and stores the result back in ST0. The value at src can be a single- or double-precision number. This instruction does not pop the stack.

The two-operand version of the instruction allows us to specify the destination register:

```
fadd    dest,src
```

In this instruction, both src and dest must be FPU registers. Like the last add instruction, it does not pop the stack. For this, you have to use the pop version:

```
faddp    dest,src
```

We can add integers using the following instruction:

```
fiadd    src
```

Here src is a memory operand that is either a 16- or 32-bit integer.

18.3.3 Subtraction

The subtract instruction has a similar instruction format as the add instruction. The subtract instruction

```
fsub    src
```

performs the following operation:

```
ST0 = ST0−src
```

Like the add instruction, we can use the two-operand version to specify two registers. The instruction

```
fsub    dest,src
```

performs `dest` = `dest−src`. We can also have a pop version of this instruction:

```
fsubp    dest,src
```

Since subtraction is not commutative (i.e., $A - B$ is not the same as $B - A$), there is a reverse subtract operation. It is reverse in the sense that operands of this instruction are reversed from the previous subtract instructions. The instruction

```
fsubr    src
```

performs the operation `ST0` = `src−ST0`. Note that the `fsub` performs `ST0−src`. Now you know why this instruction is called the reverse subtract! Like the `fsub` instruction, there is a two-operand version as well as a pop version (for the pop version, use `fsubrp` opcode).

If you want to subtract an integer, you can use `fisub` for the standard subtraction, or `fisubr` for reverse subtraction. As in the `fiadd` instruction, the 16- or 32-bit integer must be in memory.

18.3.4 Multiplication

The multiplication instruction has several versions similar to the `fadd` instruction. We start with the memory operand version:

```
fmul    src
```

where the source (`src`) can a 32- or 64-bit floating-point number in memory. It multiplies this value with that in ST0 and stores the result in ST0.

As in the add and subtract instructions, we can use the two-operand version to specify two registers. The instruction

```
fmul    dest,src
```

performs `dest` = `dest` ∗ `src`. The pop version of this instruction is also available:

```
fmulp    dest,src
```

There is also a special pop version that does not take any operands. The operands are assumed to be the top two values on the stack. The instruction

```
fmulp
```

is similar to the last one except that it multiplies ST0 and ST1.

To multiply the contents of ST0 by an integer stored in memory, we can use

```
fimul    src
```

The value at `src` can be a 32- or 64-bit integer.

18.3.5 Division

The division instruction has several versions like the subtract instruction. The memory version of the divide instruction is

```
fdiv    src
```

It divides the contents of ST0 by `src` and stores the result in ST0:

```
ST0 = ST0/src
```

The `src` operand can be a single- or double-precision floating-point value in memory.

The two-operand version

```
fdiv    dest,src
```

performs `dest = dest/src`. As in the previous instructions, both operands must be in the floating-point registers. The pop version uses `fdivp` instead of `fdiv`. To divide ST0 by an integer, use the `fidiv` instruction.

Like the subtract instruction, there is a reverse variation for each of these divide instructions. The rationale is simple: A/B is not the same as B/A. For example, the reverse divide instruction

```
fdivr    src
```

performs

```
ST0 = src/ST0
```

As shown in this instruction, we get the reverse version by suffixing `r` to the opcode.

18.3.6 Comparison

This instruction can be used to compare two floating-point numbers. The format is

```
fcom    src
```

It compares the value in ST0 with `src` and sets the FPU flags. The `src` operand can be in memory or in a register. As mentioned before, the C1 bit is used to indicate stack overflow/underflow condition. The other three flags—C0, C2, and C3—are used to indicate the relationship as follows:

ST0 > `src`	C3 C2 C0 = 0 0 0
ST0 = `src`	C3 C2 C0 = 1 0 0
ST0 < `src`	C3 C2 C0 = 0 0 1
Not comparable	C3 C2 C0 = 1 1 1

If no operand is given in the instruction, the top two values are compared (i.e., ST0 is compared with ST1). The pop version is also available (`fcomp`).

The compare instruction also comes in a double-pop flavor. The instruction

```
fcompp
```

takes no operands. It compares ST0 with ST1 and updates the FPU flags as discussed before. In addition, it pops the two values off the stack, effectively removing the two numbers it just compared.

To compare the top of stack with an integer value in memory, we can use

```
ficom    src
```

The `src` can be a 16- or 32-bit integer. There is also the pop version of this instruction (`ficomp`).

A special case of comparison that is often required is the comparison with zero. The instruction

```
ftst
```

can used for this purpose. It takes no operands and compares the stack top value to 0.0 and updates the FPU flags as in the `fcmp` instruction.

The last instruction we discuss here allows us to examine the type of number. The instruction

```
fxam
```

examines the number in ST0 and returns its sign in C1 flag bit (0 for positive and 1 for negative). In addition, it returns the following information in the remaining three flag bits (C0, C2, and C3):

Type	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal	1	1	0

The *unsupported* type is a format that is not part of the IEEE 754 standard. The *NaN* represents Not-a-Number, as discussed in Appendix A. The meaning of *Normal*, *Infinity*, and *Zero* does not require an explanation. A register that does not have a number is identified as *Empty*.

Denormals are used for numbers that are very close to zero. Recall that normalized numbers have 1.XX...XX as the mantissa. In single- and double-precision numbers, the integer 1 is not explicitly stored (it is implied to save a bit). Thus, we store only XX...XX in mantissa. This integer bit is explicitly stored in the extended format.

When the number is very close to zero, we may underflow the exponent when we try to normalize it. Therefore, in this case, we leave the integer bit as zero. Thus, a denormal has the following two properties:

- The exponent is zero;
- The integer bit of the mantissa is also a zero.

18.3.7 Miscellaneous

We now give details on some of the remaining floating-point instructions. Note that there are several other instructions that are not covered in our discussion here. The NASM manual gives a complete list of the floating-point instructions implemented in NASM.

The instruction

```
fchs
```

changes the sign of the number in ST0. We use this instruction in our quadratic roots example to invert the sign. A related instruction

```
fabs
```

replaces the value in ST0 with its absolute value.

Two instructions are available for loading and storing the control word. The instruction

```
fldcw    src
```

loads the 16-bit value in memory at `src` into the FPU control word register. To store the control word, we use

```
fstcw    dest
```

Following this instruction, all four flag bits (C0 – C3) are undefined.

To store the status word, we can use the instruction

```
fstsw    dest
```

It stores the status word at `dest`. Note that the `dest` can be a 16-bit memory location or the AX register. Combining this instruction with `sahf`, which copies AH into the processor flags register, gives us the ability to use the conditional jump instructions. We use these two instructions in the quadratic roots example given in the next section. After executing this instruction, all four flag bits (C0 – C3) are undefined.

18.4 Illustrative Examples

To illustrate the application of the floating-point instructions, we give some examples here. We use mixed-mode programs in our examples. To follow these examples, you need to understand the material presented in the last chapter. The first two examples use separate assembly language modules. The last example uses inline assembly code.

Example 18.1 *Array sum example.*

This example computes the sum of an array of doubles. The C program, shown in Program 18.1, takes care of the user interface. It requests values to fill the array and then calls the assembly language procedure `array_fsum` to compute the sum.

The `array_fsum` procedure is given in Program 18.2. It copies the array pointer to EDX (line 10) and the array size to ECX (line 11). We initialize ST0 to zero by using the `fldz` instruction on line 12. The add loop consists of the code on lines 13–17. We use the `jecxz` instruction to exit the loop if the index is zero at the start of the loop.

We use the `fadd` instruction to compute the sum in ST0. Also note that the based-indexed addressing mode with a scale factor of 8 is used to read the array elements (line 16). Since C programs expect floating-point return values in ST0, we simply return from the procedure as the result is already in ST0.

Program 18.1 Array sum program—C program

```
/************************************************************
 * This program reads SIZE values into an array and calls
 * an assembly language program to compute the array sum.
 * The assembly program is in the file "arrayfsuma.asm".
 ************************************************************/
#include        <stdio.h>

```

```
#define  SIZE  10

int main(void)
{
    double     value[SIZE];
    int        i;
    extern double array_fsum(double*, int);

    printf("Input %d array values:\n", SIZE);
    for (i = 0; i < SIZE; i++)
        scanf("%lf",&value[i]);

    printf("Array sum = %lf\n", array_fsum(value,SIZE));

    return 0;
}
```

Program 18.2 Array sum program—assembly language procedure

```
;-------------------------------------------------------------
; This procedure receives an array pointer and its size via
; the stack. It computes the array sum and returns it via ST0.
;-------------------------------------------------------------
segment .text
global  array_fsum

array_fsum:
      enter   0,0
      mov     EDX,[EBP+8]       ; copy array pointer to EDX
      mov     ECX,[EBP+12]      ; copy array size to ECX
      fldz                      ; ST0 = 0 (ST0 keeps the sum)
add_loop:
      jecxz   done
      dec     ECX               ; update array index
      fadd    qword[EDX+ECX*8]  ; ST0 = ST0 + arrary_element
      jmp     add_loop
done:
      leave
      ret
```

Example 18.2 *Quadratic equation solution.*
In this example, we find roots of the quadratic equation

$$ax^2 + bx + c = 0\,.$$

The two roots are defined as follows:

$$\text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\,,$$

$$\text{root2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,.$$

The roots are real if $b^2 \geq 4ac$, and imaginary otherwise.

As in the last example, our C program takes care of the user interface (see Program 18.3). It requests the user to input constants `a`, `b`, and `c`. It then passes these three values to the `quad_roots` assembly language procedure along with two pointers to `root1` and `root2`. This procedure returns 0 if the roots are not real; otherwise it returns 1. If the roots are real, the two roots are returned in `root1` and `root2`.

The assembly language procedure, shown in Program 18.4, receives five arguments: three constants and two pointers to return the two roots. These five arguments are assigned convenient labels on lines 7–11. The comments included in the code make it easy to follow the body of the procedure. On each line, we indicate the contents on the stack with the leftmost value being at the top of the stack.

We use the `ftst` instruction to see if $(b^2 - 4ac)$ is negative (line 30). We move the FPU flag bits to AX and then to the processor flags register using the `fstsw` and `sahf` instructions on lines 31 and 32. Once these bits are copied into the flags register, we can use the conditional jump instruction `jb` (line 33). The rest of the procedure body is straightforward to follow.

Program 18.3 Quadratic equation solution—C program

```
/************************************************************
 * This program reads three constants (a, b, c) and calls an
 * assembly language program to compute the roots of the
 * quadratic equation.
 * The assembly program is in the file "quada.asm".
 ************************************************************/
#include        <stdio.h>

int main(void)
{
    double     a, b, c, root1, root2;
    extern int quad_roots(double, double, double, double*, double*);

    printf("Enter quad constants a, b, c: ");
    scanf("%lf %lf %lf",&a, &b, &c);

    if (quad_roots(a, b, c, &root1, &root2))
        printf("Root1 = %lf and root2 = %lf\n", root1, root2);
    else
        printf("There are no real roots.\n");

    return 0;
}
```

Program 18.4 Quadratic equation solution—assembly language procedure

```
;-------------------------------------------------------------
; This procedure receives three constants a, b, c and
; pointers to two roots via the stack. It computes the two
; real roots if they exist and returns them in root1 & root2.
; In this case, EAX = 1. If no real roots exist, EAX = 0.
;-------------------------------------------------------------
%define   a       qword[EBP+8]
%define   b       qword[EBP+16]
%define   c       qword[EBP+24]
%define   root1   dword[EBP+32]
%define   root2   dword[EBP+36]

segment .text
global  quad_roots

quad_roots:
      enter   0,0
      fld     a              ; a
      fadd    ST0            ; 2a
      fld     a              ; a,2a
      fld     c              ; c,a,2a
      fmulp   ST1            ; ac,2a
      fadd    ST0            ; 2ac,2a
      fadd    ST0            ; 4ac,2a
      fchs                   ; -4ac,2a
      fld     b              ; b,-4ac,2a
      fld     b              ; b,b,-4ac,2a
      fmulp   ST1            ; b*b,-4ac,2a
      faddp   ST1            ; b*b-4ac,2a
      ftst                   ; compare (b*b-4ac) with 0
      fstsw   AX             ; store status word in AX
      sahf
      jb      no_real_roots
      fsqrt                  ; sqrt(b*b-4ac),2a
      fld     b              ; b,sqrt(b*b-4ac),2a
      fchs                   ; -b,sqrt(b*b-4ac),2a
      fadd    ST1            ; -b+sqrt(b*b-4ac),sqrt(b*b-4ac),2a
      fdiv    ST2            ; -b+sqrt(b*b-4ac)/2a,sqrt(b*b-4ac),2a
      mov     EAX,root1
      fstp    qword[EAX]     ; store root1
      fchs                   ; -sqrt(b*b-4ac),2a
      fld     b              ; b,sqrt(b*b-4ac),2a
      fsubp   ST1            ; -b-sqrt(b*b-4ac),2a
      fdivrp  ST1            ; -b-sqrt(b*b-4ac)/2a
      mov     EAX,root2
      fstp    qword[EAX]     ; store root2
      mov     EAX,1          ; real roots exist
      jmp     short done
no_real_roots:
      sub     EAX,EAX        ; EAX = 0 (no real roots)
done:
      leave
      ret
```

Example 18.3 *Array sum example—inline version.*
In this example we rewrite the code for the `array_fsum` procedure using the inline assembly method. Remember that when we use this method, we have to use AT&T syntax. In this syntax, the operand size is explicitly indicated by suffixing a letter to the opcode. For the floating-point instructions, the following suffixes are used:

s	Single-precision
l	Double-precision
t	Extended-precision

The inline assembly code, shown in Program 18.5, is similar to that in Program 18.2. You will notice that on line 10 we use `=t` output specifier to indicate that variable `sum` is mapped to a floating-point register (see page 499 for a discussion of these specifiers). Since we map `value` to EBX and `size` to ECX (line 11), we use these registers in the assembly language code to access the array elements (see line 7).

Program 18.5 Array sum example—inline version

```
double array_fsum(double* value, int size)
{
     double sum;
     asm("          fldz;                    "  /* sum = 0 */
         "add_loop: jecxz   done;            "
         "          decl    %%ecx;           "
         "          faddl   (%%ebx,%%ecx,8);"
         "          jmp     add_loop;        "
         "done:                              "
          :"=t"(sum)                  /* output */
          :"b"(value),"c"(size)       /* inputs */
          :"cc");                     /* clobber list */
     return(sum);
}
```

18.5 Summary

We presented a brief description of the floating-point unit organization. Specifically, we concentrated on the registers provided by the FPU. It provides eight floating-point data registers that are organized as a stack. The floating-point instructions include several arithmetic and nonarithmetic instructions. We discussed some of these instructions. Finally, we presented some examples that used the floating-point instructions discussed.

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:

- Control register
- Data registers
- Denormals
- Floating-point instructions
- Precision control
- Rounding control
- Status register
- Tag register

18.6 Exercises

18–1 What is the purpose of the TOS field in the status register?

18–2 What is the purpose of the condition code bits in the status register?

18–3 Why do we need to keep the tag field information on each data register?

18–4 We stated that the floating-point data registers use the extended format. What is the motivation for this selection?

18–5 What is a denormal floating-point number? Why do we need it?

18–6 Explain how we can control the program flow based on the result of the `fcom` instruction.

18.7 Programming Exercises

18–P1 Write a mixed-mode program that reads a set of marks and computes the average mark. The main procedure (written in C) handles the user interface. It reads the input marks (as `double` values) into an array and passes the array and its size to the assembly language procedure `favg`. This procedure computes the average mark and returns it to the main procedure. The main procedure prints the average value.

18–P2 Modify the last program to return minimum and maximum marks as well. The assembly language procedure receives a pointer to the marks array, its size, and three pointers to return the average, minimum, and maximum marks.

18–P3 In Programming Exercise 18–P1, you were asked to write the assembly language procedure (`favg`) as a separate module. Rewrite this function in inline assembly.

18–P4 Write a mixed-mode program that reads the radius (r) of a circle and computes its area. The main procedure (written in C) handles the user interface. It reads the radius (as a `double`) and passes it to the assembly language procedure `farea`. This procedure computes the area (πr^2) and returns it to the main procedure. The main procedure prints the area.

18–P5 Rewrite the function `farea` in inline assembly.

Appendices

This part consists of seven appendices, which provide a wealth of reference material. Appendix A primarily discusses the number systems and their internal representation. Appendix B gives information on the use of I/O routines provided with this book and the assembler software. The debugging aspect of assembly language programming is discussed in Appendix C. The SPIM simulator details are given in Appendix D. Selected Pentium and MIPS instructions are given in Appendices E and F, respectively. Finally, Appendix G gives the standard ASCII table.

Appendix A

Internal Data Representation

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
- To describe character 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 error-prone. 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 530; 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.

$$\begin{aligned}10100111B &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 \\ &\qquad\qquad + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 167D.\end{aligned}$$

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 $\times$ $b$ ) $+$ $d_i$
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 $\neq$ 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 532. □

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 $d_n d_{n-1} \ldots d_0$ 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 533, 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 533). □

Table A.1 Three-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

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.

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.

$$\begin{aligned} 10100111\text{B} &= \overbrace{\mathbf{0}10}^{2}\,\overbrace{100}^{4}\,\overbrace{111}^{7}\,\text{B} \\ &= 247\text{Q}\,. \end{aligned}$$

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.

Table A.2 Four-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

Example A.8 *Conversion from octal to binary.*

The following two examples illustrate conversion from octal to binary:

$$105\text{Q} = \overbrace{001}^{1}\ \overbrace{000}^{0}\ \overbrace{101}^{5}\text{B},$$

$$247\text{Q} = \overbrace{010}^{2}\ \overbrace{100}^{4}\ \overbrace{111}^{7}\text{B}.$$

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.

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.*

$$\begin{array}{rl} & \quad\texttt{001110} \leftarrow \mathrm{C_{in}} \\ \texttt{174D} & = \texttt{10101110B} \\ \underline{\texttt{75D}} & = \underline{\texttt{01001011B}} \\ \texttt{249D} & = \texttt{11111001B} \end{array}$$

In this example, there is no overflow. □

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.

$$\begin{array}{rcl} & & \overset{\substack{\text{indicates}\\ \text{overflow}\\ \downarrow}}{\mathbf{1}}\texttt{1111110} \leftarrow \mathrm{C_{in}} \\ \texttt{174D} & = & \texttt{10101110B} \\ \underline{\texttt{91D}} & = & \underline{\texttt{01011011B}} \\ \texttt{265D} & \neq & \texttt{00001001B} \end{array}$$

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.

$$\begin{array}{rl} & \texttt{1111110} \leftarrow \mathrm{B_{in}} \\ \texttt{201D} & = \texttt{11001001B} \\ \underline{\texttt{110D}} & = \underline{\texttt{01101110B}} \\ \texttt{91D} & = \texttt{01011011B} \end{array}$$

□

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
            ↓
            111111110 ← 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. □

Example A.16 *Binary multiplication of unsigned integers.*

Binary multiplication follows exactly the same procedure except that the base-2 arithmetic is performed, as shown in the next example.

```
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 $\geq$ 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

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

- Excess-M
- 1's complement
- 2's complement

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.

1010B ⟶complement bits⟶ 0101 = 5D.

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:

$$\begin{array}{rcrl}
\texttt{+5D} & = & \texttt{0101B} & \\
\underline{\texttt{+3D}} & = & \underline{\texttt{0011B}} & \\
\texttt{+8D} & \neq & \texttt{1000B} & (= \ -7\text{D})
\end{array}
\qquad\qquad
\begin{array}{rcrl}
\texttt{-5D} & = & \texttt{1010B} & \\
\texttt{-4D} & = & \underline{\texttt{1011B}} & \\
 & & \texttt{10101B} & \\
 & & \hookrightarrow \ \texttt{1} & \\
\overline{\texttt{-9D}} & \neq & \overline{\texttt{0110B}} & (= +6\text{D})
\end{array}$$

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 the 1's complement representation of -7, and add this to 4.

$$\begin{array}{lr}
\texttt{+4D} = \texttt{0100B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{0100B} \\
\underline{\texttt{-7D}} = \texttt{0111B} \xrightarrow{1' \text{s complement}} & \underline{\texttt{1000B}} \\
\texttt{-3D} = & \texttt{1100B}
\end{array}$$

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 $5 - (-3)$.

$$\begin{array}{llr}
\texttt{+5D} & = \texttt{0101B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{0101B} \\
\underline{\texttt{-(-3D)}} & = \texttt{1100B} \xrightarrow{1' \text{s complement}} & \underline{\texttt{0011B}} \\
\texttt{+8D} & \neq & \texttt{1000B}
\end{array}$$

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 $-5 - (3)$.

$$\begin{array}{lll}
\texttt{-5D} & = \texttt{1010B}\longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{1010B} \\
\texttt{-(+3D)} & = \texttt{0011B}\xrightarrow{1'\text{s complement}}\longrightarrow\longrightarrow\longrightarrow & \underline{\texttt{1100B}} \\
 & & \texttt{10110B} \\
 & & \hookrightarrow \underline{\quad\texttt{1}} \\
\overline{\texttt{-8D}} & \neq & \texttt{0111B}
\end{array}$$

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.

$$\begin{array}{llr}
\texttt{6D} = \texttt{0110B}\longrightarrow & \text{complement } \longrightarrow & \texttt{1001B} \\
 & \text{add 1} & \underline{\texttt{1B}} \\
 & & \texttt{1010B}
\end{array}$$

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.

```
1010B⟶ complement ⟶0101B
       add 1           1B
                   -----
                   0110B   (= 6D)
```

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
+2D = 0010B
---   -----
+7D = 0111B
```

```
+5D =  0101B
-2D =  1110B
---   ------
+3D   10011B
```

Discarding the carry leaves the result `0011B` = +3D.

```
-5D = 1011B
+2D = 0010B
---   -----
-3D = 1101B
```

```
-5D =  1011B
-2D =  1110B
---   ------
-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

$$0.11001\mathrm{B}$$

the decimal value it represents is

$$1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 0.78125\mathrm{D}.$$

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 $0.d_1d_2 \ldots d_{n-1}d_n$ 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.

$$0.\underbrace{1100}_{12=\text{C}}\underbrace{1\mathbf{000}}_{8} = 0.\text{C8H}\,.$$

We have to add three 0s on the right to make the number of bits equal to 8 (that is, 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 $\times$ $b$)
    value = fraction (value $\times$ $b$)
    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 558.

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 one bit to four bits, the range decreases approximately by half to $0 \leq X <\ 2^4$. 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, is error-prone, 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

$$+1.101101\text{E}11101.$$

To represent such normalized numbers, we might use the format shown below:

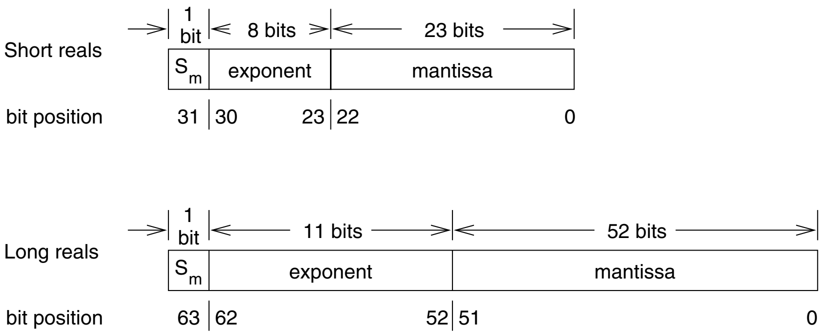
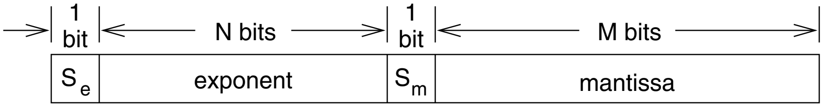


Figure A.1 Floating-point formats.



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 in Figure A.1. 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 534).
1b: Convert the fractional part to binary using the procedure described in Section A.5.1 (page 554).
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.

Next we give an example to illustrate the above procedure.

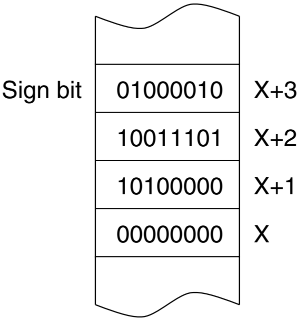
Example A.33 *Conversion to floating-point format.*

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.

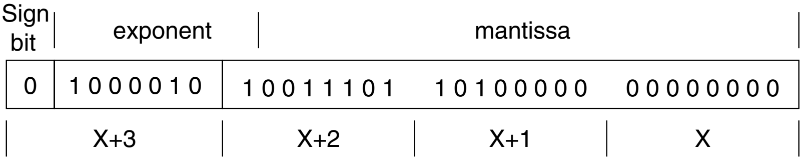
Step 4: Separate the three components.
Sign: 0 (positive number)
mantissa: 0011101101
(1 to the left of the binary point is implied)
exponent: 10000101.

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:



Sign bit	01000010	X+3
	10011101	X+2
	10100000	X+1
	00000000	X

If we lay these four bytes linearly, they look like this:



Sign bit	exponent	mantissa		
0	1000010	10011101	10100000	00000000
X+3		X+2	X+1	X

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

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	$\neq 0$

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 the following reference:

D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys,* Vol. 23, No. 1, March 1991, pp. 5–48.

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 Character Representation

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

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 the 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 (linefeed), etc.

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 1's 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 eight 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, etc. make up the additional 128 characters. Appendix G shows the standard as well as the extended ASCII character codes.

You will notice from the table in Appendix G 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 four bits represent the binary equivalent of the corresponding digit. For example, digit 5 is encoded as 0110101. If you take the rightmost four 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.

A.7 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.

The last section discussed character representation. We identified some desirable properties that a character encoding scheme should satisfy in order to facilitate efficient character processing. While there are two character codes—EBCDIC and ASCII—most computers including the IBM PC use ASCII. We noted that ASCII satisfies the requirements of an efficient character code.

A.8 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,
(b) 11010011B,
(c) AB15H,
(d) 1234H,
(e) 1234Q,
(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 answers by converting your answers 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.9 Programming Exercises

A–P1 Implement the algorithm on page 533 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 534 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 557 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

Assembling and Linking

Objectives

- To give details on NASM assembler installation
- 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 start our discussion with details on the NASM assembler. The next section looks at the structure of assembly language programs that we use in this book. Unlike high-level languages, the assembly language does not provide a convenient mechanism to do input/output. To overcome this deficiency, we developed a set of I/O routines to facilitate character, string, and numeric input/output. These routines are described in Section B.3.

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 version. Section B.4 gives details of these steps. The appendix concludes with a summary.

B.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 (see Section B.6) has clear instructions on how to install NASM under Linux. Here is a summary extracted from this 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. Alternatively, you can use an RPM distribution for the Red Hat Linux. This version is simpler to install—just double-click the RPM file.

NASM can support several object file formats including the ELF (execute and link format) format used by Linux. The assembling and linking process is simple. For example, to assemble `addigits.asm`, we use

```
nasm -f elf addigits.asm
```

This generates the `addigits.o` object file. 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 these two files in your current directory. We give details about these files in the next section.

B.2 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 those details that are irrelevant. We achieve this by (1) providing special I/O routines and (2) defining a basic assembly language template.

Facilitating Input/Output

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.

The 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.

In order to facilitate I/O in assembly language programs, it is necessary to write the required procedures. We developed a set of I/O routines to read and display signed integers, characters, and strings. 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 I/O procedure. These two functions are separated into two I/O files:

- The `io.mac` file contains the macro definitions for the I/O routines. This file is included in our assembly program by using the `%include` directive (see Figure B.1),
- The `io.o` contains the I/O procedures that actually perform the operation. This file is needed by the linker (discussed later).

The next section gives details about these routines.

Assembly Language Template

To simplify writing assembly language programs, we use the template shown in Figure B.1. As mentioned before, we include the `io.mac` file by using the `%include` directive. This directive allows us to include the contents of `io.mac` in the assembly language program. If you had used other assemblers like TASM or MASM, it is important to note that NASM is case-sensitive.

The data part is split into two: the `.DATA` macro is used for initialized data and the `.UDATA` for uninitialized data. The code part is identified by the `.CODE` macro. The `.STARTUP` macro handles the code for setup. The `.EXIT` macro returns control to the operating system.

B.3 Input/Output Routines

The I/O routines facilitate reading and displaying characters, strings, and integers. We now describe these routines in detail. Table B.1 provides a summary of the I/O routines defined in `io.mac`.

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, or a byte in memory, or a character value. Some examples follow:

```
;brief title of program             file name
;
;            Objectives:
;                Inputs:
;               Outputs:
;
%include  "io.mac"

.DATA
 (initialized data go here)

.UDATA
 (uninitialized data go here)

.CODE
        .STARTUP            ; setup
          . . .
          . . .
    (code goes here)
          . . .
          . . .
        .EXIT               ; returns control
```

Figure B.1 Template for the assembly language programs used in the book.

```
PutCh    'A'        ; displays character A
PutCh    AL         ; displays the character in AL
PutCh    [response] ; displays the byte located in
                    ; memory (labeled response)
```

Note that the memory operands should be in [ ]. The format of `GetCh` is

```
GetCh    destination
```

where destination can be either an 8-bit register or a byte in memory. Some examples are

```
GetCh    DH
GetCh    [response]
```

In addition, a `nwln` macro is defined to display a newline (e.g., `\n` in C). It takes no operands.

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 must be

Table B.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

the NULL ASCII character, which signals the end of the string. Strings are discussed in Chapter 10.

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`. If the buffer does not contain a NULL-terminated string, a maximum of 80 characters are displayed.

The format of `GetStr` is

```
GetStr    destination [,buffer_size]
```

where destination is the buffer name in which the string is stored. The input string can be terminated by `return`. 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 the input. Here are some examples:

```
GetStr    in_string    ; reads at most 80 characters
GetStr    TR_title,41  ; reads at most 40 characters
```

Numeric I/O

There are four macros to perform integer I/O: two are defined for 16-bit integers and the remaining 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 the source and destination can be a 16-bit register or a memory word.

`PutInt` displays the signed number at the source. It suppresses all leading 0's. `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

```
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 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

```
PutLint   EAX
GetLint   EDX
```

An Example

Program B.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 11–15). 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 user name, and another byte is reserved for response (lines 18 and 19). In both cases, we use `resb` to reserve space for uninitialized data.

Program B.1 An example assembly program

```
 1:  ;An example assembly language program     SAMPLE.ASM
 2:  ;
 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:  %include  "io.mac"
 9:
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:  .UDATA
18:  user_name    resb  16                ; buffer for user name
19:  response     resb  1
20:
21:  .CODE
22:        .STARTUP
23:        PutStr  name_msg               ; prompt user for his/her name
24:        GetStr  user_name,16           ; read name (max. 15 characters)
25:  ask_count:
26:        PutStr  query_msg              ; prompt for repeat count
27:        GetInt  CX                     ; read repeat count
28:        PutStr  confirm_msg1           ; confirm repeat count
29:        PutInt  CX                     ; by displaying its value
30:        PutStr  confirm_msg2
31:        GetCh   [response]             ; read user response
32:        cmp     byte [response],’y’    ; if ’y’, display welcome message
```

```
33:        jne     ask_count             ; otherwise, request repeat count
34:  display_msg:
35:        PutStr  welcome_msg           ; display welcome message
36:        PutStr  user_name             ; display the user name
37:        nwln
38:        loop    display_msg           ; repeat count times
39:        .EXIT
```

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 23 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 24). The PutStr on line 26 requests a repeat count, which is read by GetInt on line 27. The confirmation message is displayed by lines 28–30. The response of the user y/n is read by GetCh on line 31. If the response is y, the loop (lines 34–38) 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? 5
Repeat welcome message 5 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
Welcome to Assembly Language Programming Veda
```

B.4 Assembling and Linking

Figure B.2 shows the steps involved in converting an assembly language program into an executable code. 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.o). The linker takes one or more object programs (e.g., sample.o and io.o) and combines them into an executable program (e.g., sample). The following subsections describe each of these steps in detail.

B.4.1 The Assembly Process

The general format to assemble a program is

```
nasm  -f <format> <source-file> [-o <object-file>][-l <list-file>]
```

where the specification of fields in [ ] is optional. If we specify only the source file, NASM

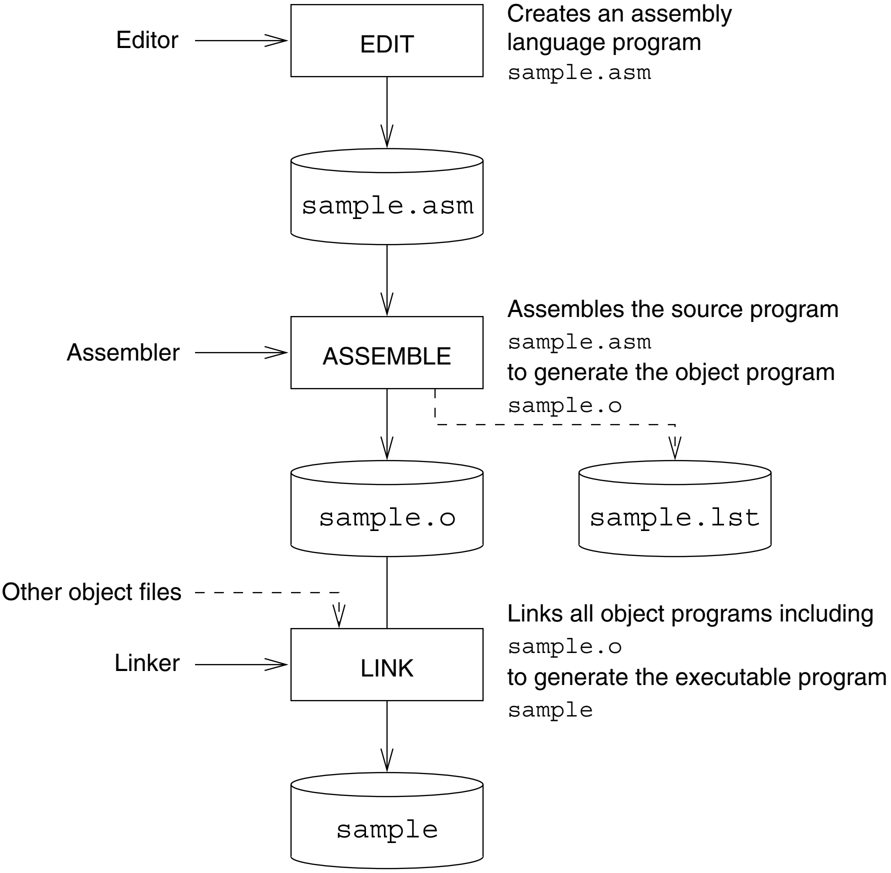

Editor
EDIT
Creates an assembly language program sample.asm
sample.asm
Assembler
ASSEMBLE
Assembles the source program sample.asm to generate the object program sample.o
sample.o
sample.lst
Other object files
Linker
LINK
Links all object programs including sample.o to generate the executable program sample
sample

Figure B.2 Assembling and linking assembly language programs (optional inputs and outputs are shown by dashed lines).

produces only the object file. Thus to assemble our example source file `sample.asm`, we can use the command

```
nasm -f elf  sample.asm
```

After successfully assembling the source program, NASM generates an object file with the same file name as the source file but with `.o` extension. Thus, in our example, it generates the `sample.o` file.

If you want the assembler to generate the listing file, you can use

```
nasm  -f elf sample.asm -l sample.lst
```

This command produces two files: `sample.o` and `sample.lst`. The list file contains detailed information as we shall see next.

The List File

Program B.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 22, 23 and 26, 27). The sum of the input numbers is computed on lines 30–32.

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 36). If there is no overflow, the sum is displayed (lines 42 and 43).

Program B.2 An assembly language program to add two integers `sumprog.asm`

```
 1:  ;Assembly language program to find sum   SUMPROG.ASM
 2:  ;
 3:  ;         Objective: To add two integers.
 4:  ;            Inputs: Two integers.
 5:  ;            Output: Sum of input numbers.
 6:  %include  "io.mac"
 7:
 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:  .UDATA
15:  number1      resd  1        ; stores first number
16:  number2      resd  1        ; stores first number
17:  sum          resd  1        ; stores sum
18:
19:  .CODE
20:        .STARTUP
21:        ; prompt user for first number
22:        PutStr  prompt1_msg
23:        GetLInt [number1]
24:
25:        ; prompt user for second number
26:        PutStr  prompt2_msg
27:        GetLInt [number2]
```

```

        ; find sum of two 32-bit numbers
        mov     EAX,[number1]
        add     EAX,[number2]
        mov     [sum],EAX

        ; check for overflow
        jno     no_overflow
        PutStr  error_msg
        nwln
        jmp     done

        ; display sum
no_overflow:
        PutStr  sum_msg
        PutLInt [sum]
        nwln
done:
        .EXIT
```

The list file for the source program `sumprog.asm` is shown in Program B.3. In addition to the original source code lines, it contains a lot of useful information about the results of the assembly. This additional information includes the actual machine code generated for the executable statements and the offset of each statement.

List File Contents

The format of the list file lines is

```
line#  offset  machine-code  nesting-level  source-line
```

`line#`: is 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, etc., as shown in Program B.3.

`offset`: is an 8-digit hexadecimal offset value of the machine code for the source statement. For example, the offset of the first instruction (line 187) is `00000000H`, and that of the add instruction on line 219 is `00000035H`. Source lines such as comments do not generate any offset.

`machine-code`: is the hexadecimal representation of the machine code for the assembly language instruction. For example, the machine language encoding of

```
mov     EAX,[number1]
```

is `A1[00000000]` (line 218) and requires five bytes. The value zero in [ ] is the offset of `number1` in the data segment (see line 173).

Similarly, the machine language encoding of

```
jmp     done
```

is `E91D000000` (line 231), requiring five bytes of memory.

`nesting-level`: is the level of nesting of “include files” and macros.

`source-line`: is a copy of the original source code line. As you can see from Program B.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 is used in the instruction encoding. The actual value is fixed up by the linker after all the object files are combined (for example, `io.o` in our example). You also notice that the macro definitions are expanded. For example, the `PutStr` on line 186 is expanded on lines 187 through 190.

Program B.3 The list file for the example assembly program `sumprog.asm`

```
     1                                  ;Assembly language program to find sum. . .
     2                                  ;
     3                                  ;         Objective: To add two integers.
     4                                  ;            Inputs: Two integers.
     5                                  ;            Output: Sum of input numbers.
     6                                  %include  "io.mac"
     7                              <1> extern   proc_nwln, proc_PutCh, proc_PutStr
     8                              <1> extern   proc_GetStr, proc_GetCh
     9                              <1> extern   proc_PutInt, proc_GetInt
    10                              <1> extern   proc_PutLInt, proc_GetLInt
    11                              <1>
    12                              <1> ;;------------------------------------
    13                              <1> %macro  .STARTUP  0
    14                              <1> ;group dgroup .data .bss
    15                              <1>         global   _start
    16                              <1> _start:
    17                              <1> %endmacro
    18                              <1> ;;------------------------------------
    19                              <1>
    20                              <1>
    21                              <1> ;;------------------------------------
    22                              <1> %macro  .EXIT  0
    23                              <1>         mov    EAX,1
    24                              <1>         xor    EBX,EBX
    25                              <1>         int    0x80
    26                              <1> %endmacro
    27                              <1> ;;------------------------------------
```

```
 28                              <1>
 29                              <1>
 30                              <1> ;;------------------------------------
 31                              <1> %macro  .DATA 0
 32                              <1>         segment .data
 33                              <1> %endmacro
 34                              <1> ;;------------------------------------
 35                              <1>
 36                              <1> ;;------------------------------------
 37                              <1> %macro  .UDATA 0
 38                              <1>         segment .bss
 39                              <1> %endmacro
 40                              <1> ;;------------------------------------

158                                  .DATA
159                              <1>  segment .data
160 00000000 456E74657220666972-     prompt1_msg  db  ’Enter first number: ’,0
161 00000009 7374206E756D626572-
162 00000012 3A2000
163 00000015 456E74657220736563-     prompt2_msg  db  ’Enter second number: ’,0
164 0000001E 6F6E64206E756D6265-
165 00000027 723A2000
166 0000002B 53756D2069733A2000      sum_msg      db  ’Sum is: ’,0
167 00000034 4F766572666C6F7720-     error_msg    db  ’Overflow has occurred!’,0
168 0000003D 686173206F63637572-
169 00000046 7265642100
170
171                                  .UDATA
172                              <1>  segment .bss
173 00000000 <res 00000004>          number1      resd  1   ; stores first number
174 00000004 <res 00000004>          number2      resd  1   ; stores first number
175 00000008 <res 00000004>          sum          resd  1   ; stores sum
176
177                                  .CODE
178                              <1>  segment .data
179                              <1>  segment .bss
180                              <1>  segment .text
181                                        .STARTUP
182                              <1>
183                              <1>  global _start
184                              <1> _start:
185                                        ; prompt user for first number
186                                        PutStr  prompt1_msg
187 00000000 51                  <1>  push ECX
188 00000001 B9[00000000]        <1>  mov ECX,%1
189 00000006 E8(00000000)        <1>  call proc_PutStr
190 0000000B 59                  <1>  pop ECX
191                                        GetLInt [number1]
192                              <1> %ifnidni %1,EAX
193 0000000C 50                  <1>  push EAX
194 0000000D E8(00000000)        <1>  call proc_GetLInt
```

```
195 00000012 A3[00000000]        <1>  mov %1,EAX
196 00000017 58                  <1>  pop EAX
197                              <1> %else
198                              <1>  call proc_GetLInt
199                              <1> %endif
200
201                                        ; prompt user for second number
202                                        PutStr  prompt2_msg
203 00000018 51                  <1>  push ECX
204 00000019 B9[15000000]        <1>  mov ECX,%1
205 0000001E E8(00000000)        <1>  call proc_PutStr
206 00000023 59                  <1>  pop ECX
207                                        GetLInt [number2]
208                              <1> %ifnidni %1,EAX
209 00000024 50                  <1>  push EAX
210 00000025 E8(00000000)        <1>  call proc_GetLInt
211 0000002A A3[04000000]        <1>  mov %1,EAX
212 0000002F 58                  <1>  pop EAX
213                              <1> %else
214                              <1>  call proc_GetLInt
215                              <1> %endif
216
217                                        ; find sum of two 32-bit numbers
218 00000030 A1[00000000]                  mov     EAX,[number1]
219 00000035 0305[04000000]                add     EAX,[number2]
220 0000003B A3[08000000]                  mov     [sum],EAX
221
222                                        ; check for overflow
223 00000040 7116                          jno     no_overflow
224                                        PutStr  error_msg
225 00000042 51                  <1>  push ECX
226 00000043 B9[34000000]        <1>  mov ECX,%1
227 00000048 E8(00000000)        <1>  call proc_PutStr
228 0000004D 59                  <1>  pop ECX
229                                        nwln
230 0000004E E8(00000000)        <1>  call proc_nwln
231 00000053 E91D000000                    jmp     done
232
233                                        ; display sum
234                                    no_overflow:
235                                        PutStr  sum_msg
236 00000058 51                  <1>  push ECX
237 00000059 B9[2B000000]        <1>  mov ECX,%1
238 0000005E E8(00000000)        <1>  call proc_PutStr
239 00000063 59                  <1>  pop ECX
240                                        PutLInt [sum]
241 00000064 50                  <1>  push EAX
242 00000065 A1[08000000]        <1>  mov EAX,%1
243 0000006A E8(00000000)        <1>  call proc_PutLInt
244 0000006F 58                  <1>  pop EAX
245                                        nwln
```

```
246 00000070 E8(00000000)        <1>  call proc_nwln
247                                  done:
248                                        .EXIT
249 00000075 B801000000          <1>  mov EAX,1
250 0000007A 31DB                <1>  xor EBX,EBX
251 0000007C CD80                <1>  int 0x80
```

B.4.2 Linking Object Files

Linker is a program that takes one or more object programs as its input and produces executable code. In our example, since I/O routines are defined separately, we need two object files—`sample.o` and `io.o`—to generate the executable file `sample` (see Figure B.2). To do this, we use the command

```
ld -s -o sample sample.o io.o
```

If you intend to debug your program using `gdb`, you should use the `stabs` option during the assembly in order to export the necessary symbolic information. We discuss this in the next appendix, as it deals with debugging.

B.5 Summary

We presented details about the NASM assembler. We also presented the template used to write standalone assembly language programs. Since the 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. We used simple examples to illustrate the use of these I/O routines in a typical standalone assembly language program.

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 executable code. We used NASM to assemble the programs. Note that NASM produces additional files that provide information on the assembly process. The list file is the one we often use to see the machine code and other details.

B.6 Web Resources

Documentation (including the NASM manual) and download information on NASM are available from `http://sourceforge.net/projects/nasm`.

B.7 Exercises

B–1 In the assembly language program structure used in this book, how are the data and code parts specified?

B–2 What do we mean by a standalone assembly language program?

B–3 What is an assembler? What is the purpose of it?

B–4 What is the function of the linker? What is the input to the linker?

B–5 Why is it necessary to define our own I/O routines?

B–6 What is a NULL-terminated string?

B–7 Why is a buffer size specification necessary in `GerStr` but not in `PutStr`?

B–8 What happens if the buffer size parameter is not specified in `GetStr`?

B–9 What happens if the buffer specified in `PutStr` does not contain a NULL-terminated string?

B–10 What is the range of numbers that `GetInt` can read from the keyboard? Give an explanation for the range.

B–11 Repeat the last exercise for `GetLint`.

B.8 Programming Exercises

B–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.

B–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.

B–P3 Modify `sample.asm` by deliberately introducing errors into the program. Assemble the program and see the type of errors reported by the assembler. Also, generate the corresponding list file and briefly explain its contents.

Appendix C

Debugging Assembly Language Programs

Objectives

- To present basic strategies to debug assembly language programs
- To give information on preparing the assembly language programs for debugging
- To describe the GDB debugger
- To explain the basic features of the DDD

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 C.1 gives a discussion of these strategies. Since you are familiar with debugging high-level language programs, this discussion is rather brief. The next section explains how you can prepare your Pentium assembly language program for symbolic debugging.

Section C.3 discusses the GNU debugger (GDB). This is a command-line debugger. A nice visual interface to it is provided by Dynamic Data Display (DDD). We describe it in Section C.4. We use a simple example to explain some of the commands of GDB (in Section C.3) and DDD (in Section C.4). The appendix concludes with a summary.

C.1 Strategies to Debug Assembly Language Programs

Programming is a complicated task. Loosely speaking, a program can be thought of as mapping a set of input values to a set of output values. 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. Several debuggers are available to help us in the debugging process. We will look at two such tools in this appendix—GDB and DDD. Note that our goal here is to introduce the basics of the debugging process, as the best way to get familiar with debugging is to use a debugger.

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 you 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 few procedures are added to the program to add some specific functionality. The program must be tested before adding other functionality to the program. In general, it is a bad idea to write the whole program and then test it, 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 you attempt to translate it into an 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 we 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 a program in *single-step* mode. In this mode, a program executes a single statement and pauses. Then we can examine contents of registers, data in memory, stack contents, and so on. 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 in 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. The program execution stops at breakpoints, giving us a chance to look at the state of the program.

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 sections, we discuss two debuggers and show how they are useful in debugging the program `procex1.asm` discussed in Chapter 5 (this program is reproduced in Program C.1). We selected this program as it is simple so that we focus our attention on the debugging process, yet it contains a procedure to facilitate exploration of the stack frames.

Program C.1 The example program used in the debugging sessions

```
;Parameter passing via registers            PROCEX1.ASM
;
;        Objective: To show parameter passing via registers.
;            Input: Requests two integers from the user.
;           Output: Outputs the sum of the input integers.
%include "io.mac"
.DATA
prompt_msg1  db   "Please input the first number: ",0
prompt_msg2  db   "Please input the second number: ",0
sum_msg      db   "The sum is ",0

.CODE
      .STARTUP
      PutStr  prompt_msg1    ; request first number
      GetInt  CX             ; CX = first number

      PutStr  prompt_msg2    ; request second number
      GetInt  DX             ; DX = second number

      call    sum            ; returns sum in AX
      PutStr  sum_msg        ; display sum
      PutInt  AX
      nwln
done:
      .EXIT

;-----------------------------------------------------------
;Procedure sum receives two integers in CX and DX.
;The sum of the two integers is returned in AX.
;-----------------------------------------------------------
sum:
      mov     AX,CX          ; sum = first number
```

```
        add     AX,DX           ; sum = sum + second number
        ret
```

C.2 Preparing Your Program

The assembly process described in Appendix B works fine if we just want to assemble and run our program. However, we need to prepare our program slightly differently to debug the program. More specifically, we would like to pass the source code and symbol table information so that we can debug using the source-level statements. This is much better than debugging using disassembled code.

We use GNU debugger `gdb` to debug our assembly language programs. This is a command-line debugger, which is discussed in the next section. Section C.4 describes DDD, which acts as a nice GUI for `gdb`. To facilitate debugging at the source code level, we need to pass the symbolic information (source code, variable names, etc.) to the debugger. The `gdb` expects this symbolic information in the `stabs` format.

Recent versions of NASM support exporting symbolic information in this format. Since earlier versions did not support this feature, make sure that you have the most recent version of NASM (0.98.38-1 or a later version). To verify that your version of NASM supports `stabs` format, use the command

```
nasm -f elf -y
```

and see if `stabs` is listed as one of the formats supported.

Assuming that you have an NASM version that supports `stabs` format, we can assemble the program (say, `procex1.asm`) as follows:

```
nasm -f elf -g -F stabs procex1.asm
```

The `-g` option specifies that NASM should generate symbolic debug information. The `-F stabs` specifies the format for this information. The next step is to make sure that our linker will pass this information to the executable file. We do this by the following command:

```
ld -o procex1 procex1.o io.o
```

Note that we are not using the `-s` option as this option strips off the debugging information. The executable program `procex1` would have the necessary symbolic information to help us in the debugging process. As discussed in the last appendix, we need to include the I/O file `io.o` because our programs use the I/O routines described in Appendix B.

C.3 GNU Debugger

This section describes the GNU debugger `gdb`. It is typically invoked by

```
gdb file_name
```

For example, to debug the `procex1` program, we can use

```
gdb procex1
```

We can also invoke `gdb` without giving the file name. We can specify the file to be debugged by using the `file` command inside the `gdb`. Details on the `file` command are available in the gdb manual (see Section C.6). You know that the `gdb` is running the show when you see the `(gdb)` prompt. At this prompt, it can accept one of several commands. Tables C.1 and C.3 show some of the `gdb` commands useful in debugging programs.

C.3.1 Display Group

Displaying Source Code

When debugging, it is handy to keep a printed version of the source code with line numbers. However, `gdb` has list commands that allow us to look at the source code. A simple list command takes no arguments. The command

```
list
```

displays the default number of lines, which is 10 lines. If we issue this command again, it displays the next 10 lines. We can abbreviate this command to `l`. We can use `list –` to print lines just before the last printed lines.

We can specify a line number as an argument. In this case, it displays 10 lines centered on the specified line number. For example, the command

```
l 20
```

displays lines 15 through 24, as shown in Program C.2 on page 596. The list command can also take other arguments. For example,

```
l first,last
```

displays the lines from `first` to `last`.

The default number of lines displayed can be changed to `n` with the following command:

```
set listsize n
```

The command `show listsize` gives the current default value.

Displaying Register Contents

When debugging an assembly language program, we often need to look at the contents of the registers. The `info` can be used for this purpose. The

Table C.1 Some of the GDB Display Commands

Display Commands	
Source code display commands	
`list`	Lists default number of source code lines from the last displayed lines (default is 10 lines). It can be abbreviated as `l`.
`list -`	Lists default number of source code lines preceding the last displayed lines (default is 10 lines).
`list linenum`	Lists default number of lines centered around the specified line number `linenum`.
`list first,last`	Lists the source code lines from `first` to `last`.
Register display commands	
`info registers`	Displays the contents of registers except floating-point registers.
`info all-registers`	Displays the contents of registers.
`info register ...`	Displays contents of the specified registers.
Memory display commands	
`x address`	Displays the contents of memory at `address` (uses defaults).
`x/nfu adddress`	Displays the contents of memory at `address`.
Stack frame display commands	
`backtrace`	Displays backtrace of the entire stack (one line for each stack frame). It can be abbreviated as `bt`.
`backtrace n`	Displays backtrace of the innermost `n` stack frames.
`backtrace -n`	Displays backtrace of the outermost `n` stack frames.
`frame n`	Select frame `n` (Frame zero is the innermost frame i.e., currently executing frame). It can be abbreviated as `f`.
`info frame`	Displays a description of the selected stack frame (details include the frame address, program counter saved in it, addresses of local variable and arguments, addresses of next and previous frames, and so on).

```
info registers
```

displays the contents of the integer registers. To display all registers including the floating-point registers, use

```
info all-registers
```

Often we are interested in looking at the contents of a select few registers. To avoid cluttering the display, gdb allows specification of the registers in the command. For example, we can use

```
info eax ecx edx
```

to check the contents of the eax, ecx, and edx registers.

Displaying Memory Contents

We can examine memory contents by using the x command (x stands for examine). It has the following syntax:

```
x/nfu address
```

where n, f, and u are optional parameters that specify the amount of memory to be displayed starting at address and its format. If the optional parameters are not given, the x command can be written as

```
x address
```

In this case the default values are used for the three optional parameters. Details about these parameters are given in Table C.2.

Next we look at some examples of the x command. When gdb is invoked with the program given on page 585, we can examine the contents of the memory at prompt_msg1 by using the following x command:

```
(gdb) x/1sb &prompt_msg1
0x80493e4 <prompt_msg1>:      "Please input the first number: "
```

This command specifies the three optional parameters as n = 1, f = s, and u = b. We get the following output when we change the n value to 3:

```
(gdb) x/3sb &prompt_msg1
0x80493e4 <prompt_msg1>:      "Please input the first number: "
0x8049404 <prompt_msg2>:      "Please input the second number: "
0x8049425 <sum_msg>:          "The sum is "
```

As you can see from the program listing, it matches the three strings we declared in procex1.asm program.

Displaying Stack Frame Contents

This group of display commands helps us trace the history of procedure invocations. The backtrace command gives a list of procedure invocations at that point. It can be abbreviated as bt. This list consists of one line for each stack frame of the stack. As an example, consider a program that calls sum procedure that calls compute, which in turn calls another procedure get_values. If we stop the program in the get_values procedure and issue a backtrace command, we see the following output:

Table C.2 Details of the x Command Optional Parameters

n	Repeat count (decimal integer) Specifies the number of units (in u) of memory to be displayed Default value is 1	
f	Display format	
	x	displays in hexadecimal
	d	displays in decimal
	u	displays in unsigned decimal
	o	displays in octal
	t	displays in binary (t for two)
	a	displays address both in hexadecimal and as an offset from the nearest preceding symbol
	c	displays as a character
	s	displays as a NULL-terminated string
	t	displays as a floating-point number
	i	displays as a machine instruction
	Initial default is x.	
u	Unit size	
	b	bytes
	h	halfwords (2 bytes)
	w	words (4 bytes)
	g	giant words (8 bytes)
	Initial default is w. The default changes when a unit is specified with an x command.	

```
(gdb) bt
#0  get_values () at testex.asm:50
#1  0x080480bc in compute () at testex.asm:41
#2  0x080480a6 in sum () at testex.asm:27
```

This output clearly shows the invocation sequence of procedure calls with one line per invocation. The innermost stack frame is labelled #0, the next stack frame as #1, and so on. Each line gives the source code line that invoked the procedure. For example, the call instruction on line 27 (in source file testex.asm) invoked the compute procedure. The program counter value 0x080480a6 gives the return address. As discussed in Chapter 5, this is the address of the instruction following the

```
call    compute
```

instruction in the `sum` procedure. Similarly, the `call` instruction on line 41 in `compute` procedure invoked the `get_values` procedure. The return address for the `get_values` procedure is `0x080480bc`.

We can also restrict the number of stack frames displayed in the `backtrace` command by giving an optional argument. Details on this optional argument are given in Table C.1. For example, `bt 2` gives the innermost two stack frames as shown below:

```
 (gdb) bt 2
#0  get_values () at testex.asm:50
#1  0x080480bc in compute () at testex.asm:41
(More stack frames follow...)
```

The last line clearly indicates that there are more stack frames. To display the outermost two stack frames, we can issue `bt -2`. This command produces the following output for our example program:

```
(gdb) bt -2
#1  0x080480bc in compute () at testex.asm:41
#2  0x080480a6 in sum () at testex.asm:27
```

The `frame` and `info frame` commands allow us to examine the contents of the frames. We can select a frame by using the `frame` command. For our test program, `frame 1` gives the following output:

```
 (gdb) frame 1
#1  0x080480bc in compute () at testex.asm:41
41      call get_values
```

Once a frame is selected, we can issue the `info frame` command to look at the contents of this stack frame. Note that if no frame is selected using the `frame` command, it defaults to frame 0. The output produced for our example is shown below:

```
 (gdb) info f
Stack level 1, frame at 0xbffffa00:
 eip = 0x80480bc in compute (testex.asm:41); saved eip 0x80480a6
 called by frame at 0xbffffa08, caller of frame at 0xbffff9f8
 source language unknown.
 Arglist at 0xbffffa00, args:
 Locals at 0xbffffa00, Previous frame's sp is 0x0
 Saved registers:
  ebp at 0xbffffa00, eip at 0xbffffa04
(gdb)
```

In our example, each stack frame consists of the return address (4 bytes) and the EBP value stored by `enter 0,0` instruction on entering a procedure. The details given here indicate that the current stack frame is at `0xbffffa00` and previous and next frames are at

0xbffffa08 and 0xbffff9f8, respectively. It also shows where the arguments and locals are located as well as the registers saved on the stack. In our example, only the return address (EIP) and stack pointer (EBP) are stored on the stack for a total of eight bytes.

C.3.2 Execution Group

Breakpoint Commands

Breakpoints can be inserted using the break commands. As indicated in Table C.3, breakpoints can be specified using the source code line number, function name, or the address. For example, the following commands insert breakpoint at line 20 and function sum on line 32 in the procex1.asm program:

```
(gdb) b 20
Breakpoint 1 at 0x80480b0: file procex1.asm, line 20.
(gdb) b sum
Breakpoint 2 at 0x80480db: file procex1.asm, line 32.
(gdb)
```

Notice that each breakpoint is assigned a sequence number in the order we establish them.

We can use info breakpoints (or simply info b) to get a summary of breakpoints and their status. For example, after establishing the above two breakpoints, if we issue the info command, we get the following output:

```
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080480b0 procex1.asm:20
2   breakpoint     keep y   0x080480db procex1.asm:32
(gdb)
```

The Disp (Disposition) column indicates the action needed to be taken (keep, disable, or delete) when hit. By default, all breakpoints are of ‘keep’ type as in our example here. The enb indicates whether the breakpoint is enabled or disabled. A ‘y’ in this column indicated that the breakpoint is enabled.

We can use tbreak command to set a breakpoint with ‘delete’ disposition as shown below:

```
(gdb) tbreak 22
Breakpoint 3 at 0x80480c1: file procex1.asm, line 22.
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080480b0 procex1.asm:20
2   breakpoint     keep y   0x080480db procex1.asm:32
3   breakpoint     del  y   0x080480c1 procex1.asm:22
(gdb)
```

We can use the enable and disable commands to enable or disable the breakpoints. The following example disables breakpoint 2:

Table C.3 Some of the GDB Commands

Execution Commands	
Breakpoint commands	
`break linenum`	Sets a breakpoint at the specified line number in the current source file.
`break function`	Sets a breakpoint at entry to the specified function in the current source file.
`break *address`	Sets a breakpoint at the specified address. This command is useful if the debugging information or the source files are not available.
`info breakpoints`	Gives information on the breakpoints set. The information includes the breakpoint number, where the breakpoint is set in the source code, address, status (enabled or disabled), and so on.
`delete`	Deletes all breakpoints. By default, GDB runs this in query mode asking for confirmation for each breakpoint to be deleted. We can also specify a range as arguments (`delete range`). This command can be abbreviated as `d`.
`tbreak arg`	Sets a breakpoint as in break. The `arg` can be a line number, function name, or address as in the `break` command. However, the breakpoint is deleted after the first hit.
`disable range`	Disables the specified breakpoints. If no range is given, all breakpoints are disabled.
`enable range`	Enables the specified breakpoints. If no range is given, all breakpoints are enabled.
`enable once range`	Enables the specified breakpoints once; i.e., when the breakpoint is hit, it is disabled. If no range is given, all breakpoints are enabled once.
Program execution commands	
`run`	Executes the program under GDB. To be useful, you should set up appropriate breakpoints before issuing this command. It can be abbreviated as `r`.
`continue`	Continues execution from where the program has last stopped (e.g., due to a breakpoint). It can be abbreviated as `c`.

Table C.3 *(continued)*

Single-stepping commands	
`step`	Single-steps execution of the program (i.e., one source line at a time). In case of a procedure call, it single-steps into the procedure code. It can be abbreviated as `s`.
`step count`	Single-steps program execution `count` times. If it encounters a breakpoint before reaching the count, it stops execution.
`next`	Single-steps like the `step` command; however, procedure call is treated as a single statement (does not jump into the procedure code). As in the `step` command, we can specify a `count` value. It can be abbreviated as `n`.
`next count`	Single-steps program execution `count` times. If it encounters a breakpoint before reaching the count, it stops execution.
`stepi`	Executes one machine instruction. Like the `step` command, it single-steps into the procedure body. For assembly language programs, both `step` and `stepi` tend to behave the same. As in the `step` command, we can specify a `count` value. It can be abbreviated as `si`.
`nexti`	Executes one machine instruction. Like the `next` command, it treats a procedure call as a single machine instruction and executes the whole procdure. As in the `next` command, we can specify a `count` value. It can be abbreviated as `ni`.
Miscellaneous Commands	
`set listsize n`	Sets the default list size to `n` lines
`show listsize`	Shows the default list size
`q`	Quits `gdb`

```
(gdb) disable 2
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080480b0 procex1.asm:20
2   breakpoint     keep n   0x080480db procex1.asm:32
3   breakpoint     del  y   0x080480c1 procex1.asm:22
(gdb)
```

If we want to enable this breakpoint, we do so by the following command:

```
(gdb) enable 2
```

We use the `enable once` command to set a breakpoint with ‘disable’ disposition as shown below:

```
(gdb) enable once 2
(gdb) info b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080480b0 procex1.asm:20
2   breakpoint     dis  y   0x080480db procex1.asm:32
3   breakpoint     del  y   0x080480c1 procex1.asm:22
(gdb)
```

Program Execution Commands

Program execution command `run` is used to start the execution of the program. To be able to debug the program, breakpoints must be established before issuing the `run` command. The `continue` command resumes program execution from the last stop point (typically due to a breakpoint).

Single-Stepping Commands

GDB provides two basic single-stepping commands: `step` and `next`. The `step` command executes one source line at a time. In case of a procedutre call, it traces procedure execution in the single-step mode. The `next` command is similar to the `step` command except that it does not single-step the procedure body. Instead, it executes the entire procedure. Both `step` and `next` commands can take a `count` argument as shown in Table C.3. This table also gives details on the machine instruction version of these commands (see `stepi` and `nexti` commands).

C.3.3 Miscellaneous Group

The commands in Table C.3 are useful to manipulate the list size and exit the `gdb`.

C.3.4 An Example

A sample `gdb` session on `procex1.asm` is shown in Program C.2. The `l 20` command on line 9 displays the source code centered on the source code line 20. Before issuing the `r` command on line 22, we insert a breakpoint at source code line 20 using the `break` command on line 20. The run command executes the program until it hits line 20. Then it stops and prints breakpoint information. Notice that we entered two input numbers (1234 and 5678) before hitting the breakpoint.

To check that these two input numbers are read into ECX and EDX registers, we can look at the contents of these two registers by issuing the `info registers` command specifying these two registers (see line 28). The output of this command shows that these registers indeed received the two input numbers.

Program C.2 A sample `gdb` session

```
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.
Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) l 20
15       GetInt  CX             ; CX = first number
16
17       PutStr  prompt_msg2    ; request second number
18       GetInt  DX             ; DX = second number
19
20       call    sum            ; returns sum in AX
21       PutStr  sum_msg        ; display sum
22       PutInt  AX
23       nwln
24 done:
(gdb) break 20
Breakpoint 1 at 0x80480b0: file procex1.asm, line 20.
(gdb) r
Starting program: /mnt/hgfs/winXP_D/temp/gdb_test/procex1
Please input the first number: 1234
Please input the second number: 5678
Breakpoint 1, _start () at procex1.asm:20
20       call    sum            ; returns sum in AX
(gdb) info registers ecx edx
ecx            0x4d2 1234
edx            0x162e 5678
(gdb) si
32       mov     AX,CX          ; sum = first number
(gdb) si
33       add     AX,DX          ; sum = sum + second number
(gdb) si
34       ret
(gdb) info registers eax ecx edx
eax            0x1b00 6912
ecx            0x4d2 1234
edx            0x162e 5678
(gdb) c
Continuing.
```

```
43:  The sum is 6912
44:
45:  Program exited normally.
46:  (gdb) q
```

We run the sum procedure in single-step mode (see commands on lines 31, 33, and 35). To see if the result in EAX is the sum of the two input values, we display the contents of the three registers (lines 38–40) using the info registers command on line 37. After verifying, we let the program continue its execution using the continue command on line 41. Finally, on line 46, we use the quit command to exit gdb.

C.4 Data Display Debugger

The Data Display Debugger (DDD) is front-end to a command-line debugger. DDD supports several command-line debuggers including gdb, dbx, jdb, and so on. Our interest here is in using DDD as a front-end to gdb discussed in the last section. Since DDD is a GUI to gdb, we prepare our program exactly as we do for the gdb (see Section C.2 on page 586).

We can invoke DDD on procex1 by

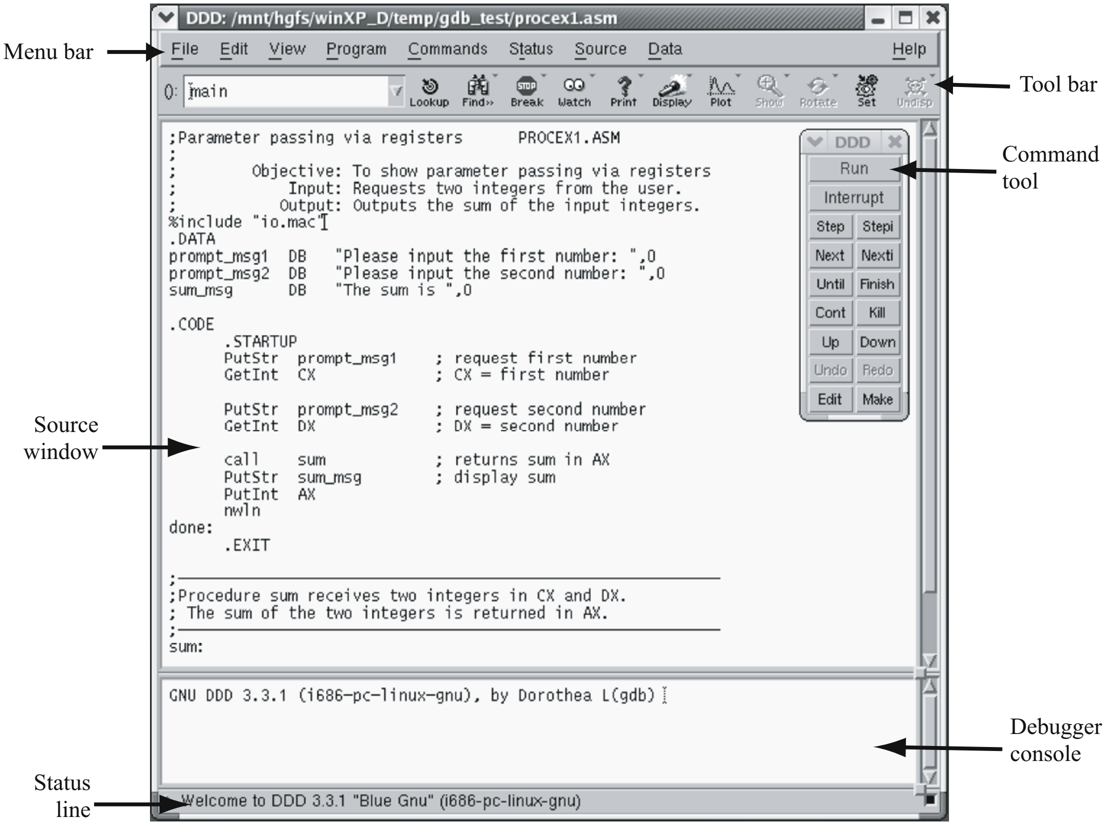
```
ddd  procex1
```

Figure C.1 shows the initial screen that appears after invoking DDD. The screen consists of the Source Window that displays the source program, Debugger Console, Status Line, Command Tool window, Menu Bar, and Tool Bar. The debugger console acts as the program's input/output console to display messages and to receive input and so on.

We can insert a breakpoint using the Tool Bar. For example, to insert a breakpoint on line 20, place the cursor to the left of line 20 and click the the breakpoint (red stop sign) on the Tool Bar. This inserts a breakpoint on line 20, which is indicated by a red stop sign on line 20, as shown in Figure C.2. This figure also shows source code line numbers and the Machine Code window. Both of these can be selected from the Source pull down menu in the Menu Bar.

Once this breakpoint is inserted, we can run the program by clicking Run in the Command Tool. The big arrow next to the stop sign (on line 20) indicates that the program execution stopped at that line. While executing the program before reaching the breakpoint on line 20, the program takes two input numbers as shown in the Debugger Console (see Figure C.2). We can get information on the breakpoints set in the program by selecting Breakpoints... in the Source pull-down menu. For our example program, it gives details on the single breakpoint we set on line 20 (see Figure C.3). The details provided in this window are the same as those discussed in the last section. The breakpoint information also includes the number of hits as shown in Figure C.3.

All the execution commands of gdb, discussed in the last section, are available in the Program pull-down menu (see Figure C.4). Figure C.5 shows the screen after single-



```
;Parameter passing via registers      PROCEX1.ASM
;
;         Objective: To show parameter passing via registers
;             Input: Requests two integers from the user.
;            Output: Outputs the sum of the input integers.
%include "io.mac"
.DATA
prompt_msg1  DB   "Please input the first number: ",0
prompt_msg2  DB   "Please input the second number: ",0
sum_msg      DB   "The sum is ",0

.CODE
      .STARTUP
      PutStr  prompt_msg1    ; request first number
      GetInt  CX             ; CX = first number

      PutStr  prompt_msg2    ; request second number
      GetInt  DX             ; DX = second number

      call    sum            ; returns sum in AX
      PutStr  sum_msg        ; display sum
      PutInt  AX
      nwln
done:
      .EXIT

;-------------------------------------------------------------
;Procedure sum receives two integers in CX and DX.
; The sum of the two integers is returned in AX.
;-------------------------------------------------------------
sum:
```

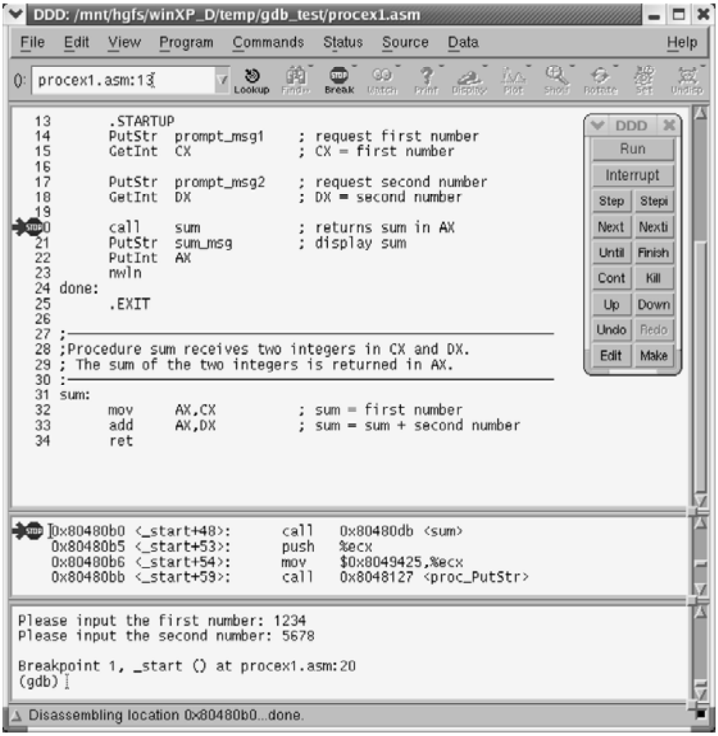
```
GNU DDD 3.3.1 (i686-pc-linux-gnu), by Dorothea L(gdb)
```

```
Welcome to DDD 3.3.1 "Blue Gnu" (i686-pc-linux-gnu)
```

Figure C.1 DDD window at the start of `procex1` program.

stepping through the `sum` procedure. The program is stopped at the `ret` instruction on line 34. To verify the functionality of the procedure, we can display the contents of the registers. This is done by selecting `Registers...` in the `Status` pull-down menu. The contents of the registers, shown in Figure C.6, clearly indicate that the sum of the two input numbers (in ECX and EDX registers) is in the EAX register.

The examination commands of `gdb` are available under `Data` pull-down menu. A sample memory examination window is shown in Figure C.7. This window allows us to specify the memory location, format to be used to display the contents, size of the data, and number of data items to be examined. In the window of Figure C.7, we specified `&prompt_msg1` as the location and `string` as the output format. The size is given as `bytes` and the number of strings to be examined is set to 1.


DDD: /mnt/hgfs/winXP_D/temp/gdb_test/procex1.asm
File Edit View Program Commands Status Source Data Help
(): procex1.asm:13
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

```
13        .STARTUP
14        PutStr  prompt_msg1    ; request first number
15        GetInt  CX             ; CX = first number
16
17        PutStr  prompt_msg2    ; request second number
18        GetInt  DX             ; DX = second number
19
20        call    sum            ; returns sum in AX
21        PutStr  sum_msg        ; display sum
22        PutInt  AX
23        nwln
24 done:
25        .EXIT
26
27 ;---------------------------------------------------------------
28 ;Procedure sum receives two integers in CX and DX.
29 ; The sum of the two integers is returned in AX.
30 :---------------------------------------------------------------
31 sum:
32        mov     AX,CX          ; sum = first number
33        add     AX,DX          ; sum = sum + second number
34        ret
```

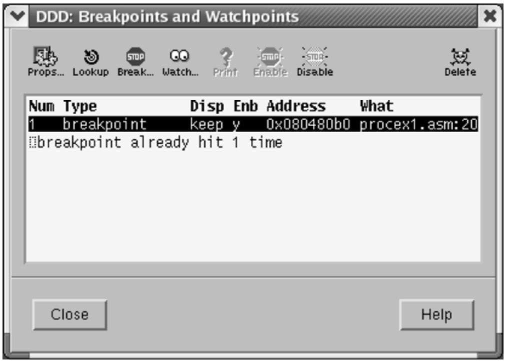
```
0x80480b0 <_start+48>:      call   0x80480db <sum>
0x80480b5 <_start+53>:      push   %ecx
0x80480b6 <_start+54>:      mov    $0x8049425,%ecx
0x80480bb <_start+59>:      call   0x8048127 <proc_PutStr>
```

```
Please input the first number: 1234
Please input the second number: 5678

Breakpoint 1, _start () at procex1.asm:20
(gdb)
```

Disassembling location 0x80480b0...done.

Figure C.2 DDD window at the breakpoint on line 20. This screenshot also shows the machine code window and the source code line numbers.


DDD: Breakpoints and Watchpoints
Props... Lookup Break... Watch... Print Enable Disable Delete
Close
Help

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080480b0 procex1.asm:20
    breakpoint already hit 1 time
```

Figure C.3 Breakpoints window.

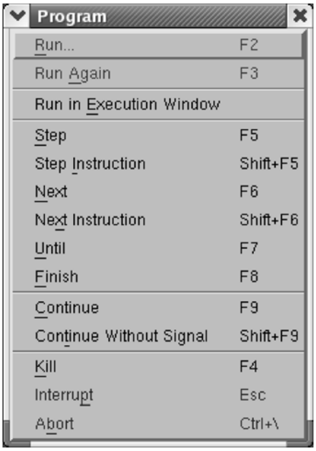
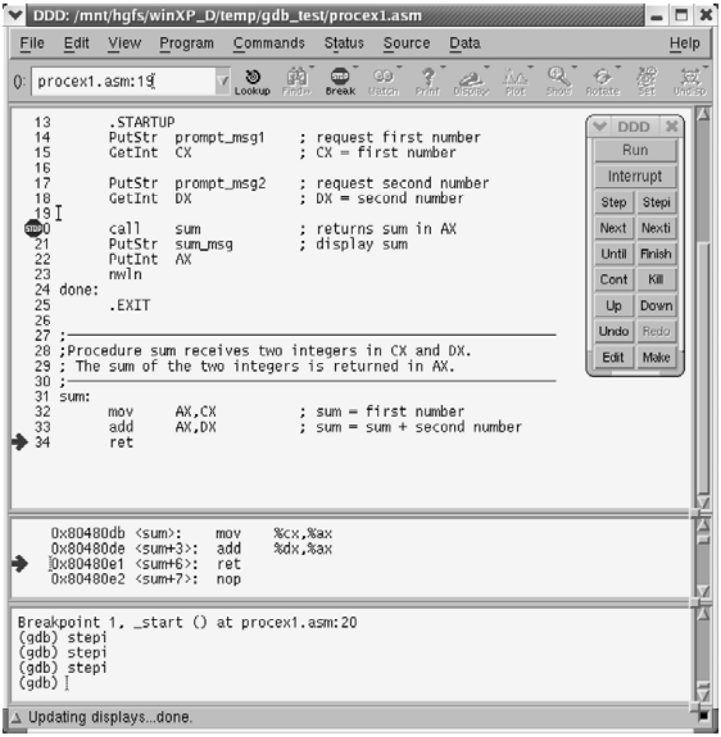


Figure C.4 Details of the `Program` pull-down menu.



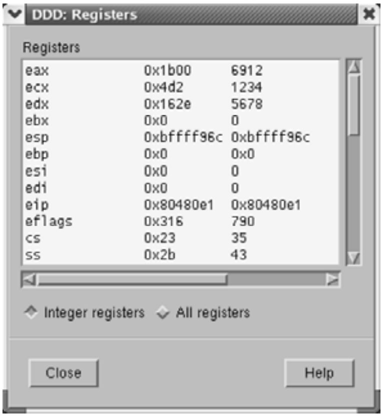
```
 13        .STARTUP
 14        PutStr  prompt_msg1    ; request first number
 15        GetInt  CX             ; CX = first number
 16
 17        PutStr  prompt_msg2    ; request second number
 18        GetInt  DX             ; DX = second number
 19
 20        call    sum            ; returns sum in AX
 21        PutStr  sum_msg        ; display sum
 22        PutInt  AX
 23        nwln
 24 done:
 25        .EXIT
 26
 27 ;-----------------------------------------------------------
 28 ;Procedure sum receives two integers in CX and DX.
 29 ; The sum of the two integers is returned in AX.
 30 ;-----------------------------------------------------------
 31 sum:
 32        mov     AX,CX          ; sum = first number
 33        add     AX,DX          ; sum = sum + second number
 34        ret
```

```
0x80480db <sum>:    mov    %cx,%ax
0x80480de <sum+3>:  add    %dx,%ax
0x80480e1 <sum+6>:  ret
0x80480e2 <sum+7>:  nop
```

```
Breakpoint 1, _start () at procex1.asm:20
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb)
```

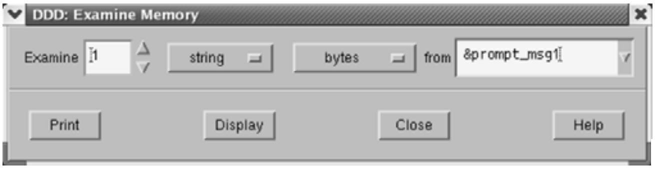
Updating displays...done.

Figure C.5 DDD window after single stepping from the breakpoint on line 20.



```
eax            0x1b00      6912
ecx            0x4d2       1234
edx            0x162e      5678
ebx            0x0         0
esp            0xbffff96c  0xbffff96c
ebp            0x0         0x0
esi            0x0         0
edi            0x0         0
eip            0x80480e1   0x80480e1
eflags         0x316       790
cs             0x23        35
ss             0x2b        43
```

Figure C.6 Register Window after the single-stepping shown in Figure C.5.



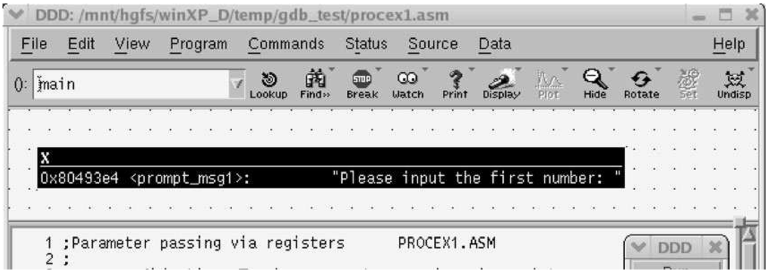
Examine 1 string bytes from &prompt_msg1

Figure C.7 Memory Examination Window set to display a string.

By clicking `Display`, the contents are displayed in the Data Window that appears above the Source Window as shown in Figure C.8. We can pick the windows we want to see by selecting them from the `View` pull-down menu. The `View` menu gives control to select any of the four windows: Debuger Console Window, Machine Code Window, Source Window, and Data Window.

We can also elect to display the contents in the Debugger Console Window using the `Print` command. Figure C.9 shows how we can display the three strings in our program in the Console window. This Examine Memory window is similar to that shown in Figure C.7 except that we set the number of strings to be displayed as 3. The result of executing this `x` command is shown in Figure C.10, which shows the three strings in our program.

Both `gdb` and DDD provide several other features that are useful in debugging programs. Our intent here is to introduce some of the basic features of these debuggers. More details on these debuggers are available from their Web sites. We provide pointers to these Web sites at the end of this appendix.

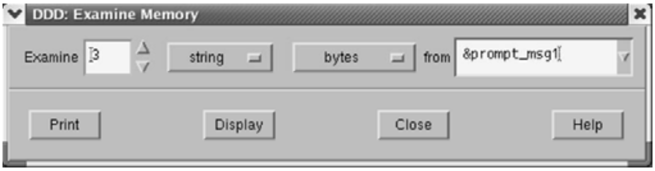


```
(): main
```

```
x
0x80493e4 <prompt_msg1>:          "Please input the first number: "
```

```
 1 ;Parameter passing via registers      PROCEX1.ASM
 2 ;
```

Figure C.8 Data Window displays the string.



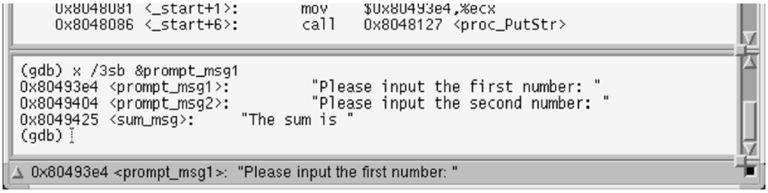
Examine 3 string bytes from &prompt_msg1

Figure C.9 Memory Examination Window set to display three strings.

C.5 Summary

We started this appendix with a brief discussion of some basic debugging techniques. Since assembly language is a low-level programming language, debugging tends to be even more tedious than debugging a high-level language program. It is, therefore, imperative to follow good programming practices in order to simplify debugging of assembly language programs.

There are several tools available for debugging programs. We discussed two debuggers—gdb and DDD—in this appendix. While gdb is a command line-oriented debugger, the DDD provides a nice front-end to the gdb. The best way to learn to use these debuggers is by hands-on experience.



```
    0x8048081 <_start+1>:       mov    $0x80493e4,%ecx
    0x8048086 <_start+6>:       call   0x8048127 <proc_PutStr>
```

```
(gdb) x /3sb &prompt_msg1
0x80493e4 <prompt_msg1>:         "Please input the first number: "
0x8049404 <prompt_msg2>:         "Please input the second number: "
0x8049425 <sum_msg>:     "The sum is "
(gdb)
```

```
0x80493e4 <prompt_msg1>:  "Please input the first number: "
```

Figure C.10 Console Window displays the three strings.

C.6 Web Resources

Details on gdb are available from http://www.gnu.org/software/gdb.

Details on DDD are available from http://www.gnu.org/software/ddd.

C.7 Exercises

C–1 Discuss some general techniques useful in debugging programs.

C–2 How are window-oriented debuggers like DDD better than line-oriented debuggers like gdb?

C–3 What is the difference between the step and next commands of gdb?

C–4 Discuss how breakpoints are useful in debugging programs.

C–5 Is the Machine Code Window of DDD more useful in debugging assembly language programs? If so, explain your reasons.

C.8 Programming Exercises

C–P1 Take a program from Chapter 4 and ask your friend to deliberately introduce some logical errors into the program. Then use your favorite debugger to locate and fix errors. Discuss the features of your debugger that you found most useful.

C–P2 Using your debugger's capability to display flags, verify the values of the flags given in Table 7.1 on page 210.

Appendix D

SPIM Simulator and Debugger

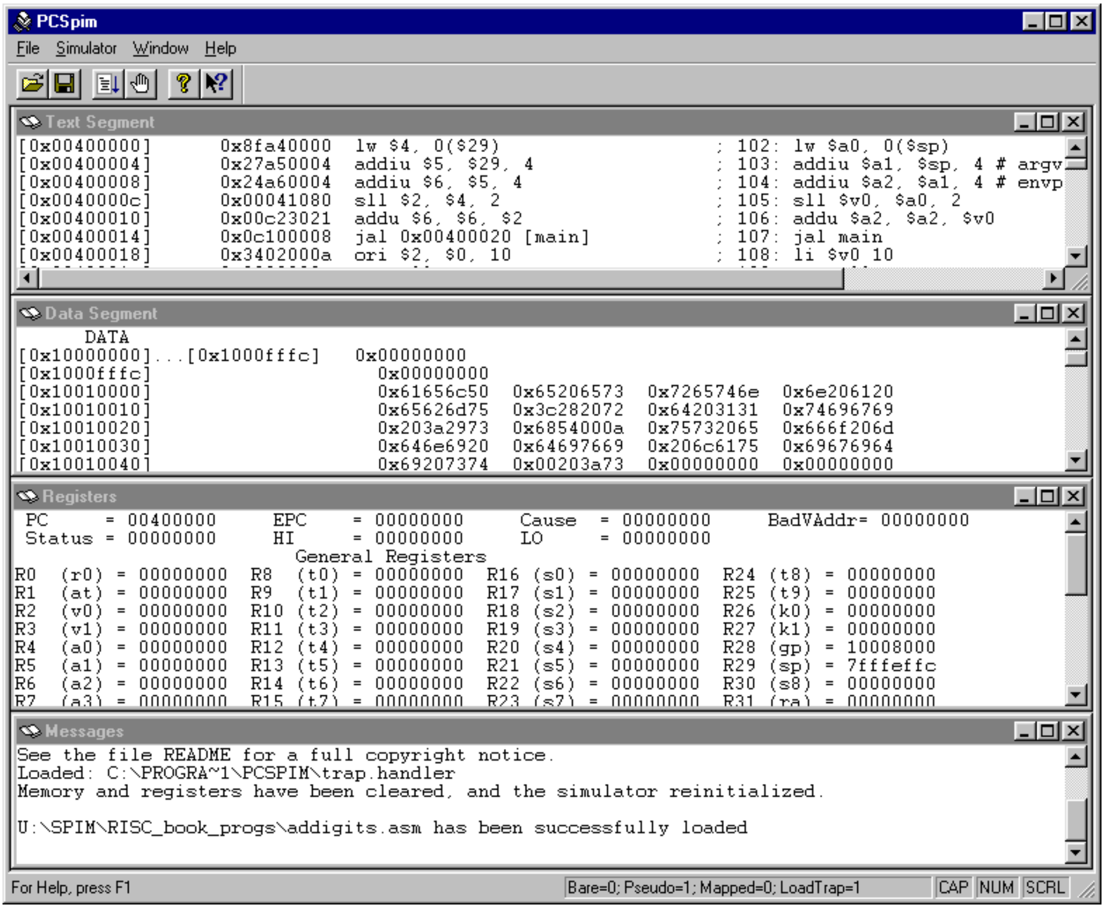
Objectives

- 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 13. Details about loading and running an MIPS program are discussed in the next section. This section also presents debugging facilities provided by SPIM. We conclude the appendix with a summary.

D.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.



```
[0x00400000]      0x8fa40000  lw $4, 0($29)                  ; 102: lw $a0, 0($sp)
[0x00400004]      0x27a50004  addiu $5, $29, 4               ; 103: addiu $a1, $sp, 4 # argv
[0x00400008]      0x24a60004  addiu $6, $5, 4                ; 104: addiu $a2, $a1, 4 # envp
[0x0040000c]      0x00041080  sll $2, $4, 2                  ; 105: sll $v0, $a0, 2
[0x00400010]      0x00c23021  addu $6, $6, $2                ; 106: addu $a2, $a2, $v0
[0x00400014]      0x0c100008  jal 0x00400020 [main]          ; 107: jal main
[0x00400018]      0x3402000a  ori $2, $0, 10                 ; 108: li $v0 10
```

```
      DATA
[0x10000000]...[0x1000fffc]   0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x61656c50  0x65206573  0x7265746e  0x6e206120
[0x10010010]                    0x65626d75  0x3c282072  0x64203131  0x74696769
[0x10010020]                    0x203a2973  0x6854000a  0x75732065  0x666f206d
[0x10010030]                    0x646e6920  0x64697669  0x206c6175  0x69676964
[0x10010040]                    0x69207374  0x00203a73  0x00000000  0x00000000
```

```
 PC      = 00400000     EPC     = 00000000     Cause   = 00000000     BadVAddr= 00000000
 Status  = 00000000     HI      = 00000000     LO      = 00000000
                             General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
R6  (a2) = 00000000  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
```

```
See the file README for a full copyright notice.
Loaded: C:\PROGRA~1\PCSPIM\trap.handler
Memory and registers have been cleared, and the simulator reinitialized.

U:\SPIM\RISC_book_progs\addigits.asm has been successfully loaded
```

For Help, press F1 | Bare=0; Pseudo=1; Mapped=0; LoadTrap=1

Figure D.1 SPIM windows.

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 PCSpim. 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 D.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 D.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 D.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 an 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 D.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 D.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.

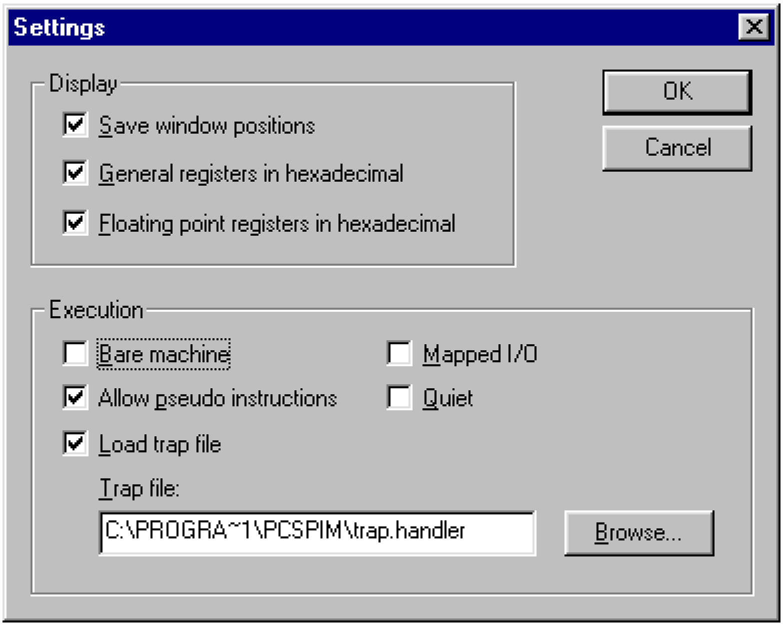
- **Status Bar:** The status bar at the bottom of the PCSpim window presents three pieces of information:
  - The left area is used to give information about the menu items and toolbar buttons. For example, when the mouse arrow is on the open file icon (first button) on the toolbar, this area displays 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.

D.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 D.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 display 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 D.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 13 for details on the assembler-supported pseudoinstructions and addressing modes. Since the example MIPS programs presented in Chapter 13 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.



☑ Save window positions

☑ General registers in hexadecimal

☑ Floating point registers in hexadecimal

□ Bare machine

□ Mapped I/O

☑ Allow pseudo instructions

□ Quiet

☑ Load trap file

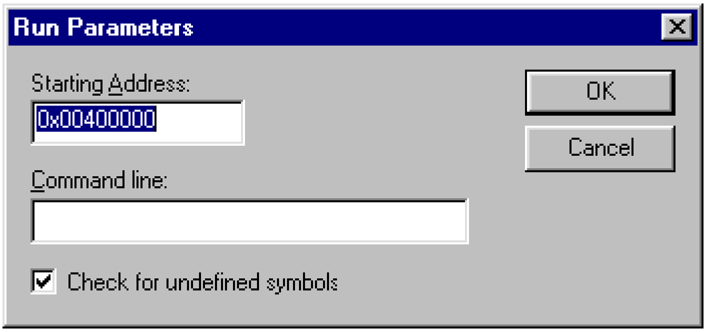
Trap file: C:\PROGRA~1\PCSPIM\trap.handler

Figure D.2 SPIM settings window.

- **Mapped I/O:** If this setting is selected, SPIM enables the memory-mapped I/O facility. Memory-mapped I/O is discussed in Chapter 2. When this setting is selected, you cannot use SPIM system calls, described in Section 13.2.1 on page 373, to read from the terminal. Thus, this setting should not be selected to run our example programs from Chapter 13.

- **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.



Starting Address: 0x00400000

Command line:

☑ Check for undefined symbols

Figure D.3 Run window.

D.3 Running and Debugging a Program

D.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 D.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` 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 13.

D.3.2 Debugging

SPIM provides the standard facilities to debug programs. As discussed in Appendix C, 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 C, 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 D.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).

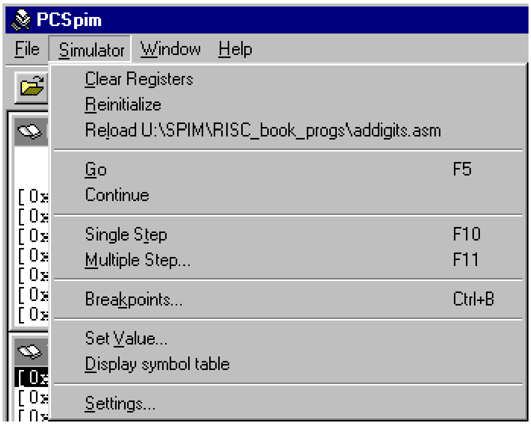
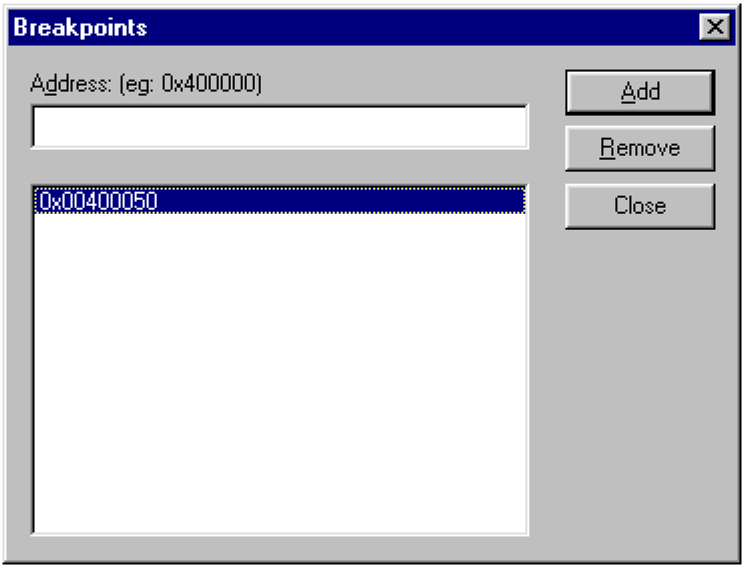


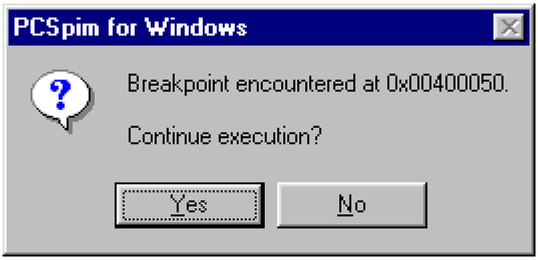
Figure D.4 Debug commands available under the `Simulator` menu.

- **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.
- **Multiple Step:** This is a debug command we have not discussed in Appendix C. 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 D.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 D.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.



0x00400050

Figure D.5 Breakpoints dialog box.



Breakpoint encountered at 0x00400050.

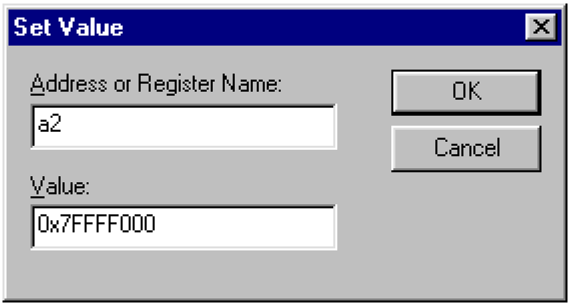
Continue execution?

Figure D.6 Breakpoint query window.

- **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 D.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 609. We have discussed the simulator settings in detail in Section D.2.



Address or Register Name: a2

Value: 0x7FFFF000

Figure D.7 Set value dialog box.

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.

D.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.

D.5 Exercises

D–1 Discuss the situations where the `Multiple Step` command is useful in debugging programs.

D–2 In our setup, the run command displays the execution start address as 0x00400000. Explain why.

D–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.

D.6 Programming Exercises

D–P1 Take a program from Chapter 13 and ask a friend to deliberately introduce some logical errors into the program. Then use the SPIM debugger to locate and fix the errors.

Appendix E

IA-32 Instruction Set

Objectives

- To describe the instruction format
- To present selected IA-32 instructions

Instruction format and encoding encompass a variety of factors: addressing modes, number of operands, number of registers, sources of operands, etc. Instructions can be of fixed length or variable length. In a fixed-length instruction set, all instructions are of the same length. The IA-32 instruction set uses variable-length instructions to accommodate the complexity of the instructions. Section E.1 discusses the IA-32 instruction format. A subset of this instruction set is given in Section E.2.

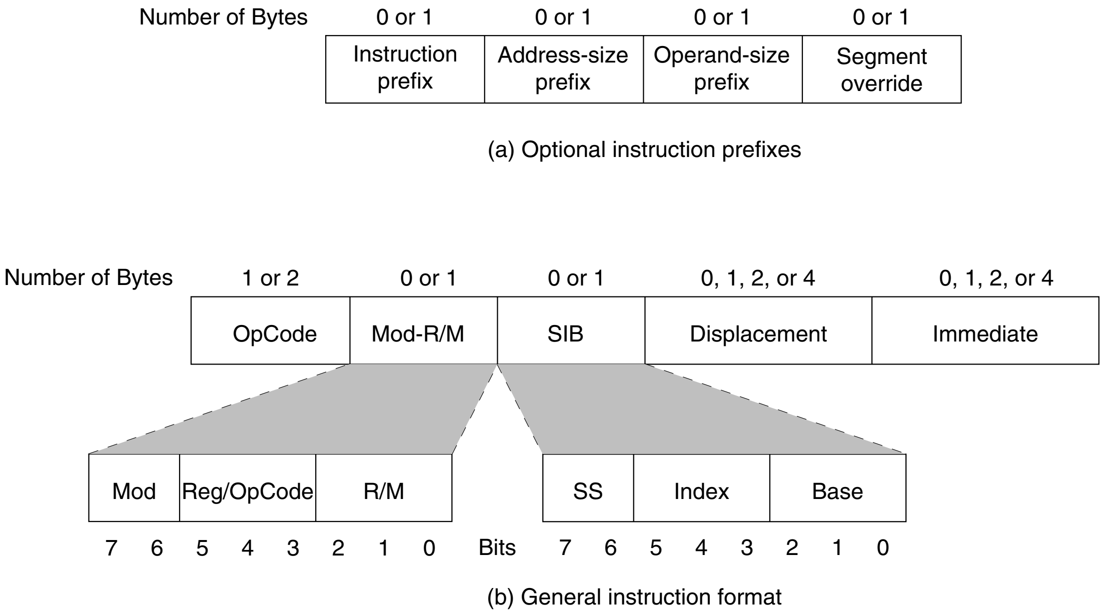
E.1 Instruction Format

In the IA-32 architecture, instruction length varies between 1 and 16 bytes. The instruction format is shown in Figure E.1. The general instruction format is shown in Figure E.1b. In addition, instructions can have several optional instruction prefixes shown in Figure E.1a. The next two subsections discuss the instruction format in detail.

E.1.1 Instruction Prefixes

There are four instruction prefixes, as shown in Figure E.1a. 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 10. This group of prefixes consists of `rep`, `repe/repz`, `repne/repnz`, and `lock`. The three repeat prefixes were discussed in detail in Chapter 10. The `lock` prefix is useful in multiprocessor systems to ensure exclusive use of shared memory.



(a) Optional instruction prefixes

(b) General instruction format

Figure E.1 The IA-32 instruction format.

- *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 by using a segment prefix. We saw an example of this in Chapter 5 (see Program 5.6 on page 149). The following segment override prefixes are available: CS, SS, DS, ES, FS, and GS.
- *Address-Size Override Prefix*: This prefix is useful in overriding the default address size. As discussed in Chapter 3, the D bit indicates the default address and operand size. A D bit of 0 indicates the default address and operand sizes of 16 bits and a D bit of 1 indicates 32 bits. The address size can be either 16 bits or 32 bits long. This prefix can be used to switch between the two sizes.
- *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, using a 32-bit register, for example, is possible by prefixing the instruction with the operand-size override prefix.

These four prefixes can be used in any combination, and in any order.

E.1.2 General Instruction Format

The general instruction format consists of the Opcode, an optional address specifier consisting of a Mod R/M byte and SIB (scale-index-base) byte, an optional displacement, and an immediate data field, if required. Next we briefly discuss these five fields.

- *Opcode*: This field can be 1 or 2 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 with an immediate value for the shift count takes two bytes (the opcode is 0FA4H). The opcode field also contains other smaller encoding fields. These fields include the register encoding, direction of operation (to or from memory), the size of displacement, and whether the immediate data must be sign-extended. For example, the instructions

  ```
  push    EAX
  push    ECX
  push    EDX
  push    EBX
  ```

  are encoded as 50H, 51H, 52H, and 53H, respectively. Each takes only one byte that includes the operation code (push) as well as the register encoding (EAX, ECX, EDX, or EBX).
- *Mod R/M*: This byte and the SIB byte together provide addressing information. The Mod R/M byte consists of three fields, as shown in Figure E.1.
  - *Mod*: This field (2 bits) along with the R/M field (3 bits) specify one of 32 possible choices: 8 registers and 24 indexing modes.
  - *Reg/Opcode*: This field (3 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 (3 bits) either specifies a register as the location of 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 the SIB byte. The presence of the SIB byte is indicated by certain encodings of the Mod R/M byte. The SIB byte consists of three fields, as shown in Figure E.1. The SS field (2 bits) specifies the scale factor (1, 2, 4, or 8). The index and base fields (3 bits each) specify the index and base registers, respectively.
- *Displacement*: If an addressing mode requires a displacement value, this field provides the required value. When present, it is an 8-, 16- or 32-bit signed integer. For example

  ```
        jg    SHORT done
        pop   EBX
  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 the last one in the instruction. It is present in those instructions that specify an immediate operand. When present, it is an 8-, 16- or 32-bit operand. For example

```
mov    EAX,256
```

is encoded as `B8 00000100`. Note that the first byte B8 not only identifies the instruction as `mov` but also specifies the destination register as EAX (by the least significant three bits of the opcode byte). The following encoding is used for the 32-bit registers:

EAX = 0	ESP = 4
ECX = 1	EBP = 5
EDX = 2	ESI = 6
EBX = 3	EDI = 7

The last four bytes represent the immediate value 256, which is equal to 00000100H. If we change the register from EAX to EBX, the opcode byte changes from B8 to BB.

E.2 Selected Instructions

This section gives selected instructions in alphabetical order. For each instruction, instruction mnemonic, flags affected, format, and a description are given. For a more detailed description, please refer to the *Pentium Processor Family Developer's Manual—Volume 3: Architecture and Programming Manual*. The clock cycles reported are for the Pentium processor. While most of the components are self explanatory, 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

aaa — ASCII adjust after addition

C	O	Z	S	P	A
M	*	*	*	*	M

Format: `aaa`

Description: ASCII adjusts AL register contents after addition. The AF and CF are set if there is a decimal carry, cleared otherwise. See Chapter 11 for details. Clock cycles: 3.

aad — ASCII adjust before division

C	O	Z	S	P	A
*	*	M	M	M	*

Format: `aad`

Description: ASCII adjusts AX register contents before division. See Chapter 11 for details. Clock cycles: 10.

aam — ASCII adjust after Multiplication

C	O	Z	S	P	A
*	*	M	M	M	*

Format: `aam`

Description: ASCII adjusts AX register contents after multiplication. See Chapter 11 for details. Clock cycles: 18.

aas — ASCII adjust after subtraction

C	O	Z	S	P	A
M	*	*	*	*	M

Format: `aas`

Description: ASCII adjusts AL register contents after subtraction. The AF and CF are set if there is a decimal carry, cleared otherwise. See Chapter 11 for details. Clock cycles: 3.

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: 1–3.

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: 1–3.

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: 1–3

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` must be either both 16- or 32-bit operands. While the `src` operand can be either in a register or memory, `dest` must be a register. Clock cycles: 6–35 for 16-bit operands and 6–43 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 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` must be either both 16- or 32-bit operands. While the `src` operand can be either in a register or memory, `dest` must be a register. Clock cycles: 7–40 for 16-bit operands and 7–72 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 in 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: 4–9.

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 in 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: 7–13.

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 in memory. The second operand `src2` can be 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 7–13.

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 in memory. The second operand `src2` can be 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 7–13.

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 are 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 5. 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 operand pointed by the DI or EDI register from that by 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 `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.

daa — Decimal adjust after addition

C	O	Z	S	P	A
M	*	M	M	M	M

Format: `daa`

Description: The `daa` instruction is useful in BCD arithmetic. It adjusts the AL register to contain the correct two-digit packed decimal result. This instruction should be used after an addition instruction, as described in Chapter 11. Both AF and CF flags are set if there is a decimal carry; these two flags are cleared otherwise. The ZF, SF, and PF flags are set according to the result. Clock cycles: 3.

das — Decimal adjust after subtraction

C	O	Z	S	P	A
M	*	M	M	M	M

Format: `das`

Description: The `das` instruction is useful in BCD arithmetic. It adjusts the AL register to contain the correct two-digit packed decimal result. This instruction should be used after a subtract instruction, as described in Chapter 11. Both AF and CF flags are set if there is a decimal borrow; these two flags are cleared otherwise. The ZF, SF, and PF flags are set according to the result. 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 7 for details. Clock cycles: 17 for an 8-bit divisor, 25 for a 16-bit divisor, and 41 for a 32-bit divisor.

enter — Allocate stack frame

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `enter    bytes,level`

Description: This instruction creates a stack frame at procedure entry. The first operand `bytes` specifies the number of bytes for the local variable storage in the stack frame. 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 the

```
enter    XX,0
```

statement is equivalent to

```
push    EBP
mov     EBP,ESP
sub     ESP,XX
```

See Section 5.7.5 for more details on its usage. Clock cycles: 11 if `level` is zero.

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 7 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 1 and 3, 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` must both be either 16-bit or 32-bit operands. While `src` can be either in a register or in 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 in memory. The immediate constant can be a 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 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 an 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 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, 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 14 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-bit or 32-bit instruction pointer (IP or EIP) and flags (FLAGS or EFLAGS) are be used. See Chapter 14 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 8. See Chapter 8 for details on the various conditions tested like `ja`, `jbe`, etc. 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 8 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 * represent 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 load the 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 2 or 4 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 2 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 16 bits, the effective address is zero-extended to 32 bits. Clock cycles: 1.

leave — Procedure exit

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `leave`

Description: The `leave` instruction takes no operands. Effectively, it reverses the actions of the `enter` instruction. It performs two actions:

- Releases the local variable stack space allocated by the `enter` instruction;
- Old frame pointer is popped into (E)BP register.

This instruction is typically used just before the `ret` instruction. 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 a 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, doubleword operands, respectively. The `rep` prefix can be used with this instruction but is not useful, as explained in Chapter 10. 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. The `loopz` is a synonym for the `loope` instruction.

The `loopne` instruction is similar to `loopne` except that it transfers control to `target` if ZF is 0 (instead of 1 as in the `loope` instruction). See Chapter 8 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. 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 `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-bit or 32-bit register only. The source can be a register or 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 operand needs to 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 (two’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: Like the `in` instruction, this instruction has two formats. In both formats, `src` 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 an 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, 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, BP, discard next two bytes (to skip loading into SP), 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, or (ii) a memory operand, or (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 was 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; 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.

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 instruction, it can take as many as 27 clock cycles—see Pentium data book for details.

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 the `rep cmps` and `rep scas` instructions. For more details, see Chapter 10. Clock cycles: varies—see Pentium data book for details.

ret — Return form 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 5 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.

sahf — Store AH into 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 the 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 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, DI register is used; 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 a 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. While `dest` can be a register or memory operand, `src` must 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 the shift.

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. 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, DI register is used; 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: 1–3.

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 in 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) registers are used as the offset to the 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: 1–3.

Appendix F

MIPS/SPIM Instruction Set

This appendix lists the MIPS instructions implemented by the SPIM simulator. These instructions can be divided into two groups: instructions and pseudoinstructions. The first group consists of the instructions supported by the processor. The pseudoinstructions are supported by the assembler; these are not the processor instructions. These pseudoinstructions are translated into one or more processor instructions. For example, `abs` is a pseudoinstruction, which is translated into the following two-instruction sequence:

```
bgez    Rsrc,8
sub     Rdest,$0,Rsrc
```

In this appendix, as in the main text, the pseudoinstructions are indicated by a †. In the following, instructions are presented in alphabetical order.

Also note that, in all the instructions, `Src2` can be either a register or a 16-bit integer. The assembler translates the general form of an instruction to its immediate form if `Src2` is a constant. For reference, we also include the immediate form instructions. In these instructions, `Imm` represents a 16-bit integer.

abs† — Absolute value

Format: `abs     Rdest,Rsrc`

Description: Places the absolute value of `Rsrc` in `Rdest`.

add — Add with overflow

Format: `add    Rdest,Rsrc1,Src2`

Description: `Rdest` receives the sum of `Rsrc1` and `Src2`. The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated.

addi — Add immediate with overflow

Format: `addi    Rdest,Rsrc1,Imm`

Description: `Rdest` receives the sum of `Rsrc1` and `Imm`. The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated.

addiu — Add immediate with no overflow

Format: `addiu    Rdest,Rsrc1,Imm`

Description: `Rdest` receives the sum of `Rsrc1` and `Src2`. The numbers are treated as signed integers. No overflow exception is generated.

addu — Add with no overflow

Format: `addu    Rdest,Rsrc1,Src2`

Description: `Rdest` receives the sum of `Rsrc1` and `Src2`. The numbers are treated as signed integers. No overflow exception is generated.

and — Logical AND

Format: `and    Rdest,Rsrc1,Src2`

Description: Bitwise AND of `Rsrc1` and `Src2` is stored in `Rdest`.

andi — Logical AND immediate

Format: `andi    Rdest,Rsrc1,Imm`

Description: Bitwise AND of `Rsrc1` and `Imm` is stored in `Rdest`.

b† — Branch

Format: `b label`

Description: Unconditionally transfer control to the instruction at `label`. Branch instruction uses a signed 16-bit offset. This allows jumps to $2^{15} - 1$ instructions (not bytes) forward, or 2^{15} instructions backward.

bczf — Branch if coprocessor Z is false

Format: `bczf label`

Description: Conditionally transfer control to the instruction at `label` if coprocessor's Z flag is false.

bczt — Branch if coprocessor Z is true

Format: `bczt label`

Description: Conditionally transfer control to the instruction at `label` if coprocessor's Z flag is true.

beq — Branch if equal

Format: `beq Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` = `Src2`.

beqz† — Branch if equal to zero

Format: `beqz Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` = 0.

bge† — Branch if greater or equal (signed)

Format: `bge Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` $\geq$ `Src2`. The contents are treated as signed numbers.

bgeu† — Branch if greater than or equal (unsigned)

Format: `bgeu Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` $\geq$ `Src2`. The contents are treated as unsigned numbers.

bgez — Branch if greater than or equal to zero

Format: `bgez Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` ≥ 0.

bgezal — Branch if greater than or equal to zero and link

Format: `bgezal Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` ≥ 0. Save the next instruction address in register 31.

bgt† — Branch if greater than (signed)

Format: `bgt Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` $>$ `Src2`. The contents are treated as signed numbers.

bgtu† — Branch if greater than (unsigned)

Format: `bgtu    Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` $>$ `Src2`. The contents are treated as unsigned numbers.

bgtz — Branch if greater than zero (signed)

Format: `bgtz    Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` > 0. The contents are treated as signed numbers.

ble† — Branch if less than or equal (signed)

Format: `blt    Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` $\leq$ `Src2`. The contents are treated as signed numbers.

bleu† — Branch if less than or equal (unsigned)

Format: `bltu    Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` $\leq$ `Src2`. The contents are treated as unsigned numbers.

blez — Branch if less than or equal to zero (signed)

Format: `bltz    Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` ≤ 0. The contents are treated as signed numbers.

blt† — Branch if less than (signed)

Format: `blt    Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` < `Src2`. The contents are treated as signed numbers.

bltu† — Branch if less than (unsigned)

Format: `bltu    Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` < `Src2`. The contents are treated as unsigned numbers.

bltz — Branch if less than zero (signed)

Format: `bltz    Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` < 0. The contents are treated as signed numbers.

bltzal — Branch if less than zero and link

Format: `bltzal    Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` < 0. Save the next instruction address in register 31.

bne — Branch if not equal

Format: `bne    Rsrc1,Src2,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc1` ≠ `Src2`.

bnez† — Branch if not equal to zero

Format: `bnez    Rsrc,label`

Description: Conditionally transfer control to the instruction at `label` if `Rsrc` ≠ 0.

div — Divide (signed)

Format: `div    Rsrc1,Rsrc2`

Description: Performs division of two signed numbers in `Rsrc1` and `Rsrc2` (i.e., `Rsrc1/Rsrc2`). The quotient is placed in register `lo` and the remainder in register `hi`. If an operand is negative, the remainder is unspecified by the MIPS architecture. The corresponding SPIM value depends on the machine it is running.

divu — Divide (unsigned)

Format: `div    Rsrc1,Rsrc2`

Description: Same as `div` above except that the numbers in `Rsrc1` and `Rsrc2` are treated as unsigned.

div† — Divide (signed)

Format: `div    Rdest,Rsrc1,Src2`

Description: Performs division of two signed numbers in `Rsrc1` and `Src2` (i.e., `Rsrc1/Src2`). The quotient is placed in register `Rdest`. `Src2` can be a register or a 16-bit immediate value.

divu† — Divide (signed)

Format: `divu    Rdest,Rsrc1,Src2`

Description: Same as the last `div` pseudoinstruction except that the numbers in `Rsrc1` and `Src2` are treated as unsigned.

j — Jump

Format: `j    label`

Description: Unconditionally transfer control to the instruction at `label`. Jump instruction uses a signed 26-bit offset. This allows jumps to $2^{25} - 1$ instructions forward, or 2^{25} instructions backward.

jal — Jump and link

Format: `jal    label`

Description: Unconditionally transfer control to the instruction at `label`. Save the next instruction address in register 31.

jalr — Jump and link register

Format: `jalr    Rsrc`

Description: Unconditionally transfer control to the instruction whose address is in `Rsrc`. Save the next instruction address in register 31.

jr — Jump register

Format: `jr    Rsrc`

Description: Unconditionally transfer control to the instruction whose address is in `Rsrc`. This instruction is used to return from procedures.

la† — Load address

Format: `la    Rdest,address`

Description: Loads `address` into register `Rdest`.

lb — Load byte (signed)

Format: `lb    Rdest,address`

Description: Loads the byte at `address` into register `Rdest`. The byte is sign-extended.

lbu — Load byte (unsigned)

Format: `lbu    Rdest,address`

Description: Loads the byte at `address` into register `Rdest`. The byte is zero-extended.

ld† — Load doubleword

Format: `ld    Rdest,address`

Description: Loads the doubleword (64 bits) at `address` into registers `Rdest` and `Rdest+1`.

lh — Load halfword (signed)

Format: `lh    Rdest,address`

Description: Loads the halfword (16 bits) at `address` into register `Rdest`. The halfword is sign-extended.

lhu — Load halfword (unsigned)

Format: `lhu    Rdest,address`

Description: Loads the halfword (16 bits) at `address` into register `Rdest`. The halfword is zero-extended.

li† — Load immediate

Format: `li    Rdest,Imm`

Description: Loads the immediate value `Imm` into register `Rdest`.

lui — Load upper immediate

Format: `lui    Rdest,Imm`

Description: Loads the 16-bit immediate `Imm` into the upper halfword of register `Rdest`. The lower halfword of `Rdest` is set to 0.

lw — Load word

Format: `lw    Rdest,address`

Description: Loads the word (32 bits) at `address` into register `Rdest`.

lwc$z$ — Load word from coprocessor z

Format: `lwcz    Rdest,address`

Description: Loads the word (32 bits) at `address` into register `Rdest` of coprocessor z.

lwl — Load word left

Format: `lwl    Rdest,address`

Description: Loads the left bytes from the word at `address` into register `Rdest`. This instruction can be used along with `lwr` to load an unaligned word from memory. The `lwl` instruction starts loading bytes from (possibly unaligned) `address` until the lower-order byte of the word. These bytes are stored in `Rdest` from the left. The number of bytes stored depends on the address. For example, if the address is 1, it stores the three bytes at addresses 1, 2, and 3. As another example, if the address is 2, it stores the two bytes at addresses 2 and 3. See `lwr` for more details.

lwr — Load word right

Format: `lwr    Rdest,address`

Description: Loads the right bytes from the word at `address` into register `Rdest`. This instruction can be used along with `lwl` to load an unaligned word from memory. The `lwr` instruction starts loading bytes from (possibly unaligned) `address` until the higher-order byte of the word. These bytes are stored in `Rdest` from the right. As in the `lwl` instruction, the number of bytes stored depends on the address. However, the direction is opposite of that used in the `lwl` instruction. For example, if the address is 4, it stores just one byte at address 4. As another example, if the address is 6, it stores the three bytes at addresses 6, 5, and 4. In contrast, the `lwl` instruction with address 6 would load the two bytes at addresses 6 and 7.

As an example, let us look at an unaligned word stored at addresses 1, 2, 3, and 4. We could use `lwl` with address 1 to store the bytes at addresses 1, 2, 3; the `lwr` with address 4 can be used to store the byte at address 4. At the end of this two-instruction sequence, the unaligned word is stored in `Rdest`.

mfc*z* — Move from coprocessor *z*

Format: `mfcz    Rdest,CPsrc`

Description: Move contents of coprocessor *z*'s register `CPsrc` to CPU register `Rdest`.

mfhi — Move from hi

Format: `mfhi    Rdest`

Description: Copy contents of `hi` register to `Rdest`.

mflo — Move from lo

Format: `mflo    Rdest`

Description: Copy contents of `lo` register to `Rdest`.

move† — Move

Format: `move Rdest,Rsrc`

Description: Copy contents of `Rsrc` to `Rdest`.

mtc$z$ — Move to coprocessor z

Format: `mtcz Rsrc,CPdest`

Description: Move contents of CPU register `Rsrc` to coprocessor z's register `CPdest`.

mthi — Move to hi

Format: `mfhi Rsrc`

Description: Copy contents of `Rdest` to `hi` register.

mtlo — Move to lo

Format: `mflo Rsrc`

Description: Copy contents of `Rdest` to `lo` register.

mul† — Signed Multiply (no overflow)

Format: `mul Rdest,Rsrc1,Src2`

Description: Performs multiplication of two signed numbers in `Rsrc1` and `Src2`. The result is placed in register `Rdest`. `Src2` can be a register or a 16-bit immediate value. No overflow exception is generated.

mulo† — Signed Multiply (with overflow)

Format: `mulo Rdest,Rsrc1,Src2`

Description: Performs multiplication of two signed numbers in `Rsrc1` and `Src2`. The result is placed in register `Rdest`. `Src2` can be a register or a 16-bit immediate value. If there is an overflow, an overflow exception is generated.

mulou† — Signed Multiply (with overflow)

Format: `mulou Rdest,Rsrc1,Src2`

Description: Performs multiplication of two unsigned numbers in `Rsrc1` and `Src2`. The result is placed in register `Rdest`. `Src2` can be a register or a 16-bit immediate value. If there is an overflow, an overflow exception is generated.

mult — Multiply (signed)

Format: `mult Rsrc1,Rsrc2`

Description: Performs multiplication of two signed numbers in `Rsrc1` and `Rsrc2`. The lower-order word of result is placed in register `lo` and the higher-order word in register `hi`.

multu — Multiply (unsigned)

Format: `multu Rsrc1,Rsrc2`

Description: Same as `mult` but treats the numbers as unsigned.

neg† — Negation (with overflow)

Format: `neg Rdest,Rsrc`

Description: Places the negative value of the integer in `Rsrc` in `Rdest`. This pseudo-instruction generates overflow exception.

negu† — Negation (no overflow)

Format: `neg Rdest,Rsrc`

Description: Places the negative value of the integer in `Rsrc` in `Rdest`. No overflow exception is generated.

nor — Logical NOR

Format: `nor    Rdest,Rsrc1,Src2`

Description: Places the logical NOR of `Rsrc1` and `Src2` in `Rdest`.

not† — Logical NOT

Format: `not    Rdest,Rsrc`

Description: Places the logical NOT of `Rsrc` in `Rdest`.

or — Logical OR

Format: `or    Rdest,Rsrc1,Src2`

Description: Places the logical OR of `Rsrc1` and `Src2` in `Rdest`.

ori — Logical OR immediate

Format: `ori    Rdest,Rsrc1,Imm`

Description: Places the logical OR of `Rsrc1` and `Imm` in `Rdest`.

rem† — Remainder (signed)

Format: `rem    Rdest,Rsrc1,Src2`

Description: Places the remainder from dividing two signed numbers in `Rsrc1` and `Src2` (`Rsrc1/Src2`) in register `Rdest`. `Src2` can be a register or a 16-bit immediate value. If an operand is negative, the remainder is unspecified by the MIPS architecture. The corresponding SPIM value depends on the machine it is running.

remu† — Remainder (unsigned)

Format: `remu    Rdest,Rsrc1,Src2`

Description: Same as `rem` except that the numbers are treated as unsigned.

rol† — Rotate left

Format: `rol    Rdest,Rsrc1,Src2`

Description: Rotates the contents of register `Rsrc1` left by the number of bit positions indicated by `Src2` and places the result in `Rdest`.

ror† — Rotate left

Format: `ror    Rdest,Rsrc1,Src2`

Description: Rotates the contents of register `Rsrc1` right by the number of bit positions indicated by `Src2` and places the result in `Rdest`.

sb — Store byte

Format: `sb    Rsrc,address`

Description: Stores the lowest byte from register `Rdest` at `address`.

sd — Store doubleword

Format: `sd    Rsrc,address`

Description: Stores the doubleword (64 bits) from registers `Rdest` and `Rdest`+1 at `address`.

seq† — Set if equal

Format: `seq    Rdest,Rsrc1,Src2`

Description: Set register `Rdest` to 1 if `Rsrc1` is equal to `Src2`; otherwise, `Rdest` is 0.

sge† — Set if greater than or equal (signed)

Format: `sge    Rdest,Rsrc1,Src2`

Description: Set register `Rdest` to 1 if `Rsrc1` is greater than or equal to `Src2`; otherwise, `Rdest` is 0. `Rsrc1` and `Src2` are treated as signed numbers.

sgeu† — Set if greater than or equal (unsigned)

Format: `sgeu    Rdest,Rsrc1,Src2`

Description: Same as `sge` except that `Rsrc1` and `Src2` are treated as unsigned numbers.

sgt† — Set if greater than (signed)

Format: `sgt    Rdest,Rsrc1,Src2`

Description: Set register `Rdest` to 1 if `Rsrc1` is greater than `Src2`; otherwise, `Rdest` is 0. `Rsrc1` and `Src2` are treated as signed numbers.

sgtu† — Set if greater than (unsigned)

Format: `sgtu    Rdest,Rsrc1,Src2`

Description: Same as `sgt` except that `Rsrc1` and `Src2` are treated as unsigned numbers.

sh — Store halfword

Format: `sh    Rsrc,address`

Description: Stores the lower halfword (16 bits) from register `Rsrc` at `address`.

sle† — Set if less than or equal (signed)

Format: `sle    Rdest,Rsrc1,Src2`

Description: Set register `Rdest` to 1 if `Rsrc1` is less than or equal to `Src2`; otherwise, `Rdest` is 0. `Rsrc1` and `Src2` are treated as signed numbers.

sleu† — Set if less than or equal (unsigned)

Format: `sleu    Rdest,Rsrc1,Src2`

Description: Same as `sle` except that `Rsrc1` and `Src2` are treated as unsigned numbers.

sll — Shift left logical

Format: `sll Rdest,Rsrc1,count`

Description: Shifts the contents of register `Rsrc1` left by `count` bit positions and places the result in `Rdest`. Shifted out bits are filled with zeros.

sllv — Shift left logical variable

Format: `sllv Rdest,Rsrc1,Rsrc2`

Description: Shifts the contents of register `Rsrc1` left by the number of bit positions indicated by `Rsrc2` and places the result in `Rdest`. Shifted out bits are filled with zeros.

slt — Set if less than (signed)

Format: `slt Rdest,Rsrc1,Src2`

Description: Set register `Rdest` to 1 if `Rsrc1` is less than `Src2`; otherwise, `Rdest` is 0. `Rsrc1` and `Src2` are treated as signed numbers.

slti — Set if less than immediate (signed)

Format: `slti Rdest,Rsrc1,Imm`

Description: Set register `Rdest` to 1 if `Rsrc1` is less than `Imm`; otherwise, `Rdest` is 0. `Rsrc1` and `Imm` are treated as signed numbers.

sltiu — Set if less than immediate (unsigned)

Format: `sltiu Rdest,Rsrc1,Imm`

Description: Same as `slti` except that `Rsrc1` and `Imm` are treated as unsigned numbers.

sltu — Set if less than (unsigned)

Format: `sltu Rdest,Rsrc1,Src2`

Description: Same as `slt` except that `Rsrc1` and `Src2` are treated as unsigned numbers.

sne† — Set if not equal

Format: `sne     Rdest,Rsrc1,Src2`

Description: Set register `Rdest` to 1 if `Rsrc1` is not equal to `Src2`; otherwise, `Rdest` is 0.

sra — Shift right arithmetic

Format: `sra     Rdest,Rsrc1,count`

Description: Shifts the contents of register `Rsrc1` right by `count` bit positions and places the result in `Rdest`. Shifted out bits are filled with the sign bit.

srav — Shift right arithmetic variable

Format: `srav    Rdest,Rsrc1,Rsrc2`

Description: Shifts the contents of register `Rsrc1` right by the number of bit positions indicated by `Rsrc2` and places the result in `Rdest`. Shifted out bits are filled with the sign bit.

srl — Shift right arithmetic

Format: `srl     Rdest,Rsrc1,count`

Description: Shifts the contents of register `Rsrc1` right by `count` bit positions and places the result in `Rdest`. Shifted out bits are filled with zeros.

srlv — Shift right arithmetic variable

Format: `srlv    Rdest,Rsrc1,Rsrc2`

Description: Shifts the contents of register `Rsrc1` right by the number of bit positions indicated by `Rsrc2` and places the result in `Rdest`. Shifted out bits are filled with zeros.

sub — Subtract with overflow

Format: `sub Rdest,Rsrc1,Src2`

Description: `Rdest` receives the difference of `Rsrc1` and `Src2` (i.e., `Rsrc1−SRc2`). The numbers are treated as signed integers. In case of an overflow, an overflow exception is generated.

subu — Subtract with no overflow

Format: `subu Rdest,Rsrc1,Src2`

Description: Same as `sub` but no overflow exception is generated.

sw — Store word

Format: `sw Rsrc,address`

Description: Stores the word from register `Rsrc` at `address`.

swc*z* — Store word coprocessor *z*

Format: `sw Rsrc,address`

Description: Stores the word from register `Rsrc` of coprocessor *z* at `address`.

swl — Store word left

Format: `swl Rsrc,address`

Description: Copies the left bytes from register `Rsrc` to memory at `address`. This instruction can be used along with `swr` to store a word in memory at an unaligned address. The `swl` instruction starts storing the bytes from the most significant byte of `Rsrc` to memory at `address` until the lower-order byte of the word in memory is reached. For example, if the address is 1, it stores the three most significant bytes of `Rsrc` at addresses 1, 2, and 3. As another example, if the address is 2, it stores the two most significant bytes of `Rsrc` at addresses 2 and 3.

swr — Store word right

Format: `swr Rsrc,address`

Description: Copies the right bytes from register `Rsrc` to memory at `address`. This instruction can be used along with `swl` to store a word in memory at an unaligned address. The `swr` instruction starts storing the bytes from the least significant byte of `Rsrc` to memory at `address` until the higher-order byte of the word in memory is reached. For example, if the address is 1, it stores the two least significant bytes of `Rsrc` at addresses 1 and 0. As another example, if the address is 2, it stores the three least significant bytes of `Rsrc` at addresses 2, 1 and 0.

ulh† — Unaligned load halfword (signed)

Format: `ulh Rdest,address`

Description: Loads the halfword (16 bits) from the word at `address` into register `Rdest`. The address could be unaligned. The halfword is sign-extended.

ulhu† — Unaligned load halfword (unsigned)

Format: `ulhu Rdest,address`

Description: Loads the halfword (16 bits) from the word at `address` into register `Rdest`. The address could be unaligned. The halfword is zero-extended.

ulw† — Unaligned load word

Format: `ulw Rdest,address`

Description: Loads the word (32 bits) at `address` into register `Rdest`. The address could be unaligned.

ush[†] — Unaligned store halfword

Format: `ush    Rsrc,address`

Description: Stores the lower halfword (16 bits) from register `Rsrc` at `address`. The address could be unaligned.

usw[†] — Unaligned store word

Format: `usw    Rsrc,address`

Description: Stores the word (32 bits) from register `Rsrc` at `address`. The address could be unaligned.

xor — Logical XOR

Format: `xor    Rdest,Rsrc1,Src2`

Description: Places the logical XOR of `Rsrc1` and `Src2` in `Rdest`.

xori — Logical XOR immediate

Format: `xori    Rdest,Rsrc1,Imm`

Description: Places the logical XOR of `Rsrc1` and `Imm` in `Rdest`.

Appendix G

ASCII Character Set

The next two pages give the standard ASCII (American Standard Code for Information Interchange) 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 675.

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

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	ˆ	7E	126	˜
3F	63	?	5F	95	_			

Note that 7FH (127 in decimal) is a control character listed on the previous page.

Index