UNDERGRADUATE TEXTS IN COMPUTER SCIENCE

Editors
David Gries
Fred B. Schneider

Springer Science+Business Media, LLC

UNDERGRADUATE TEXTS IN COMPUTER SCIENCE

Beidler, Data Structures and Algorithms

Bergin, Data Structure Programming

Brooks, Problem Solving with Fortran 90

Dandamudi, Introduction to Assembly Language Programming

Grillmeyer, Exploring Computer Science with Scheme

Jalote, An Integrated Approach to Software Engineering, Second Edition

Kizza, Ethical and Social Issues in the Information Age

Kozen, Automata and Computability

Merritt and Stix, Migrating from Pascal to C++

Pearce, Programming and Meta-Programming in Scheme

Zeigler, Objects and Systems

Sivarama P. Dandamudi

Introduction to Assembly Language Programming

From 8086 to Pentium Processors

With 63 Illustrations

Springer

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

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 : from 8086 to Pentium processors / Sivarama P. Dandamudi.
p. cm. — (Undergraduate texts in computer science)
Includes bibliographical references.
ISBN 978-1-4757-2860-6 ISBN 978-1-4757-2858-3 (eBook)
DOI 10.1007/978-1-4757-2858-3
1. Assembler language (Computer program language).
2. Microprocessing—Programming. I. Title. II. Series.
QA76.73.A8D36 1998
005.265–dc2 98-17534

Printed on acid-free paper.

Production managed by Anthony K. Guardiola; manufacturing supervised by Jeffrey Taub.
Photocomposed pages prepared from the author's TeX files.

9 8 7 6 5 4 3 2 1

ISBN 978-1-4757-2860-6

To
my wife Sobha
and
my daughter Veda

Preface

There are three main reasons for writing this book. While several assembly language books are on the market, almost all of them cover only the 8086 processor—a 16-bit processor Intel introduced in 1979. A modern computer organization or assembly language course requires treatment of a more recent processor like the Pentium, which is a 32-bit processor in the Intel family. This is one of the main motivations for writing this book.

There are two other equally valid reasons. The book approaches assembly language programming from the high-level language viewpoint. As a result, it focuses on the assembly language features that are required to efficiently implement high-level language constructs.

Performance is another reason why people program in assembly language. This is particularly true with real-time application programming. Our treatment of assembly language programming is oriented toward performance optimization. Every chapter ends with a performance section that discusses the impact of specific sets of assembly language statements on the performance of the whole program. Put another way, this book focuses on performance-oriented assembly language programming.

Intended Use

This book is intended as an introduction to assembly language programming using the Intel 80X86 family of processors. We have selected the assembly language of the Intel 80X86 processors (including the Pentium processor) because of the widespread availability of PCs and assemblers. Both Microsoft and Borland provide assemblers for the PCs.

Assembly language programming is part of an undergraduate curriculum 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 of the performance-oriented assembly language programming style advocated by the book, it is especially useful in real-time programming courses in engineering.

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

Prerequisites

The 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 rudimentary background in the software development cycle, as is obtained in a typical high-level programming course. Of course, the student is assumed to be familiar with the PC and its operating system.

Features

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

- The book is self-contained and does not assume background in computer organization. All necessary background material on computer organization is presented in the book.
- The book uses a methodical organization of chapters for a step-by-step introduction to the assembly language.
- The book covers all processors in the Intel 80X86 series (from 8086 to Pentium).
- 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 to provide the rationale as to why the instruction has been designed the way it is. This is the best way of understanding the strengths and weaknesses of the Intel 80X86 series of processors.
- Procedures are introduced early to encourage modular programming in developing assembly language programs.
- A set of input and output routines is provided so the student can focus on developing assembly language programs rather than spending time in understanding how input and output can be done using the basic I/O functions provided by the operating system.
- 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 about how these programs work.

- All examples and other required software are available online (check my home page for information) to give opportunities for students to perform hands-on assembly programming.
- Most chapters are written in such a way that each one can be covered in two or three 60-minute lectures by giving proper reading assignments. Typically, important concepts are emphasized in the lectures while leaving the other material in the book as a reading assignment. Our emphasis on extensive examples facilitates this pedagogical approach.
- Since performance is one of the two main objectives in using assembly language, each chapter contains a "Performance" section that discusses the performance implications of the topics/instructions discussed in that chapter. This aspect is important in those courses that deal with real-time programming. However, the performance section is completely independent and can be omitted altogether.
- Inter-chapter 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 book is divided into four parts. Part I presents introductory topics and consists of the first three chapters. Chapter 1 provides an introduction to assembly language and presents reasons for programming in assembly language. Chapter 2 presents basics of computer organization with a focus on the Intel 80X86 family of processors. In particular, this chapter gives sufficient details on the 16- and 32-bit Intel processors so the student can effectively program in assembly language. Chapter 3 gives an overview of assembly language. After covering these three chapters, one can write simple stand-alone assembly language programs without requiring further information from the other chapters. These three chapters should be covered in the sequence presented in the book. The amount of time spent on this part can vary depending on the background of the students.

Part II provides basic topics and consists of five chapters—Chapters 4 through 8. To emphasize the importance of modular programming, procedures are introduced early on (in Chapter 4). The other chapters in this part expand on the overview given in Chapter 3. Chapter 5 presents the addressing modes supported by the Intel 16- and 32-bit processors. This chapter also contains a detailed discussion of the motivation for providing the various addressing modes. Chapter 6 discusses the arithmetic instructions and the use of the flags register. Chapters 7 and 8 present conditional and bit manipulation instructions. A feature of these two chapters is that they explain how high-level language statements can be implemented using the instructions discussed in these two chapters. The first two chapters of this part—chapters 4 and 5—should be covered in some detail for proper grounding in assembly language programming. However, the remaining three chapters can be studied in any order. Also, the depth that these three chapters cover can be varied without sacrificing the effectiveness, depending on the time available and importance to the course objective.

The remaining five of the thirteen chapters constitute Part III. These chapters deal with advanced topics. Chapter 9 discuses the string processing instructions in detail. Macros and conditional assembly directives are discussed in Chapter 10. ASCII and BCD arithmetic instructions are presented in Chapter 11. Chapter 12 takes a detailed look at the interrupt mechanism and input/output interface. This is an important chapter in computer organization and real-time system programming courses. The final chapter deals with high-level language interface, which allows mixed-mode programming in more than one language. We use C and assembly language to cover the principles involved in mixed-mode programming. The chapters in this part can be covered in any order the instructor wishes. While most of the topics of this part are optional, a good, well-rounded course should cover some aspects of macros (Chapter 10), interrupts (Chapter 12), and high-level language interface (Chapter 13).

The five appendices provide a wealth of reference material needed by the student. 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. Debugging is discussed in Appendix C. Selected Pentium instructions are given in Appendix D. Finally, Appendix E 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 wife Sobha has taken care of our daughter so that I could spend more time on the book. My daughter Veda, who will be four in January 1999, contributed in her own way by allowing me to proofread some of the chapters while playing in her favorite park in Ottawa.

I want to thank Martin Gilchrist, former Senior Editor at Springer-Verlag, for his positive feedback on the initial proposal and Bill Sanders, Senior Editor, for his continuous and enthusiastic support for the project. I would also like to express my appreciation to the staff at the Springer-Verlag production department for converting my LaTeX files into the book in front of you. Some people directly involved with the book are Fred Bartlett, Anthony Guardiola, and Chrisa Hotchkiss.

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

Feedback

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

Ottawa, Canada
July 1998

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

Contents

Part I

Introductory Topics

Chapter 1

Introduction

Objectives

- To introduce assembly language and to explain where it fits in the hierarchy of computer languages
- To discuss the advantages and disadvantages associated with programming in assembly language
- To provide motivation to learn assembly language
- To demonstrate the performance advantages of 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 (e.g., a spreadsheet or 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 PASCAL can be used to develop modular programs. These languages provide several high-level constructs (`if-then-else`*,* `while`*, etc.) that aid in faster program development and maintenance. After giving a brief introduction to assembly language in Section 1.2, we will discuss the main advantages of the high-level languages in Section 1.3. The advantages of programming in assembly language are highlighted in Section 1.4.*

Section 1.5 identifies some typical application areas that benefit from programming in assembly language. Section 1.6 discusses some reasons for learning assembly language.

The performance advantage of programming in assembly language over programming in C is demonstrated in Section 1.7. A summary of the chapter is given in the last section.

S. P. Dandamudi, *Introduction to Assembly Language Programming*
© Springer Science+Business Media New York 1998

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 simply limited to the interface provided by an application software such as spreadsheet, word processor, etc. The user is expected to have only a rudimentary knowledge of how to operate the system. Problem solving at this level, for example, might be composing a letter by using a word processor application software.

At the next level, problem solving is done in one of the *high-level languages* such as C, PASCAL, FORTRAN, BASIC, etc. A user interacting with the system at this level should have a detailed knowledge of software development using a high-level language. 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, have a very detailed knowledge about the system (unless they are also involved in developing system software such as device drivers, assemblers, or operating systems).

Both levels 4 and 5 are *system independent*, i.e., independent of the particular processor (CPU) used in the system. For example, an application program written in C can be executed on a system based on an Intel 80X86 CPU or on a Motorola 680X0 CPU without any modification to the source code. All you have to do is recompile the program with a C compiler native to the target system. In contrast, software development done at all levels below level 4 is *system dependent.*

Assembly language programming is also 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 particular CPU used in the system. For example, a program written in the 80X86 assembly language cannot be executed on a system based on a 680X0 CPU. Programming in assembly language also requires a detailed knowledge about the system components such as the CPU, memory, and so on.

Machine language is a close relative of the assembly language. Typically, there is a one-to-one correspondence between the instructions of assembly language and the corresponding machine language. The CPU only understands the machine language, whose instructions consist of a string of 1's and 0's. More on the assembly and machine languages will be said in the next section.

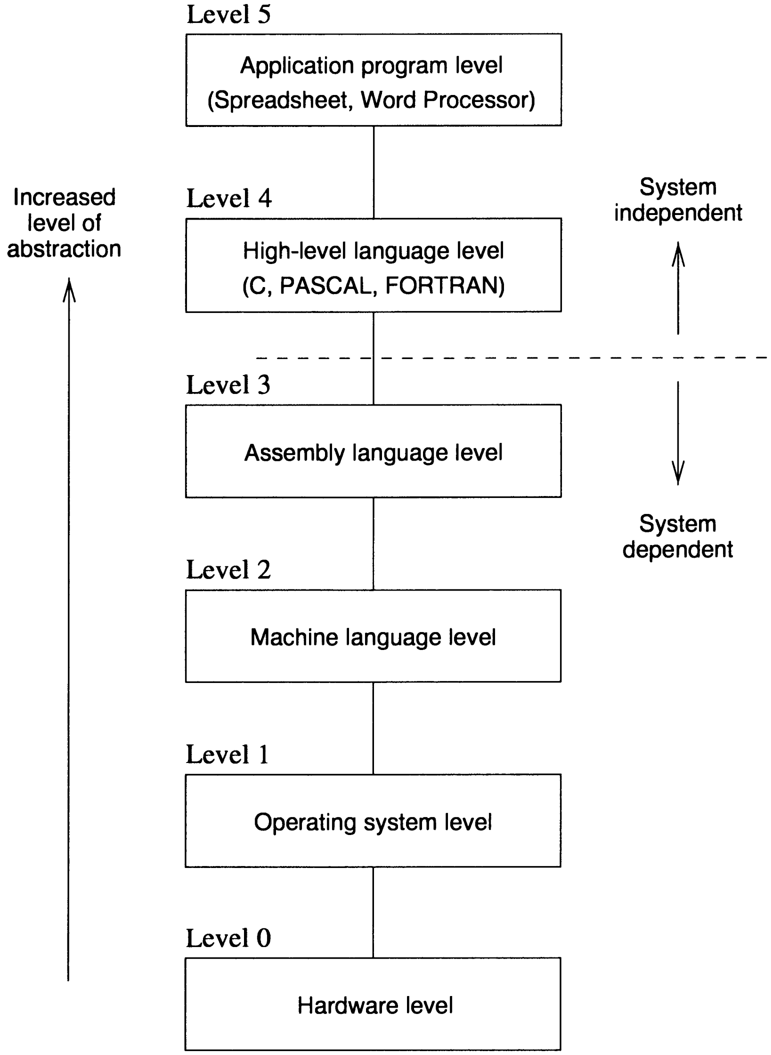


Figure 1.1 A user's view of various levels of a computer system.

Even though assembly language is considered to be a low-level language, programming in assembly language will not expose you to all the nuts and bolts of the system. Your operating system (e.g., DOS) hides several of the low-level details so that the assembly language programmer can breath easy.

For example, if you want to read the input given from the system keyboard, you can rely on the services provided by your operating system to do the job.

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.

1.2 What Is Assembly Language?

Assembly language is a low-level programming language with a one-to-one correspondence between its instructions and the machine language instructions of a CPU. Assembly language is directly influenced by the instruction set and architecture of a CPU. The assembly language code must be processed by an assembler in order to generate the machine language code. An *assembler* is a program that translates assembly language code into machine language. MASM (Microsoft Assembler) and TASM (Borland Turbo Assembler) are the two popular assemblers for the PC.

Assembly language instructions specify low-level tasks (hence, the name low-level language). Here are some examples from the assembly language of the Intel 80X86 CPU.

```
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.

The above examples illustrate several points:

1. Assembly language instructions are cryptic.
2. Assembly language operations are expressed by using mnemonics (like and, inc, add etc.).
3. Assembly language instructions are low level. For example, you cannot write the following:

   ```
   mov   class_size, value
   add   marks, 10.8
   ```

   The first instruction is invalid because two variables class_size and value cannot be used in a single instruction. The second instruction is not a valid assembly language instruction because real (i.e., fractional) numbers cannot be used.

You would appreciate the readability of the assembly language instructions when you look at the equivalent machine language instructions. Here are some examples:

Assembly language		Machine language (in hex)
inc	result	FF060A00
mov	class_size, 45	C7060C002D00
and	mask, 128	80260E0080
add	marks, 10	83060F000A

In the above table, machine language instructions are written in the hexadecimal number system. If you are not familiar with the hexadecimal number system, consult Appendix A for a detailed discussion of various number systems. 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 assembly language to the machine language. *Assembler* is the software that achieves this code translation. As a result, only a masochist would consider programming in a machine language.

1.3 Advantages of High-Level Languages

High-level languages such as C 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 rather than in an assembly language.

1. *Program development is faster in a high-level language.* Many high-level languages provide structures (sequential, selection, iterative) that facilitate program development. Programs written in a high-level language are relatively small (compared to the equivalent programs written in an assembly language) and easier to code and debug.

2. *Programs written in a high-level language are easier to maintain.* Programming for a new application can take several weeks to several months and the 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 written in a high-level language are portable.* High-level language programs contain very few machine-dependent details, and they can be used with little or no modification on different computer systems. In contrast, assembly language programs are written for a particular system and cannot be used for a different system.

To illustrate the differences between programs written in C and assembly languages, Section 1.7 presents a concrete example that sorts an array of numbers using the bubble sort algorithm. You can get an idea of how readable and compact the code written in C is by comparing the `bubble_sort` procedure written in C (see Program 1.2) and assembly language (see Program 1.3). A more detailed discussion is deferred until Section 1.7.

1.4 Why Program in Assembly Language?

While the previous discussion enumerated the disadvantages of assembly languages, there are certain advantages associated with programming in an assembly language.

There are two main reasons why programming is still done in assembly language: (i) efficiency, and (ii) accessibility to system hardware.

Efficiency refers to how "good" a program is in achieving a given objective. Here we consider two objectives based on space (space-efficiency) and time (time-efficiency).

Space-efficiency refers to the memory requirements of a program (i.e., the size of the code). Program A is said to be more space-efficient if it takes less memory space than program B to perform the same task. Very often, programs written in an assembly language tend to be more compact than when written in a high-level language. You should not confuse the size of source code with that of the executable code (see Section 1.7 for an example).

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. Programs written in an assembly language tend to run faster than those written in a high-level language. Section 1.7 demonstrates this advantage of assembly language through an example.

As an aside, note that 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 *programmer productivity*, and assembly language loses the battle to high-level languages as discussed before.

The superiority of assembly language in generating compact code is becoming increasingly less important for several reasons. First, the savings in space pertain only to the program code and not to its data space. Thus, depending on the application, the savings in space obtained by converting an application program from some high-level language to an assembly language may not be substantial. Second, the cost of memory (i.e., cost per bit) has been decreasing and memory capacity has been increasing. Thus, the size of a program is not a major hurdle anymore. Finally, compilers are becoming "smarter" in generating code that is both space- and time-efficient. However, there are areas such as embedded controllers in which space-efficiency is important (see also Section 1.5).

One of the main reasons for writing programs in assembly language is to generate a code that is time-efficient. The superiority of assembly language programs in producing a code that runs faster is a direct manifestation of *specificity*. That is, assembly language programs contain only the necessary code to perform the given task. Even here, a "smart" compiler can optimize the code that can compete well with its equivalent written in an assembly language. Although the gap is narrowing with improvements in compiler technology, assembly language still retains its advantage for now.

The other main reason for writing programs in an assembly language 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 an interface program (called device driver) to a new printer on the market almost certainly requires programming in an assembly language. Since assembly language does not impose any restrictions, you can have direct control over all of the system hardware. If you are developing system software (e.g., assembler, linker), you cannot avoid writing programs in assembly language.

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

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

Accessibility to hardware: Systems software often requires direct control over the system hardware. Examples include operating systems, assemblers, compilers, linkers, loaders, device drivers, and network interfaces.

Some applications require hardware control as well. The most notable example is video games. Another example is computer animation.

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 of a program is also important in spacecraft control systems.

1.6 Why Learn Assembly Language?

Programming in assembly language is a tedious and error-prone process. The natural preference of a programmer is to program the application in some high-level language. However, there are some good reasons why some applications cannot be programmed in a high-level language. Even the applications that require coding in assembly language do not require the whole program to be written in assembly language. In such instances, part of the program can be written in assembly language and the rest can be written in some high-level language. Such programs are referred to as *hybrid programs* or *mixed-mode programs*. Very often, programs that require assembly language are actually hybrid programs. In Chapter 13, we will discuss how you can write hybrid programs.

Learning assembly language has both practical and educational purposes. Even if you do not plan to write in an assembly language, studying it provides a good understanding of computer systems. When you program in a high-level language such as C, you are shielded from low-level details on purpose and provided only a "black-box" view of the system. When programming in assembly language, you need to understand the internal details of the system (how data is stored, how code can be made time-efficient, and so on). To understand assembly language is to understand the computer system itself!

This book uses the PC to explore these internal details. The reason for this is that PCs are popular and their architecture encompasses several important characteristics to provide a good understanding of some fundamental concepts, yet simple enough to provide a gentle introduction to computers beyond the black-box view.

A final reason to learn assembly language is the personal satisfaction that comes with learning something complex. Sure, learning assembly language is more difficult than learning C. But assembly language gives you complete control over the system hardware. It is very easy to write a simple program in assembly language that can crash the system. Try to achieve the same with a high-level language! You feel powerful with assembly language on your side, making the time spent learning assembly language worth your while. The insights provided by assembly language would benefit you even when you are programming in some high-level language.

1.7 Performance: C Versus Assembly Language

We stated in Section 1.4 that one of the main reasons for programming in an assembly language is to produce a code that runs faster than the corresponding

```
      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 1.2 Actions taken during the first pass of the bubble sort algorithm.

code produced by a high-level language compiler. In this section, we will see how much better we can do by writing programs in assembly language. As an example, let us consider the problem of sorting an array of numbers. Our strategy is to write a sort procedure in C (a representative high-level language) and in the 80X86 assembly language and compare the time required to sort the array by these two versions.

There are several algorithms to sort an array of numbers. The particular algorithm that we are using here is called the *bubble sort* algorithm. We describe the algorithm next.

The bubble sort algorithm consists of several passes through the array of numbers to be sorted in ascending order. 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. If at least a single swap is done during a pass, it will initiate another pass to scan the array.

Figure 1.2 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 1.3 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

```
       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 1.3 Behavior of the bubble sort algorithm.

on. This is why this algorithm is called the bubble sort: during the first pass, the largest element bubbles to the top, the second largest bubbles to the top during the second pass, and so on. Even though the array is in sorted order after the third pass, one more pass is required by the algorithm to detect that the array is sorted.

The number of passes required to sort an array depends on how unsorted the initial array is. If the array elements are already in sorted order, only a single pass is required. At the other extreme, if the array is completely unsorted (i.e., elements are initially in the descending order), the algorithm requires a number of passes equal to one less than the number of elements in the array.

The main program is shown in Program 1.1 (see page 17). To avoid the influence of I/O, we time only the sort procedure. To do this, we use `clock()` available in C. This function 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` macro. Thus, to obtain the sort time in seconds, we have to divide the clock ticks by `CLOCKS_PER_SEC`.

The `bubble_sort` procedure given in Program 1.2 (page 18) follows directly the algorithm described here. In our experiments, the array is initialized in descending order so that the maximum number of passes is required by the bubble sort algorithm.

In the assembly language version of the program, only the `bubble_sort` procedure is written in the assembly language. The assembly language version of the bubble sort procedure is shown in Program 1.3 on page 19. At this time, you are not expected to make any sense out of this program. The purpose is to show the complexity of the assembly language programs.

Space-efficiency

The executable file sizes of the C and assembly language versions of the bubble sort program are as follows:

C version:	50,256 bytes
Assembly language version:	50,208 bytes

As you can see from the above data, there is only a marginal improvement! This is mainly because the main program is written in C. In contrast, the source file sizes (shown below) of the bubble sort procedure written in C and assembly language show the low-level nature and complexity of programming in assembly language.

Bubble sort procedure source code length:

C version:	1,340 bytes
Assembly language version:	1,851 bytes

Time-efficiency

The sort time taken by the C and assembly language versions is shown in Figure 1.4. The programs were run under Borland C++ on an 80486DX2-based system with a 66 MHz clock. The x-axis is the size of the array and the y-axis gives the sort time in seconds. Notice that the assembly language version runs substantially faster and substantiates our claim that programs written in assembly language are time-efficient. Be cautioned that the improvement obtained by writing in assembly language depends on the application, compiler, and the type of processor, etc.

In practice, assembly language programming is limited to critical sections of a program. When we say critical, we mean either due to application (as in real-time applications), or due to performance reasons. For example, if two sort utilities are on the market—one written in C and the other in assembly language—the C version would be a commercial flop. Thus, converting the bubble sort procedure into assembly language is beneficial and justifies the increased program development cost. On the other hand, writing a procedure `init_array` to initialize the array in assembly language is a waste of time and effort, as this procedure is called only once and, therefore, is not a critical procedure.

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 such as C vis-a-vis 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.

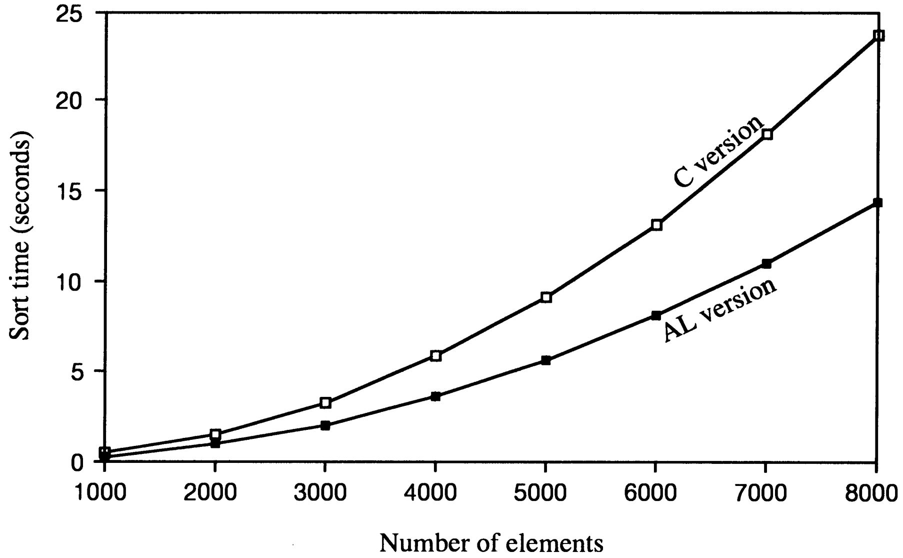


Figure 1.4 Sort time comparison of the bubble sort example: C version uses the `bubble_sort` procedure shown in Program 1.2; Assembly language (AL) version replaces the `bubble_sort` procedure by the assembly language procedure shown in Program 1.3.

In the final section of the chapter, we used the bubble sort example to illustrate the advantages and disadvantages of programming in assembly language.

1.9 Exercises

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

1–2 You are acting as a consultant to New Age Appliances, Inc. The company is bringing out a new dishwasher. You are asked to design the control software that should include as many features as possible in order to gain marketing advantage over competition. Which programming language would you choose and why?

1–3 From the example programs given in Program 1.2 and Program 1.3, you can see that the assembly language programs are long and complex. Why, then, should we learn to program in assembly language?

1–4 What is the relationship between assembly language and machine language? Under what circumstances, if any, do you consider programming in machine language?

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

1–6 Why is portability of programs important? When portability is important, which language—C or assembly language—would you use?

1–7 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.

1–8 Assume that the array used in the bubble sort program is initialized as `5 1 2 3 4`. How many passes over the array are needed for the bubble sort algorithm to sort the array in ascending order?

1–9 The bubble sort algorithm discussed in Section 1.7 sorts elements in ascending order. How difficult is it to change this algorithm to sort in descending order? Suggest the specific changes required to the algorithm.

1–10 We have stated that assembly language programs tend to produce code that is space-efficient. However, if you see the code for the bubble sort procedure, the C version is about a page (see page 18), while the assembly language version is twice as long (see Program 1.3). Explain the apparent contradiction.

1.10 Progamming Exercises

1–P1 Compile and run the C and assembly language versions of the bubble sort program on your machine. Compare the timing obtained on your machine with the data presented in Figure 1.4. After reading Chapter 2, you should be able to identify some of the reasons for the difference in the sort times.

1–P2 The objective of this exercise is to study the overhead associated with procedure calls in C. Towards this end, modify the C bubble sort procedure so that, instead of swapping elements within the procedure, it calls a procedure—say `swap`—to exchange two elements. A call to this procedure

```
swap (&x[i], &x[i+1]);
```

replaces the following three lines of code in Program 1.2 given on page 18.

```
temp = x[i];
x[i] = x[i+1];
x[i+1] = temp;
```

Compare the sort times of the new program with the sort times obtained in the last exercise for the original C version. In a later chapter, we will look at procedure call overhead in assembly language (see Chapter 4).

1.11 Program Listings

This section gives the source code listings of

`bblsortm.c`	main program
`bblsortc.c`	bubble sort procedure—C version
`bblsorta.asm`	bubble sort procedure—Assembly language version

Compilation is straightforward. For example, the command

```
bcc bblsortm.c bblsortc.c
```

can be used to compile the C version under Borland C++. Similarly,

```
bcc bblsortm.c bblsorta.asm
```

generates the assembly language version.

Program 1.1 bblsortm.c program listing

```
/*****************************************************************
 * This program initializes an array in descending order and     *
 * uses the bubble sort algorithm to sort the array in ascending *
 * order. Array size is given as input to the program.           *
 *****************************************************************/
#include        <stdio.h>
#include        <time.h>

#define ARRAY_SIZE   8000

extern void bubble_sort (int*, int);

int main(void)
{
        clock_t start, finish;
        int value[ARRAY_SIZE];
        int i, size;

        printf ("Please input the array size: ");
        scanf("%d", &size);

        /* initialize the array in descending order */
        for (i=0; i<size;i++)
            value[i] = size-i;

        start = clock();
        bubble_sort (value, size);
        finish = clock();

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

        return 0;
}
```

Program 1.2 bblsortc.c procedure listing

```
/************************************************************
 * This procedure uses the bubble sort algorithm to sort an *
 * array of integers in ascending order. The procedure      *
 * receives the array and its size as parameters.           *
 ************************************************************/

 #define  UNSORTED  0
 #define  SORTED    1

 void bubble_sort (int x[], int size)
 {
        int  state;     /* records SORTED/UNSORTED status   */
        int  end_index; /* number of comparisons to be done */
        int  i, temp;

 /* Assume that the whole array is initially unsorted */
        state = UNSORTED;
        end_index = size;

        while (state == UNSORTED)
        {
            state = SORTED;     /* Now prove that the array
                                   is not sorted            */
            end_index--;
            for (i=0; i<end_index; i++)    /* pass loop */
            {
                /* Look at adjacent pairs of elements and swap
                   if the second element is less than the first */
                if (x[i] > x[i+1])
                {
                    temp = x[i];
                    x[i] = x[i+1];
                    x[i+1] = temp;
                    state = UNSORTED;
                }
            }
        }
}
```

Program 1.3 bblsorta.asm procedure listing

```
COMMENT |         Bubble sort procedure     BBLSORTA.ASM
        Objective: To implement the bubble sort algorithm
           Inputs: A pointer to the array to be sorted
                   and its size are received via the stack.
           Output: Returns nothing but the array is sorted
|                  in ascending order.
SORTED    EQU   0
UNSORTED  EQU   1

.MODEL SMALL
.CODE
.486
PUBLIC _bubble_sort
_bubble_sort    PROC
        ; save registers used by the procedure
        pusha
        mov     BP,SP

        ;CX serves the same purpose as the end_index variable
        ; in the C procedure. CX keeps the number of comparisons
        ; to be done in each pass. Note that CX is decremented
        ; by 1 after each pass.
        mov     CX, [BP+20]  ; load array size into CX
        mov     BX, [BP+18]  ; load array address into BX

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

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

        ;SI points to element X and SI+2 to the next element
        mov     SI,BX   ; load array address into SI
pass:
        ;This loop represents one pass of the algorithm.
        ;Each iteration compares elements at [SI] and [SI+2]
        ; and swaps them if ([SI]) < ([SI+2]).
        mov     AX,[SI]
        cmp     AX,[SI+2]
        jg      swap
increment:
        ;Increment SI by 2 to point to the next element
        add     SI,2
        dec     DI
        jnz     pass

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

swap:
        ; swap elements at [SI] and [SI+2]
        xchg    AX,[SI+2]
        mov     [SI],AX
        mov     DX,UNSORTED        ; set status to UNSORTED
        jmp     SHORT increment

done:
        ; restore registers
        popa
        ret
_bubble_sort    ENDP
        END
```

Chapter 2

Basic Computer Organization

Objectives

- To provide a high-level view of computer organization
- To describe the organization of the Intel Pentium processor
- To introduce the memory organization of Pentium
- To discuss briefly 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 underlying 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 (I/O) devices.

The next three sections discuss these three components in detail. Section 2.2 discusses the architecture of the Intel Pentium processor. Sufficient details are presented here to understand the basic organization of the Pentium processor.

Section 2.3 presents some basic concepts about the memory system. Pentium memory organization is described in Section 2.4. It is important for an assembly language programmer to understand the segmented memory organization supported by Pentium.

Section 2.5 gives a brief overview of how input/output devices such as a keyboard, display screen, printer, etc. are interfaced to the system. Chapter 12 gives further details on I/O interfacing.

S. P. Dandamudi, *Introduction to Assembly Language Programming*

Section 2.6 discusses how data alignment affects the running time of programs. We use the bubble sort example discussed in Chapter 1 to illustrate the impact of data alignment. Section 2.7 concludes the chapter with a summary.

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. 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 2.1 shows details of how they are interconnected and what actually constitutes the system bus. As shown in Figure 2.1, the three major components of the system bus are the address bus, data bus, and control bus.

The width of address bus determines the amount of physical memory addressable by the processor. The width of 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}, or 1 MB, and each data transfer involves at most 16 bits. The Pentium, on the other hand, has 32 address lines and 64 data lines. Thus, Pentium can address up to 2^{32}, or a 4 GB memory. Furthermore, each data transfer can move 64 bits of data.

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, display screen, printer, modem, etc. 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 transfer. Such data transfers are called *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, 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 CPU wants to read data from the memory, it initiates a bus

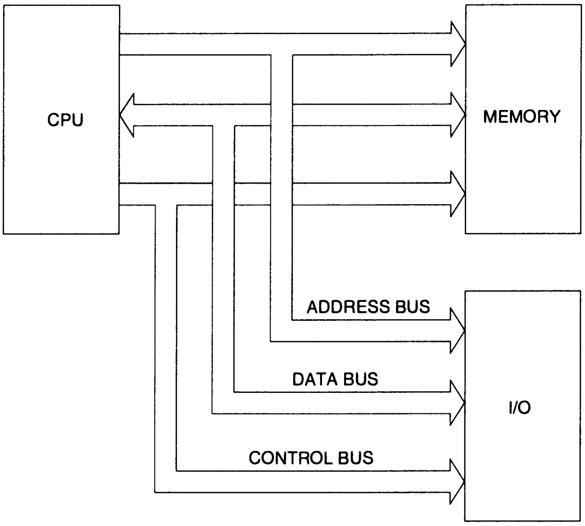


Figure 2.1 Simplified block diagram of a computer system.

transaction, also called a bus cycle, in which the CPU is the bus master and memory is the slave. The CPU 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 *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 it.

2.2 The Processor

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

1. Fetch an instruction from the memory
2. Decode the instruction (i.e., find out what the instruction is)
3. Execute the instruction (i.e., perform the action specified by the instruction).

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

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

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). The instructions that we write in whatever language will eventually be translated by a compiler or assembler to an equivalent sequence of machine language instructions that the CPU understands.

The operating system, which provides instructions to the CPU 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 CPU and instructs it to execute the program.

2.2.1 The Pentium Processor

The CPU is the heart of a computer system. The particular CPU used by a computer system determines the power and personality of the system. The goal of this section is to provide enough details on the Pentium processor that you need to know to program in the assembly language. We do not attempt to provide complete details of Pentium, as most aspects of the internal details are unimportant to the assembly language programmer. As indicated in the last section, a program is executed by the CPU by repeatedly performing the fetch-execute cycle shown in Figure 2.2.

Fetching an instruction from the main memory involves placing the appropriate address on the address bus and activating the memory read control 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 CPU, after instructing the memory unit to read, waits until the instruction is available on the data bus and then reads the instruction.

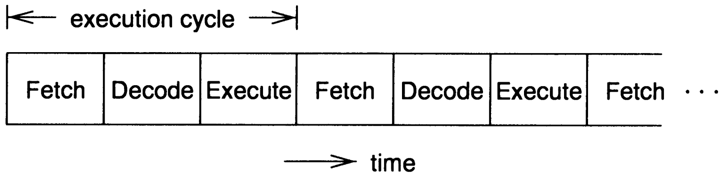


Figure 2.2 Execution cycle of a typical computer system.

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.

To *execute* an instruction, the CPU 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, Pentium provides a 16 KB on-chip cache. This is divided equally into data cache and code cache. The presence of on-chip cache is transparent to application programs—it helps improve application performance.

2.2.2 The Pentium Registers

Pentium provides several internal registers for the storage of data, control, and other information. Pentium has ten 32-bit and six 16-bit registers. These registers are grouped into general, control, and segment registers. The general registers are further grouped into data, pointer, and index registers.

Data Registers

There are four 32-bit data registers that can be used for arithmetic and logical operations (see Figure 2.3). These four registers are unique in that they can be used as:

- four 32-bit registers (EAX, EBX, ECX, EDX), or
- four 16-bit registers (AX, BX, CX, DX), or

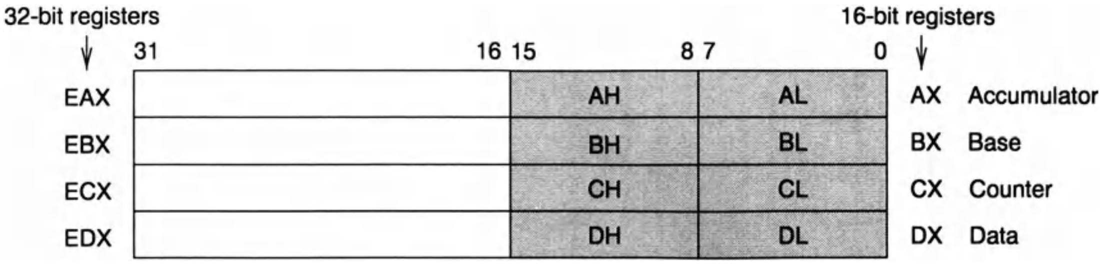


Figure 2.3 Data registers of the Pentium processor (16-bit registers are shown shaded).

- eight 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL)

As shown in Figure 2.3, 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 additional special functions when executing some specific instructions. For example, when performing a multiplication operation, one of the two data items needed should be in the EAX, AX, or AL register depending on whether the operation is on 32-bit, 16-bit, or 8-bit data items. Similarly, the ECX or CX register is assumed to contain the loop count value for the iterative instructions.

Pointer and Index Registers

Figure 2.4 shows the four 32-bit registers in this group. These registers can be used either as 16-bit registers or 32-bit registers. The two index registers play a special role in string processing instructions (discussed in Chapter 9). 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 stack is discussed in Chapter 4.

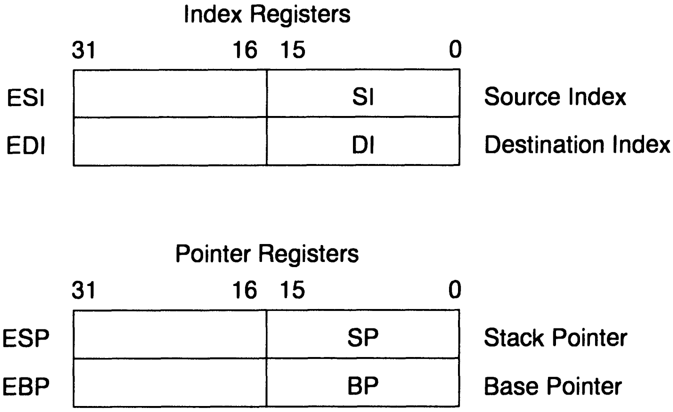


Figure 2.4 Index and pointer registers of the Pentium processor.

Control Registers

This group of registers consists of two 32-bit registers: the instruction pointer register and the flags register. The instruction pointer register is used by the processor to keep track of the location of the next instruction to be executed in the memory. In other words, the instruction pointer is kind of a marker to remember where the next instruction is located. 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 Section 2.4 for details on memory architecture).

When an instruction is fetched from memory, the instruction pointer is incremented 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 instruction, procedure call, or an interrupt).

The flags register can also 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 six *status* or *arithmetic* flags, one *control* flag, and ten *system* flags, as shown in Figure 2.5. Bits of this register can be set (to 1) or cleared (to 0). Pentium provides instructions to set or clear some flags. For example, the `clc` instruction clears the carry flag, while the `stc` instruction sets it.

The six status flags record certain information about the most recent arithmetic or logical operation. For example, if an arithmetic operation such as

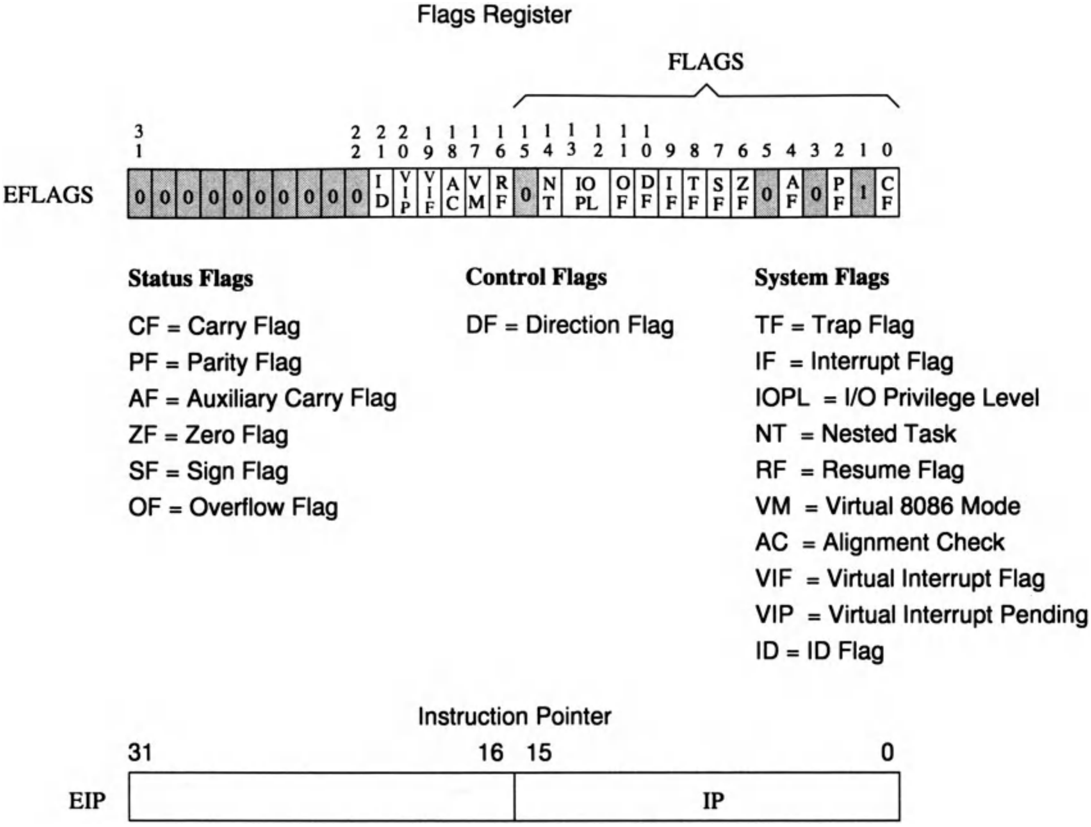


Figure 2.5 Flags and instruction pointer registers of the Pentium processor.

subtraction has resulted in a zero result, the zero flag (ZF) bit would be set (i.e., ZF = 1). Chapter 6 discusses the status flags in detail.

The control flag is useful in string operations. This determines whether a string operation is to scan the string in the forward or backward direction. The function of the direction flag is described in Chapter 9, which discusses the string instructions supported by Pentium.

The ten system flags control the operation of the processor. A detailed discussion of all ten system flags is beyond the scope of this book. Here we discuss a few flags in this group that are relevant to our objective. 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

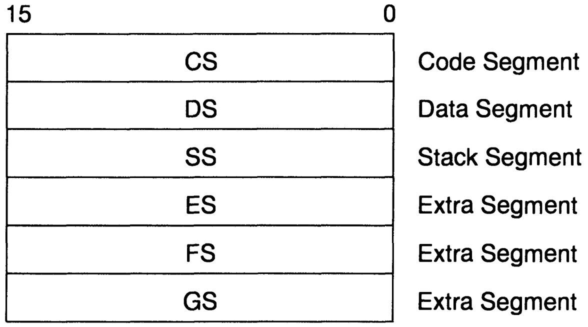


Figure 2.6 The six segment registers of the Pentium processor.

in debugging programs. These two flags are covered in Chapter 12, which discusses the interrupt processing mechanism of 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 "GenuineIntel" string), processor family, model, etc. The virtual-8086 mode (VM) flag, when set, emulates the programming environment of the 8086 processor.

The last flag that we discuss is the alignment check (AC) flag. When this flag is set, the processor operates in alignment check mode and generates exceptions when a reference is made to an unaligned memory address. Section 2.6 provides further information on data alignment and its impact on application performance.

Segment Registers

The six 16-bit segment registers of Pentium are shown in Figure 2.6. These registers support the segmented memory organization of Pentium. This memory organization is discussed in detail in Section 2.4. In such a segmented organization, memory is partitioned into segments, where each segment is a small part of the memory. The processor, at any point in time, can only access up to six segments of the main memory. The six segment registers point to where these segments are located in the memory.

Your program is logically divided into two parts: a code part that contains only the instructions, and a data part that contains only the data operated on by the instructions in the code part. The code segment (CS) register points to where your instructions are stored in the main memory, and the data segment

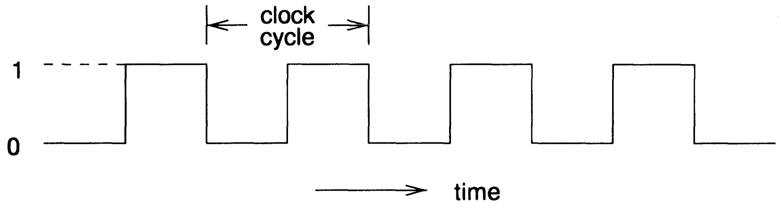


Figure 2.7 Clock signal of a computer system.

(DS) register points to your data segment location. The stack segment (SS) register points to your program's stack segment (discussed in Chapter 4).

The last three segment registers—ES, GS, and FS—are additional segment registers that can be used in a similar way as the other segment registers. For example, if a program's data could not be fit into a single data segment, it is efficient to use two data segment registers to point to the two data segments that your program uses.

2.2.3 The System Clock

System clock provides timing signal to synchronize the operation of the system. A clock is a sequence of 1's and 0's, as shown in Figure 2.7. Clock rate is measured in number of cycles per second. This number is referred to as Hertz (Hz). The abbreviation MHz is used for millions of cycles per second.

The system clock defines the *speed* at which the system is operating. All operations of the processor 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.

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

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

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

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

If it takes three clock cycles to move data on the system bus (for example, reading data from a memory location), it takes 3×10 ns = 30 ns.

One way to increase the speed of a computer system is to use a higher clock rate. For example, if we use a clock of 200 MHz, the time to move a unit of data on the system bus reduces from 30 ns to 15 ns. Clock rates increase with improvements in technology. The original IBM PC used a clock rate of 4.77 MHz. Current technology allows clock rates higher than 400 MHz.

2.2.4 The Intel 80X86 Processor Family

Intel introduced the 8086 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. While the 8088 also has a 20-bit address bus, it uses only an 8-bit data bus. The 8088, however, uses an internal 16-bit data bus. The 8088 has a 4-byte instruction queue as opposed to a 6-byte queue of the 8086.

The 80186 is a faster version of the 8086. It 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 and hence a 16 MB memory address space. The data bus is still 16 bits wide, but the 80286 has memory protection capabilities. It is backward-compatible in that it can run all of the original 8086-based software.

Intel introduced the first 32-bit CPU—the 80386—in 1985. It has a 32-bit data bus and a 32-bit address bus. The memory address space has grown substantially from that of the 80286 (from a 16 MB address space to 4 GB). Like the 80286, it can run all the programs designed to run on 8086 and 8088 CPUs. In the following year, Intel introduced the 80386SX. The 80386SX is essentially the same as the 80386 except that it has a 16-bit data bus instead of a 32-bit data bus. In other words, 80386SX is to 80386 what 8088 is to 8086.

The Intel 80486 was introduced in 1989. This is an improved version of the 80386. While maintaining the same address and data buses, it combines the coprocessor functions for performing floating-point arithmetic and includes an internal cache.

The latest in the family is the Pentium series. It is not named 80586 because Intel found belatedly that numbers cannot be trademarked! The first Pentium was introduced in 1993. Pentium is similar to 80486 but uses a 64-bit data bus. However, the instruction set of Pentium supports 32-bit operands like that of the 80486. Systems based on Pentium would fall into what is called the "workstation" category.

The number-crunching capability of a CPU can be enhanced by using special hardware to perform numeric operations. The 80X87 numeric coprocessor was designed to work with the 80X86 family CPUs to enhance the number processing capabilities. The 8087 numeric coprocessor works with the 8086/8088 to provide extensive high-speed numeric processing capabilities. The 8087, for example, provides about a hundred-fold improvement in execution time com-

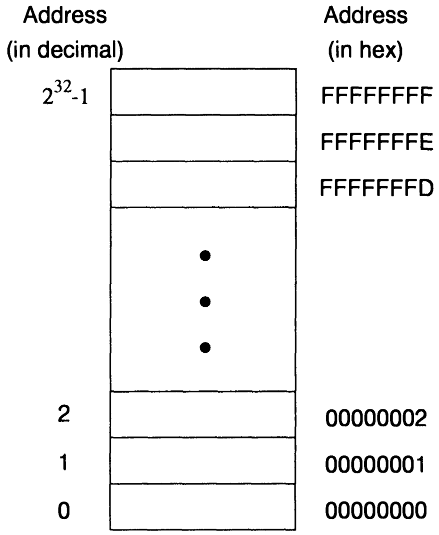


Figure 2.8 Logical view of the system memory.

pared to that of an equivalent function in software on a 5 MHz 8086. The 80287 works with the 80286 and the 80387 with the 80386. The 80486 and Pentium have built-in numeric processor capabilities and therefore do not need a special numeric processor.

2.3 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 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.

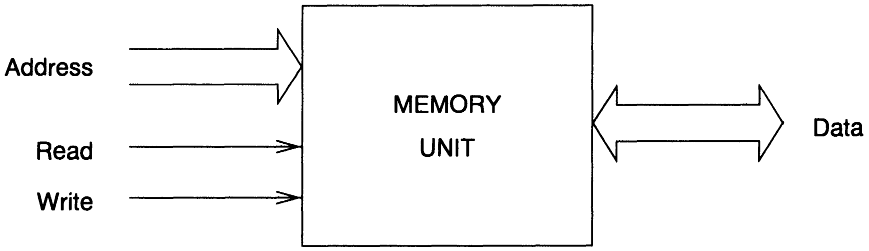


Figure 2.9 Block diagram of the system memory.

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 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 CPU used in the system. The actual memory in a system, however, is always less than or equal to the memory address space. The amount of memory in a system is determined by how much of this memory address space is *populated* with memory chips.

2.3.1 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 of the system bus, respectively. The read and write signals come from the control bus.

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

The read operation is nondestructive in the sense that one can read a location of the memory as many times as one wishes without destroying the contents of that location. The write operation, 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
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. 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 CPU and the CPU 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, 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 CPU rate, wait cycles can be introduced to extend the write cycle to give more time to the memory unit.

2.3.2 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 *read-only memory* and *read/write memory*.

A basic property of memory systems is, 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 in order 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. This memory cannot be written into by the CPU. 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.

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 EPROM can be erased by exposing them to ultraviolet light for 10–20 minutes. Once erased, EPROM can be reprogrammed again.

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 CPU 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

Table 2.1 A comparison of different memory types

Type of memory	Typical access time	Number of write cycles allowed	Volatility (power required)
ROM	50–100 ns	Once[a]	No
PROM	50–100 ns	Once	No
EPROM	50–100 ns	Many	No
SRAM	10–20 ns	Infinite	Full
DRAM	50–100 ns	Infinite	10%

[a] at the time of manufacture

refreshed to replace the charges representing 1 bit. A typical refresh period is about 4 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 representing 1 bit. Thus, DRAM is a destructive read memory.

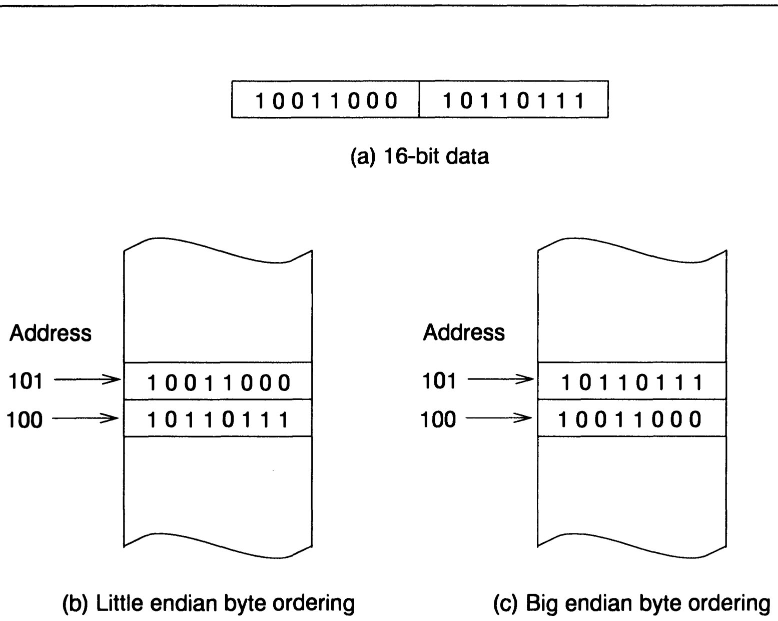
For proper operation, a read cycle is followed by a restore cycle. As a result, the DRAM cycle time, the actual time necessary between accesses, is typically about twice the read access time, which is the time necessary to retrieve a datum from the memory. Table 2.1 gives a summary.

SRAM is faster than dynamic memory but it is more expensive. Typical access time for SRAM is in the 10–20 ns range, whereas that for DRAM is in the 50–100 ns range. However, DRAM costs about $30–100 per MB, whereas the corresponding figure for SRAM is $200–400.

2.3.3 Storing Multibyte Data

Storing data often requires more than 8 bits, or a byte. For example, we need two bytes of memory to store the value of a variable that can take a number in the range 0 through 65,535. Let us assume that the value to be stored is 39,095. Its binary equivalent is shown in Figure 2.10*a*.

How can this 2-byte data be stored in memory at locations 100 and 101? Figure 2.10 shows two possibilities: least significant byte (Figure 2.10*b*) or most significant byte (Figure 2.10*c*) is stored at location 100. These two byte ordering schemes are referred to as *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).



(a) 16-bit data

(b) Little endian byte ordering

(c) Big endian byte ordering

Figure 2.10 Two byte ordering schemes.

Is one byte ordering scheme better than the other? Not really! It is largely a matter of choice for the designers. The Intel 80X86 processors use the little endian scheme, while the Motorola 680X00 processors use the big 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 byte ordering schemes. In this case, conversion from one scheme to the other is required. Pentium provides two instructions to facilitate such conversion: `xchg` can be used for 16-bit data conversion between little and big endian schemes, and `bswap` for 32-bit data. Chapter 3 discusses these instructions in detail.

2.4 Pentium Memory Architecture

Pentium supports a sophisticated memory architecture. In this section we discuss the architectural features provided under real and protected modes. The real mode, which uses 16-bit addresses, is provided to run programs written for 8086. In this mode, Pentium supports the segmented memory architecture.

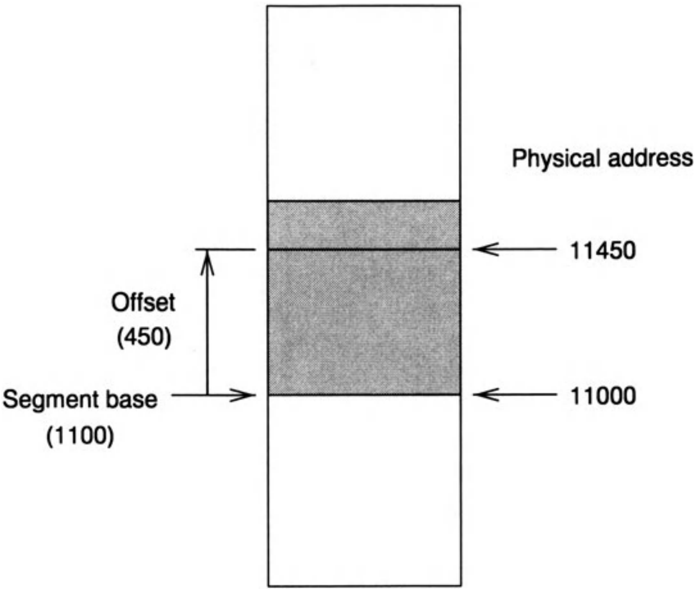


Figure 2.11 Relationship between logical address and physical address of memory (all numbers are in hex).

The protected mode, which is the native mode of Pentium, supports both segmentation and paging. Paging is useful in implementing virtual memory. One requires background in the operating system area to understand the concept of virtual memory. Furthermore, paging is transparent to the application program but segmentation is not. Therefore, we will not discuss the paging features of Pentium. The rest of the section describes the segmented memory architecture in real and protected modes.

2.4.1 Real Mode Memory Architecture

As mentioned, Pentium behaves like a faster 8086 in real mode. The memory address space of the 8086 CPU is 1 MB as its address bus width is 20. To address a memory location, which stores a byte of data, we need a 20-bit address. The address of the first location is 00000H; the last addressable memory location is at FFFFFH. Recall that numbers expressed in the hexadecimal number system are indicated by suffix H (see Appendix A).

Since all registers in the 8086 CPU are only 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. These are the *segment base* and an *offset* within the specified segment (see Figure 2.11). This two-component specification is referred to as the *logical address*. The segment base specifies the beginning address of a segment in memory and the offset specifies the address relative to the beginning of the segment. The offset is also referred to as the *effective address*. The relationship between the logical and physical addresses is shown in Figure 2.11.

The mechanism as described here will not completely solve the problem of addressing a memory address space that requires 20 bit addresses by using 16-bit registers.

Notice from Figure 2.11 that the segment beginning 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 the segments 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, which consists of specifying the segment it is located in and the offset within the segment. 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 2.11. The latter value to identify a memory location is referred to as the *physical memory address*.

As a programmer, you need to worry about logical addresses only. However, when the CPU accesses the memory, it has to supply the 20-bit physical memory address. The conversion of logical address to physical address is straightforward. This translation process, shown in Figure 2.12, involves adding four least significant zero bits to the segment base value and then adding the offset value. When using the hexadecimal number system, simply add a zero digit to the segment base address at the right and add the offset value. As an example, consider the logical address 1100:450H. The physical address is:

```
  11000   (add 0 to 16-bit segment base value)
+   450   (offset value)
  -----
  11450   (physical address)
```

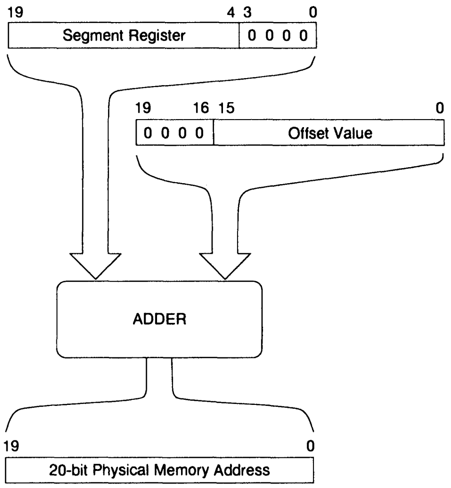


Figure 2.12 Physical address generation in 8086.

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 2.13, where logical addresses 1000:20A9H and 1200:A9H refer to the same physical address 120A9H. The location 120A9H is mapped to two segments.

In our discussion of segments, we never said anything about the actual size of a segment. The main factor limiting the size of a segment is the 16-bit offset value, which restricts the segments to at most 64 K bytes in size. In the real mode, Pentium sets the size of each segment to exactly 64 K bytes.

Programmers view the memory address space as a group of segments. These segments are defined by the programmer. At any instance, a program can access up to six segments. (The 8086 actually supports only four segments—segment registers FS and GS are not present in the 8086 processor.) Typically two of these segments contain code (program's instructions) and data (program's data). The third segment is used for the stack.

If necessary, other segments may be used, for example, to store data, as shown in Figure 2.14. 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.

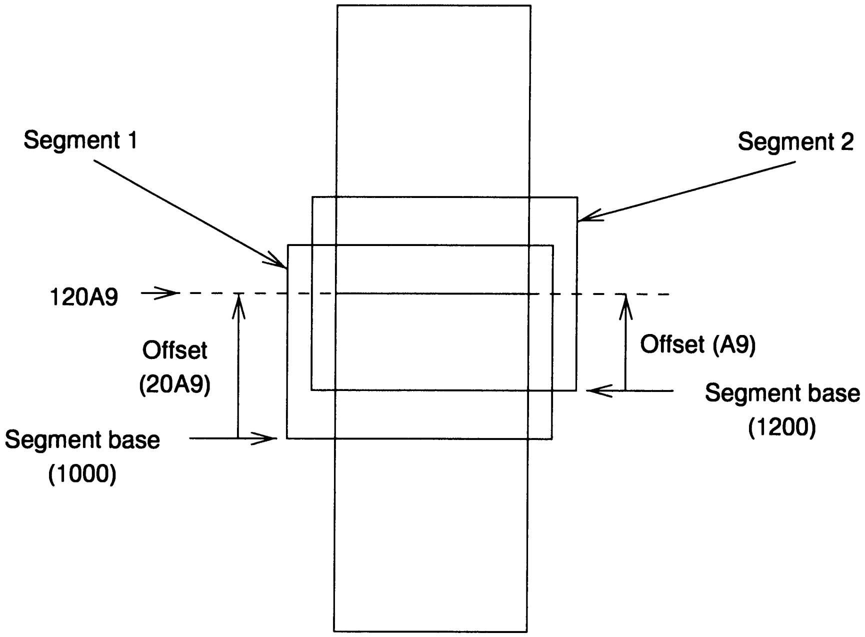


Figure 2.13 Two logical addresses map to the same physical address (all numbers are in hex).

The six segment registers of Pentium point to the six segments, as shown in Figure 2.14. There are no restrictions on the segments except that segments must begin on 16-byte memory boundaries, as described earlier. Except for this restriction, segments can be placed anywhere in the memory. The segment registers are independent and segments can be contiguous, disjoint, partially overlapped, or fully overlapped, as shown in Figure 2.15.

Even though programmers view memory as a group of segments and use the logical address to specify a memory location, all interactions between the CPU and the memory unit must use the physical address. We have seen the process involved in translating a given logical address to the corresponding physical address (see page 39). Pentium has dedicated hardware to perform the address translation, as illustrated in Figure 2.12.

Here is a summary of the real mode memory architecture:

- Segments are exactly 64 K bytes in size.
- A segment register contains a pointer to the base of the segment.
- Default operand size and effective addresses are 16 bits long.

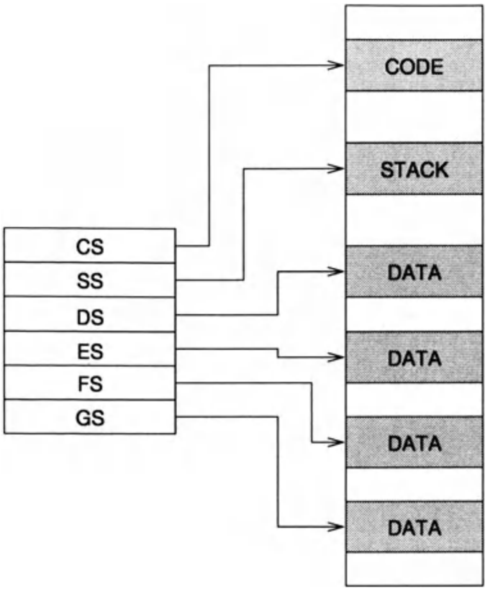


Figure 2.14 The six segments of the memory system.

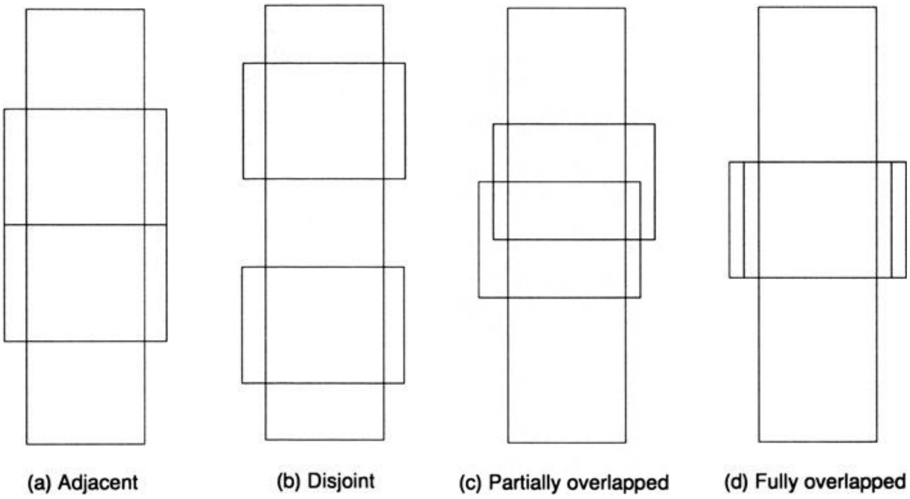


Figure 2.15 Various ways of placing segments in the memory.

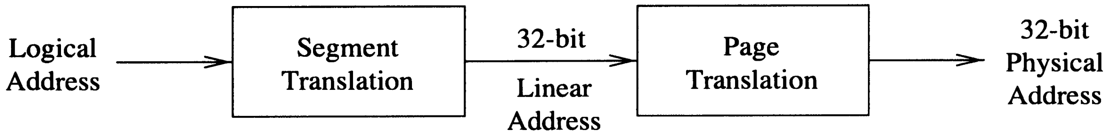


Figure 2.16 Logical to physical address translation process in the protected mode.

- Stack operations use the 16-bit SP register.
- Stack size is limited to 64 KB.
- Paging is not available. Thus, the processor uses the linear address as the physical address (see Figure 2.16).

Keep in mind that the above list is the default attributes. It is, however, possible to change some of these defaults. Section 2.4.7 discusses how 32-bit operands and addresses can be used in the real mode.

2.4.2 Protected Mode Memory Architecture

In protected mode, Pentium supports a more sophisticated segmentation mechanism in addition to paging. This section focuses on the segmentation features of the memory architecture.

As described in the previous section, application programs use the logical addresses, which consists of two components: a segment base, and an offset. Recall that the offset is also called the effective address. 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 2.16. If no paging mechanism is used, the linear address is used as the physical address. It is the physical address that is passed on to the memory to identify the location of access in memory. In the remainder of this section, we focus on the segment translation process only.

The protected mode segment translation process is different from that used in the real mode. In the real mode, which mimics the 8086 mode of operation, the physical address is 20 bits long. The physical address is obtained directly from the contents of the selected segment register and the offset, as illustrated on page 39. In protected mode, the contents of the segment register are taken as an index into a segment descriptor table to get a descriptor. The segment translation process is shown in Figure 2.17. A segment descriptor provides the 32-bit base address of the segment, its size, and access rights, as shown in Figure 2.19. To translate a logical address to the corresponding linear address,

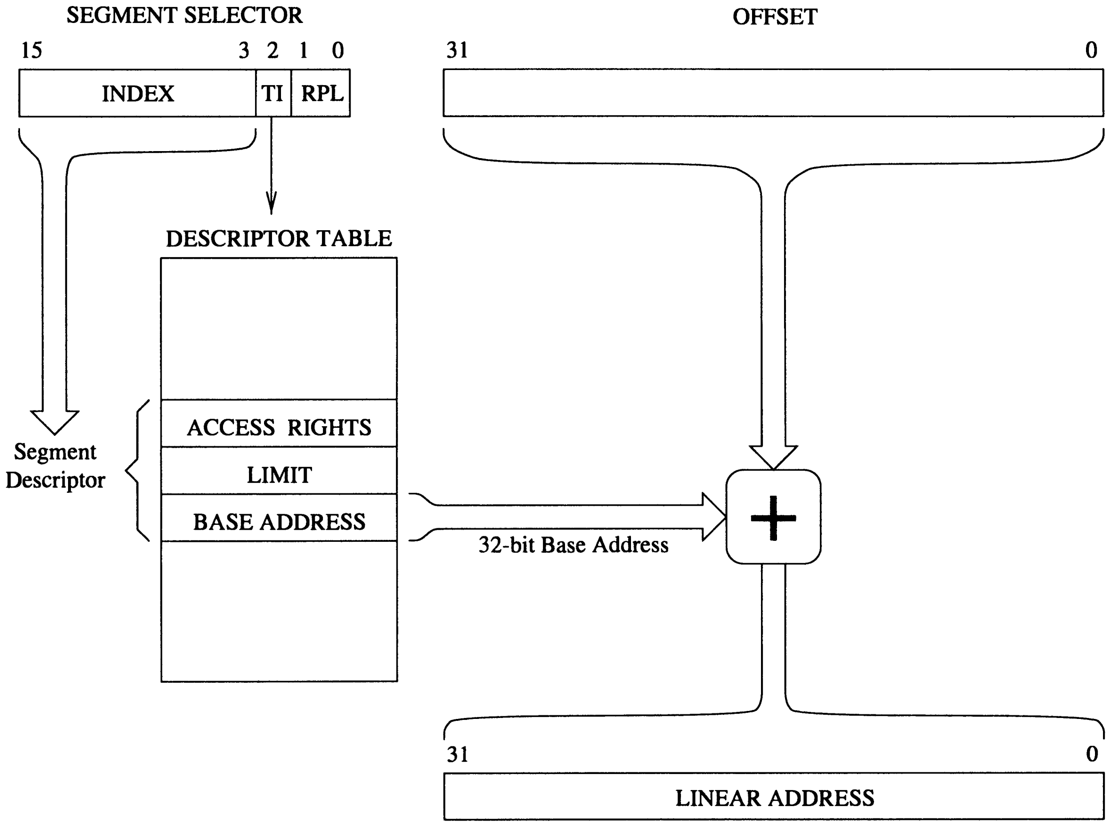


Figure 2.17 Protected mode address translation.

the offset is added to the 32-bit base address. The offset value can be either a 16-bit or a 32-bit number.

2.4.3 Segment Registers

Every segment register has a "visible" part and an "invisible" part, as shown in Figure 2.18. 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. The invisible part of the segment registers is automatically loaded by the processor from a descriptor table (described next).

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 2.18 Visible and invisible parts of segment registers.

The segment selector provides three pieces of information:

- **Index:** 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 2.19, is 8 bytes long, the processor multiplies the index by 8 and adds the result to the base address of the selected descriptor table. The base address of the two descriptor tables is contained in registers GDTR and LDTR for local and global descriptor tables, respectively.
- **Table Indicator (TI):** This bit indicates whether the local or global descriptor table should be used.

  0 = Global descriptor table
  1 = Local descriptor table

- **Requester Privilege level (RPL):** This field identifies the privilege level that is useful in providing protected access to data. The smaller the value of RPL, the higher the privilege level.

2.4.4 Segment Descriptors

A segment descriptor provides the attributes of a segment. These attributes include a 32-bit base address, a 20-bit segment size, as well as control and status information, as shown in Figure 2.19. As an application programmer, you are not concerned with the creation of segment descriptors. Typically, system software such as compilers, linkers, and loaders, creates segment descriptors.

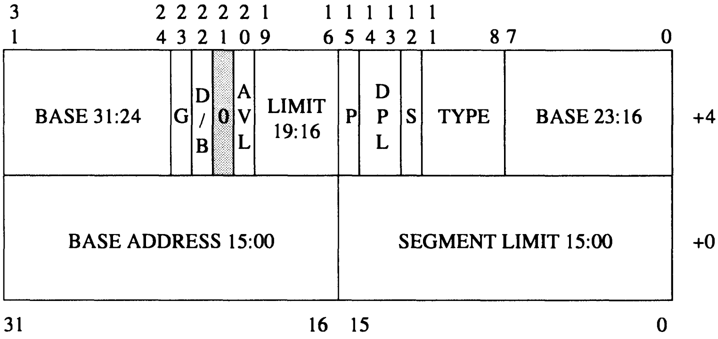


Figure 2.19 Segment descriptor.

Here we provide a brief description of some of the fields shown in Figure 2.19.

- **Base Address:** The 32-bit base 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 form the linear address (see Figure 2.17).
- **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 given to this value:

  1. If the granularity bit is zero, the 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, the 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 a 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 is set to 1.

  In a data segment, this bit is called B bit and controls the size of the stack pointer and size of the stack. If the B bit is 0, stack operations use the SP register and the upper bound for the stack is FFFFH. If the B bit is 1, the

ESP register is used for the stack operations with a stack upper bound of FFFFFFFFH.

Typically, this bit is cleared for the real mode operation and set for the protected mode operation. Section 2.4.7 describes how 16-bit 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 or an application segment. If the bit is 0, the segment is identified as a system segment; otherwise, as an application (code or data) segment.
- **Descriptor Privilege Level (DPL):** This field defines the privilege level of the segment. It is useful in controlling access to the segment using the protection mechanisms of the Pentium processor.
- **Type:** This field identifies the type of segments. The actual interpretations of this field depend 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, a type can identify it as a read-only, read-write, and so on. For a code segment, type identifies it as an execute-only, execute/read-only, and so on.
- **P bit:** This bit indicates whether the segment is present or not. If this bit is 0, the processor generates the segment-not-present exception when a selector for the descriptor is loaded into a segment register.

2.4.5 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors shown in Figure 2.19. 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. They can contain up to 8 K 8-bit descriptors. As shown in Figure 2.17, the upper 13 bits of a segment selector are used as an index into the 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 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 segment cannot be accessed by a program unless there is a descriptor for the segment in either the current LDT or GDT.

The two associated registers, LDTR and GDTR, can be loaded using `lldt` and `lgdt` instructions. Similarly, the LDTR and GDTR register values can be stored by `sldt` and `sgdt` instructions. These instructions are used by the operating system. The interrupt descriptor table is used in interrupt processing and is discussed in Chapter 12.

2.4.6 Segmentation Models

Remember that the 8086 segments are limited to 64 KB, as the offsets used are 16 bits in length. However, in Pentium, it is possible to span a segment over the entire physical address space of 4 GB. 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 *flat model* and is useful for programming environments like UNIX.

Another model that uses the capabilities segmentation to the full extent is the *multisegment model.* Figure 2.20 shows how the current six segments are mapped. A program, in fact, can have more than just six segments. In this case, the segment descriptor table associated with the program will have descriptors loaded for all the segments defined by the program. However, at any time, only six of these segments can be active. The active segments are those that have their segment selectors loaded into the six segment registers. A segment that is not active can be made active by loading its selector into one of the segment registers, and the processor automatically loads the associated descriptor (i.e., the "invisible part" shown in Figure 2.18). The processor generates a general-protection exception if an attempt is made to access memory beyond the segment limit.

2.4.7 Mixed Mode Operation

Our previous discussion of the real and protected modes of operation suggests that we use either 16-bit or 32-bit operands and addresses. The D/B bit indicates the default size. The questions is, is it possible to mix these two? For instance, can we use 32-bit registers in a 16-bit mode of operation? The answer is yes! 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 5.

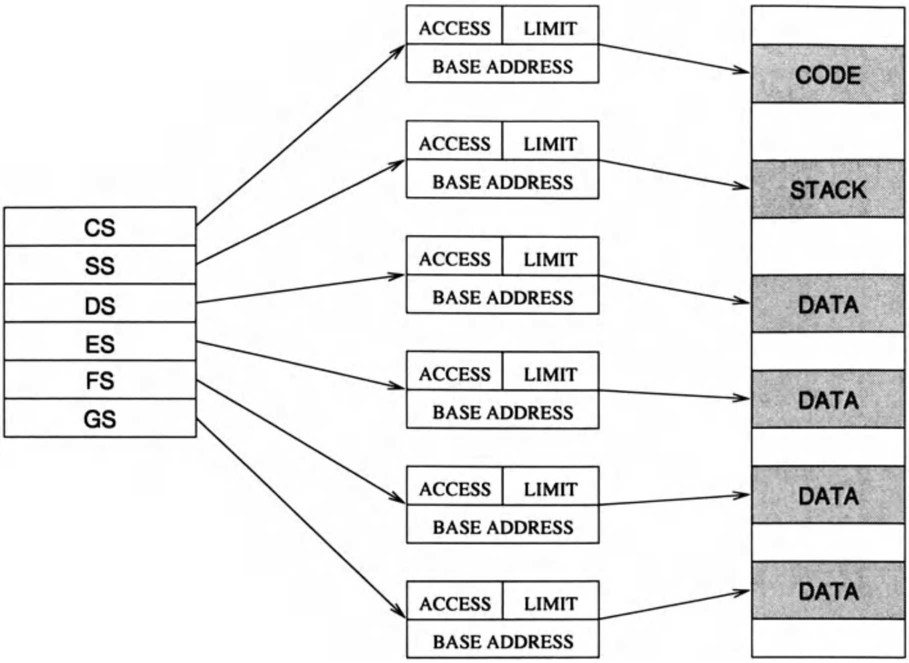


Figure 2.20 Segments in a multisegment model.

2.4.8 Which Segment Register to Use?

This discussion applies to both real and protected modes of operation. In generating a physical memory address, the processor 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 a program instruction, the CS register is used to provide the segment part of the logical address of the instruction to be fetched. 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 always 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, 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 will be said about the stack in Chapter 4.

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 start address. The offset value comes from a variety of sources depending on the addressing mode used. Addressing modes are discussed in Chapter 5.

2.5 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 a purely input device (e.g., keyboard, mouse), a purely 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. Chapter 12 provides 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 are connected directly, the CPU would have to understand and respond appropriately to each I/O device. This would cause the CPU to spend a lot of time interacting with I/O devices and spend less time executing user programs. If we use an I/O controller, this controller could provide the necessary low-level commands and data for proper operation of the associated I/O device. Often, for complex I/O devices such as disk drives, there are special I/O controller chips available.

The second reason for using an I/O controller is that the amount of electrical power used to send signals on the system bus is very low. This means that the cable connecting the I/O device has to be very short (a few centimeters at most). I/O controllers typically contain driver hardware to send current over long cables that connect 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 2.21. When

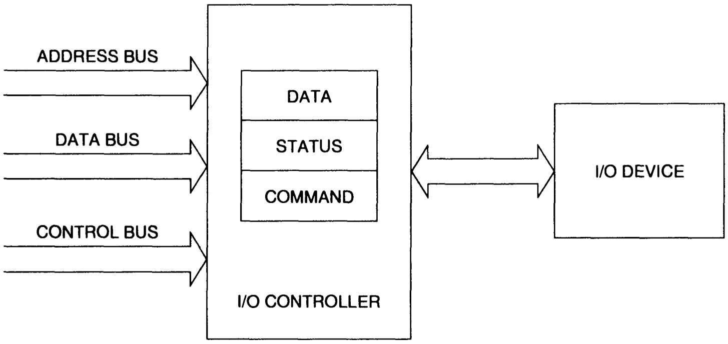


Figure 2.21 Block diagram of a generic I/O device interface.

the CPU wants to interact with an I/O device, it communicates only with the associated I/O controller.

To focus our discussion, let us consider printing a character on the printer. Before the CPU sends a character to be printed, it has to first check the status register of the associated I/O controller to see whether the printer is online/offline, busy or idle, out of paper, and so on. In the status register, three bits can be used to provide this information. For example, bit 4 can be used to indicate whether the printer is online (1) or offline (0), bit 7 can be used for busy (1) or not busy (0) status indication, and bit 5 can be used for out of paper (1) or not (0).

The data register holds the character to be printed and the command register tells the controller the operation requested by the CPU (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 CPU accesses the internal registers of an I/O controller through what are called *I/O ports*. An I/O port is simply the address of a register associated with an I/O controller.

There are two ways of mapping I/O ports. Some CPUs, for example the Motorola 68000, 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 CPUs, like the Intel 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. See Chapter 12 for 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. As I/O instructions do not go through segmentation and paging units, the I/O address space refers to the physical address rather than the linear address.

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

2.5.1 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 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 their own routines to access I/O devices (called *device drivers*). In addition, system resources could be abused either intentionally or accidentally. 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. Typically, 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 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 12.

2.6 Performance: Effect of Data Alignment

Running 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 used 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 1. One of the factors influencing the sort time is the clock cycles required to fetch data. For example, to fetch a 32-bit data item in an 8086-based system with a data bus only 16 bits wide requires two bus cycles. However, in a Pentium-based system, which uses a 64-bit wide data bus, a 32-bit data item can be fetched in a single bus cycle if the data is properly aligned. In the following we explain the concept of data alignment using a 16-bit data item accessed on a 16-bit data bus (e.g., in 8086 mode). You can easily generalize this discussion to Pentium's data bus.

A 16-bit data item is said to be aligned (i.e., word-aligned) if it is located at an even address. For example, a data word located at memory address 120 is aligned because it is located in two contiguous bytes starting at address 120. On the other hand, a data word located at memory address 135 is unaligned. A data word that is located at an even address can be fetched in one bus cycle. If the data word is located at an odd-numbered address, the processor requires two bus cycles to access the 16-bit data—one byte per bus cycle.

The reason for this peculiar behavior is simple to understand. Since the memory is byte-addressable, we supply only one address even when accessing a multibyte data object—such as the `int` data type in C that requires 16 bits of storage. The 16-bit data bus that interconnects the CPU and the memory always supplies the byte that located at an even-numbered address location in the memory on the lower half of the data bus and the byte at the next location on the upper half of the data bus, as shown in Figure 2.22. Thus, 16-bit data that starts at an even address (i.e., word-aligned) can be obtained in one bus cycle. For example, accessing a 16-bit data stored at memory locations 120 and 121 requires only a single bus cycle with the byte at address 120 placed on the lower half of the data bus and the byte at 121 on the upper half of the data bus.

On the other hand, if the 16-bit data is located at addresses 121 and 122, the processor fetches the 16 bits located at addresses 120 and 121 during the first bus cycle, and fetches the 16 bits at addresses 122 and 123 during the next bus cycle. The processor internally discards the unwanted bytes. Therefore, to maximize performance, 16-bit data should be word-aligned (i.e., stored at even addresses).

This discussion can be extended to cover other data items. To avoid a performance penalty, the data should be aligned.

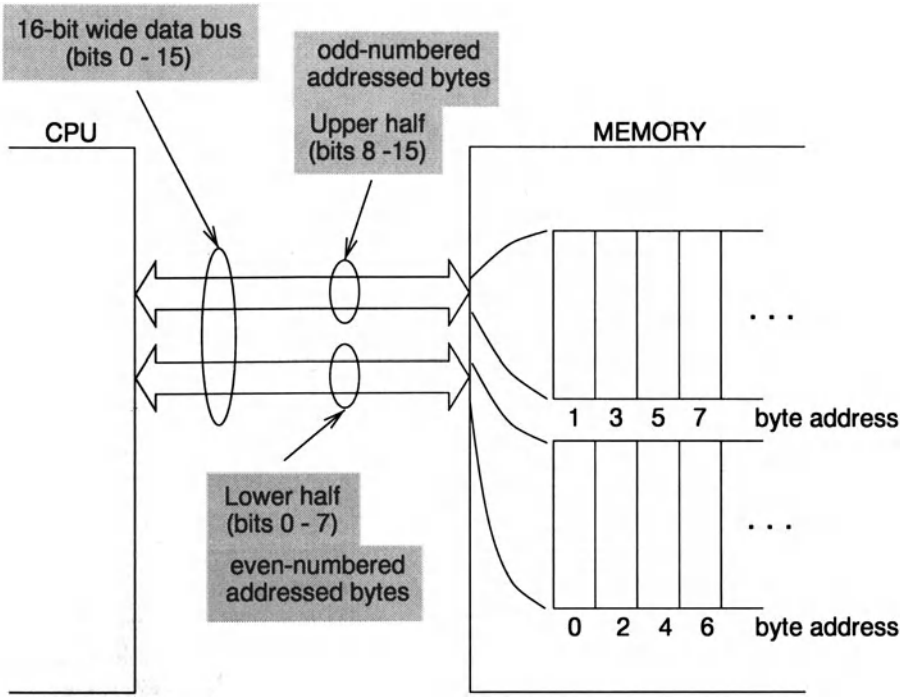


Figure 2.22 Byte-addressable memory interface to the 16-bit data bus.

- *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 2). 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 4. This implies that the least significant two bits of the address must be 0.
- *8-byte data*: A 64-bit data item is aligned if it is stored at an address that is a multiple of 8. This means that the least significant three bits of the address must be 0. This alignment is important for Pentium processors, as they have a 64-bit wide data bus. On 80486 processors, since their data bus is 32 bits wide, a 64-bit data item is in two bus cycles and alignment at 4-byte boundaries is sufficient.

Figure 2.23 shows the impact of word alignment on the sort time. When the array is not word-aligned, the sort time increases by about 16 percent. For example, to sort an 8,000 element array, it takes about four seconds more

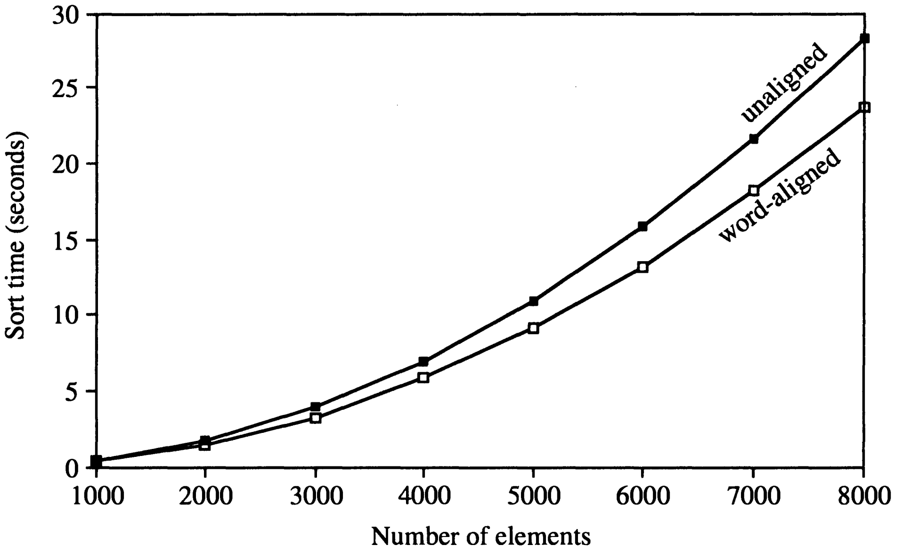


Figure 2.23 Impact of word alignment on the performance of the bubble sort algorithm.

if the array starts at an odd address (i.e., not word-aligned). Except for the performance penalty, word alignment is totally transparent to software.

The Intel 80X86 family of processors allow aligned and unaligned data items. Of course, unaligned data causes performance problems. Alignment constraints of this type are referred to as *soft alignment* constraints. Because of the performance penalty associated with unaligned data, some processors, such as Motorola 68000 and Intel i860, do not allow unaligned data. These alignment constraints are referred to as ***hard alignment*** constraints.

2.7 Summary

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

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.

We described the architecture of Pentium processors from a programmer's point of view. This knowledge is necessary, as the assembly language explicitly refers to the internal registers, and so on.

Pentium can address up to 4 GB of memory. We discussed the memory architecture of real and protected modes. In real mode, Pentium supports 16-bit addresses and the memory architecture of the 8086 processor. The protected mode is the native state of the Pentium processor. In this mode, Pentium supports both paging and segmentation. Paging is useful in implementing virtual memory and is not considered here, as it is beyond the scope of this book. We discussed the segmented memory architecture in detail, as these details are necessary to program in the assembly language.

We briefly discussed how I/O devices are interfaced to the system. More details on this topic are provided in Chapter 12.

We also considered the impact of data alignment on the run time of application programs. By using the bubble sort program discussed in Chapter 1, we demonstrated the influence of data alignment on the sort time.

2.8 Exercises

2–1 What is the execution cycle?

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

2–3 What is the purpose of providing various registers in a CPU?

2–4 What are the three address spaces supported by Pentium?

2–5 What is a segment? Why does Pentium support segmented memory architecture?

2–6 Why is segment size limited to 64 KB in size in the real mode?

2–7 What is the maximum size of a segment in the protected mode?

2–8 We stated that Pentium can access up to six segments at a time. What is the hardware reason for this limitation?

2–9 Describe the logical to physical address translation process in the real mode.

2–10 Describe the logical to linear address translation process in the protected mode.

2–11 Discuss the differences between the segmentation architectures supported in the real and protected modes.

2–12 If a system uses a 166 MHz clock, what is the clock period?

2–13 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.

2–14 What are the differences between ROM and RAM?

2–15 Compare and contrast DRAM and SRAM.

2–16 Convert the following logical addresses to physical addresses. All numbers are in hexadecimal. Assume the real address mode.

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

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

2–18 How many memory read cycles are required by the 8086 processor to read a word (i.e., 16 bits) of data located at the following logical addresses (all numbers are in hex):

(a) 1234:5678	(c) 9128:101
(b) 1ABC:755	(d) 38B0:268

2–19 Repeat the above exercise for a double word (i.e., 32 bits) and byte data.

2.9 Progamming Exercises

2–P1 Write a program in your favorite high-level language to perform logical address to physical address translation in real mode. 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.

In Chapter 3, you will be asked to repeat the exercise in assembly language. The purpose is to compare the time required to write programs in assembly and high-level languages. Therefore, while working on this exercise, you should record the amount of time you spend. Make sure to include the debugging time as well in the comparison.

2–P2 Modify the bubble sort program (C version) given in Chapter 1 to sort an array of characters. Compare the sort times to sort character and integer arrays. After you have become proficient in assembly language, come back to this exercise and give a rational explanation for any difference between the two.

Chapter 3

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 are defined in assembly language
- To demonstrate the performance benefits of translation instruction

The objective of this chapter is to review the basics of the Pentium assembly language. Assembly language statements can either instruct the CPU to perform a task, or direct the assembler during the assembly process. The latter statements are called assembler directives. Section 3.1 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 3.2. The instructions of the CPU consist of an operation code to indicate the type of operation to be performed, and the specification of the data required (also called addressing mode) by the operation. Section 3.3 describes some basic addressing modes supported by Pentium.

The instruction set of Pentium can be divided into several groups of instructions. Section 3.4 discusses the instructions that transfer data, including

S. P. Dandamudi, *Introduction to Assembly Language Programming*

`mov`, `xchg`, *and* `xlat` *instructions. Section 3.5 provides an overview of some of the Pentium instructions belonging to the other groups. Later chapters discuss these instructions in more detail.*

Section 3.6 describes the assembler directives to define constants—numeric as well as string constants. Several examples are provided in Section 3.7. The performance advantage of the translation instruction `xlat` *is demonstrated in Section 3.8. The chapter concludes with a summary.*

3.1 Assembly Language Statements

Assembly language programs are created out of three different classes of statements. Statements in the first class tell the CPU what to do. These instructions are called *executable instructions*, or *instructions* for short. Each executable instruction consists of an *operation code* (*op-code* 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 provide information to the assembler on various aspects of the assembly process. These instructions are called *assembler directives* or *pseudo-ops*. Assembler directives are non-executable 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 will use macros to provide the basic input and output capabilities to stand-alone assembly language programs. Macros are discussed in detail in Chapter 10.

Assembly language statements are entered one per line in the source file. Even though up to 128 characters can be used in a line, it is a good practice to limit a line to 80 characters so that it can be displayed on the screen. Except for a few statements, most assembly language statements require far fewer characters than 80.

All three classes of the assembly language statements use the same format:

```
[label]     mnemonic     [operands]     [;com-
ment]
```

The fields in the square brackets are optional in some statements. As a result of this format, it is a common practice to align the fields to aid readability of assembly language programs. The assembler does not care about spaces between the fields.

Assembly language statements require more characters per line only because of the comments we add to the code lines. Long comments can always be broken into multiple lines. Blank lines, comment lines (lines consisting entirely of comments), and label lines (lines just containing labels) are acceptable and should be judiciously used to structure the program in order to improve its readability and maintainability.

Label: This is an optional field. The label field serves two distinct purposes: it's used to represent either an identifier or a constant. When a label appears in an executable instruction, it is used as a marker to identify the instruction. Then, for example, you can make program execution jump to the labeled instruction. In this case, label represents the memory address of the instruction. When used with certain assembler directives like EQU, label represents a constant.

Mnemonic: This is a required field and identifies the purpose of the statement. In certain statements, this field is not required. Examples include lines consisting of a comment, or a label, or a label and a comment.

Operands: Operands specify the data to be manipulated by the statement. The number of operands required depends on the specific statement or directive. For instance, executable statements may have zero, one, two, or three operands.

Comment: This is an optional field and serves the same purpose as that in a high-level language. Comments play a more important role in assembly language, as it is a low-level language. Assembler ignores all comments. Comments begin with a semicolon (;) and extend until the end of the line. Since the readability of assembly language programs is poor, comments should be generously added to improve readability. While some authors suggest adding comments to every line of code, it is good programming practice to explain the functionality of a group of statements by several lines of comments and then add brief comments to selected code lines within the group. This is the practice followed in this book.

Now let us look at some sample assembly language statements.

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

The label `repeat` can be used to refer to this particular statement. The mnemonic `inc` indicates increment operation to be done on data stored in memory at a location referred to by the variable name `result`. The comment simply explains what the instruction is doing. Adding such self-explanatory comments is redundant and we will avoid commenting each line with such trivial comments. The following assembler directive defines a constant CR. The ASCII carriage return value is assigned to it by the EQU directive.

```
CR     EQU     ODH        ;carriage return character
```

In the previous two examples, label field has two different forms. The label in the executable instruction is followed by a colon (:) but not in the directive statement.

A label and other names can be formed from upper and lowercase letters (a–z, A–Z), digits (0 through 9), and special characters (_, %, ?, $, ., @).

A name may not begin with a digit and if a period is used, it must be the first character. For example, jump2 and repeat are valid but not go.back and 2_jump. Other characters may be used in any position. Among the special characters, the underscore character is frequently used to aid readability. (Underscores also play a special role in interfacing with C language—discussed in Chapter 13.)

A name can have many characters but only the first 31 characters are significant. Certain reserved words that have special meaning to the assembler are not allowed as names. These include mnemonics such as inc and EQU.

The assembler is normally case insensitive. For example, labels repeat and REPEAT are treated the same. The assembler can be made case sensitive by using an option (e.g., /ml option with TASM). We follow the convention that the source code is normally in lowercase except for directive mnemonics and constants defined in the program.

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 additional spaces/tabs are ignored by the assembler.

It is good programming practice to use blank lines and spaces to improve the readability of assembly language programs. As a result, you will rarely see in this book a statement containing all four fields in a single line. In particular, we will almost always write labels on a separate line unless doing so destroys the program structure. Thus, our first example statement would be written as two statements, as

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

3.2 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;        /* 1 byte is allocated   */
int     value;           /* 2 bytes are allocated */
float   total;           /* 4 bytes are allocated */
double  average_value;   /* 8 bytes are allocated */
```

These variable declarations not only specify the amount of storage required, but also indicate how the stored bit pattern should be interpreted. As an example, consider the following two statements in C:

```
unsigned   value_1;
int        value_2;
```

Both variables will have two bytes reserved for storage. However, the bit pattern stored in them would be interpreted differently. For instance, the bit pattern (8DB9H)

```
1000 1101 1011 1001
```

stored in the two bytes allocated for value_1 is interpreted as representing 36,281, while the same bit pattern stored in value_2 would be interpreted as −29,255.

In assembly language, allocation of storage space is done by the define assembler directive. The define directive can be used to reserve and initialize one or more bytes. However, no interpretation (as in C variable declarations) 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 a storage allocation statement is

```
[variable-name] define-directive initial-value [,initial-value],...
```

The square brackets indicate optional items. The variable-name is used to identify the storage space allocated. The assembler associates an offset value for each variable name defined in the data segment. Note that no colon (:) follows the variable name (unlike a label identifying an executable statement).

The define directive takes one of the five basic forms:

```
DB   Define Byte        ;allocates 1 byte
DW   Define Word        ;allocates 2 bytes
DD   Define Doubleword  ;allocates 4 bytes
DQ   Define Quadword    ;allocates 8 bytes
DT   Define Ten Bytes   ;allocates 10 bytes
```

Let us look at some examples now.

```
sorted    DB    ’y’
```

This statement allocates a single byte of storage and initializes to character y. Your assembly language program can refer to this data location by its name sorted. If you just want to reserve storage space without initialization, you can write

```
sorted    DB    ?
```

You can also use numbers to initialize. For example,

```
sorted    DB    79H
```

or

```
sorted    DB    1111001B
```

is equivalent to

```
sorted    DB    ’y’
```

Note that the ASCII value for y is 79H. The following data definition statement allocates two bytes of contiguous storage and initializes to 25159.

```
value    DW    25159
```

The decimal value 25159 is automatically converted to its 16-bit binary equivalent (6247H). Since 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. The following table shows the valid range for the numeric operands:

Directive	Valid range
DB	−128 to 255 (i.e., -2^7 to $2^8 - 1$)
DW	−32,768 to 65,535 (i.e., -2^{15} to $2^{16} - 1$)
DD	−2,147,483,648 to 4,294,967,295 (i.e., -2^{31} to $2^{32} - 1$) or a short floating-point number (32 bits)
DQ	-2^{63} to $2^{64} - 1$ or a long floating-point number (64 bits)

Using a constant that is outside the specified range can result either in an assembler error, or in assigning a wrong value. For example, the statement

```
byte1    DB    256
```

causes an assembly time error. In general, the assembler can accept a value in the range −256 to +255. However, 8 bits are not sufficient for the values between −256 and −129. Therefore, the assembler converts the number into 2's complement representation using 16 bits and stores the lower byte. For example,

```
byte2    DB    -200    ; stores 38H
```

stores 38H because the 2's complement representation of −200 is FF38H.

Similarly, the statement

```
word1    DW    -60000  ; stores 15A0H
```

assigns 15A0H because −60000 is outside the range of signed numbers that can be represented using 16 bits. Therefore, as in the last example, −60000 is converted to its 2's complement equivalent using 32 bits (FFFF15A0H), and the lower word is stored.

Short and long floating-point numbers are represented using 32 or 64 bits, respectively. See Appendix A for details. We can use DD and DQ directives to assign real numbers, as shown in the following examples:

```
float1    DD    1.234
real2     DQ    123.456
```

Multiple Definitions

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

```
sorted    DB    ’y’          ; ASCII of y = 79H
value     DW    25159        ; 25159D = 6247H
total     DD    542803535    ; 542803535D = 205A864FH
```

When several data definition statements are used as above, the assembler allocates contiguous memory locations for the variables. The memory layout for the three variables is

$$\begin{array}{ll} \text{address:} & x \quad x{+}1 \quad x{+}2 \quad x{+}3 \quad x{+}4 \quad x{+}5 \quad x{+}6 \\ \text{contents:} & \underbrace{79}_{\texttt{sorted}} \quad \underbrace{47 \qquad 62}_{\texttt{value}} \quad \underbrace{4F \qquad 86 \qquad 5A \qquad 20}_{\texttt{total}} \end{array}$$

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
```

Multiple Initializations

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

```
marks    DW    8 DUP (0)
```

The DUP directive is useful in defining arrays and tables. Here are some examples using the DUP directive.

```
table1   DW  10 DUP (?)       ;10 words,uninitialized
name1    DB  30 DUP ('?')     ;30 bytes,each byte
                              ; initialized to ?
name2    DB  30 DUP (?)       ;30 bytes,uninitialized
message  DB  3  DUP ('Bye!')  ;12 bytes,initialized
                              ; to Bye!Bye!Bye!
```

The DUP directive may also be nested. For example, to allocate storage space containing

```
***??!!!!!***??!!!!!***??!!!!!***??!!!!!
```

we can write

```
stars  DB  4 DUP (3 DUP ('*'), 2 DUP ('?'), 5 DUP ('!'))
```

A two-dimensional 10×5 matrix (10 rows, 5 columns) can be defined as

```
matrix    DW    10 DUP (5 DUP (0))
```

The initialization values of define directives can also be expressions, as shown in the following example.

```
max_marks    DW    7*25
```

This statement is equivalent to

```
max_marks    DW    175
```

The assembler evaluates such expressions at assembly time and assigns the resulting value. Use of expressions to specify initial values is not preferred because it affects the readability of your program. However, there are certain situations where using an expression actually helps clarify the code. In our example, if max_marks is representing 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. Data definitions are further discussed in Chapter 10.

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    DW   10 DUP (?)
message  DB   'The grade is:',0
char1    DB   ?
```

As we have indicated, the assembler assigns contiguous memory space for the variables. Assembler also uses the same ordering of variables that is present in the source code. Then, finding the offset values of a variable is a simple matter of counting the number of bytes allocated to 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 in Table 3.1.

Table 3.1 Symbol table for the example data segment

name	offset
`value`	0
`sum`	2
`marks`	6
`message`	26
`char1`	40

Table 3.2 Correspondence between Turbo C data types and data definition directives

Directive	C data type
DB	`char`
DW	`int, unsigned`
DD	`float, long`
DQ	`double`
DT	Not used to specify a data type but used to store intermediate float values

Correspondence to C Data Types

The correspondence between the data definition directives and the Turbo C data types is shown in Table 3.2. Some examples using DB, DW, and DD directives are shown in Table 3.3.

Two consecutive apostrophes can be used in a string to specify a single apostrophe, as in

```
message     DB     'John''s'
```

to reserve 6 bytes of storage and initialize it to `John's`. TASM and MASM also allow the use of double quotation marks to specify a string of characters, as in

```
message     DB     ''John's''
```

In a string that is delineated by double quotation marks, two consecutive double quotation marks can be used to stand for a single one. Since double quotation marks are used to specify strings in C (and is different from the

Table 3.3 Some example data definition declarations

C declaration		Assembly language data definition		
char	ch_1;	ch_1	DB	?
char	string1[30];	string1	DB	30 DUP (?)
char	name1[25] = ‘‘John’’;	name1	DB	’John’,0,20 DUP (?)
int	value = 50;	value	DW	50;
int	array[20];	array	DW	20 DUP (?)
long	total = 0;	total	DD	0

sense used here to specify a string of characters) we will exclusively use only apostrophes in this book.

LABEL Directive

The LABEL directive provides another way to name a memory location *without* actually defining any data. The syntax is

```
name     LABEL     type
```

where type specifies the variable type. The standard types BYTE, WORD, DWORD, QWORD, and TBYTE can be used to label 1-, 2-, 4-, 8-, and 10-byte data.

In the example

```
.DATA
count      LABEL     WORD
Lo_count     DB     0
Hi_count     DB     0
.CODE
               .
        mov    Lo_count,AL
        mov    Hi_count,CL
               .
```

the two bytes of memory Lo_count and Hi_count can also be referenced as a 16-bit number count. We can also individually manipulate the lower and upper halves of count.

The LABEL directive is also useful in creating an alias of another data type, as shown in the following example.

```
.DATA
byte_count   LABEL   BYTE
count    DW    0
.CODE
                 .
    mov    byte_count,CL
                 .
```

If the LABEL directive is not used in this example, we have to use the PTR directive (discussed in Section 3.4.2) to rewrite the mov statement as

```
mov    BYTE PTR count,CL
```

3.3 Where Are the Operands?

Assembly language programs can be thought of as consisting of two logical parts: *data* and *code*. Most of the assembly language instructions require specification of the location of the data to be operated on. There are a variety of ways to specify and find where the operands required by an instruction are located. These are called *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 5.

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

- in a register internal to the CPU
- in the instruction itself
- in main memory (usually in the data segment)
- at an I/O port (discussed in Chapter 12)

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. A variety of addressing modes are available to specify the location of an operand residing in memory. The motivation for providing several addressing modes comes from the need to efficiently support high-level language constructs. Chapter 5 discusses this issue in detail.

3.3.1 Register Addressing Mode

In this addressing mode, CPU 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 CPU registers. The syntax of the mov instruction is

```
mov     destination,source
```

The mov instruction copies contents of source to destination. The contents of source, however, are not destroyed as a result. 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, mov is operating on 32-bit data. However, the mov instruction can also be used on 16- and 8-bit data, as shown in the following example:

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

Using the register addressing mode is the most efficient way of specifying data because the data is residing within the CPU and, therefore, no memory access is required.

3.3.2 Immediate Addressing Mode

In this addressing mode, data is specified as part of the instruction. As a result, even though the data is 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 and immediate data is always a constant, either given directly or via the EQU directive (discussed in Section 3.6). 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 that 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* addressing modes.

3.3.3 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 addressing modes previously described.

Recall that to locate a data item in a 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 data is specified. The offset value is sometimes referred to as the *effective address*.

In the direct addressing mode, the offset value is specified directly as part of the instruction. In an assembly language program, the value is usually indicated by the variable name of the data item referenced. The assembler will translate the name into its associated offset value during the assembly process. To facilitate this translation, assembler maintains a symbol table, which stores the offset values of all variable names 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'         ;reserves one byte and
                           ; initializes with y
table1    DW   20 DUP (0)  ;reserves 40 bytes and
                           ; initializes to 0
name1     DB   'Jim Ray'   ;reserves 7 bytes and
                           ; initializes to Jim Ray
```

Here are some examples of the mov instruction:

```
mov    AL,response     ;copies character y into
                       ;  AL register
mov    response,'N'    ;N is written into the
                       ;  byte represented by
                       ;  response (Y is lost)
mov    name1,'K'       ;write K as the first
                       ;  character of name1,
                       ;  which now reads Kim Ray
```

```
mov     table1,56        ; 56 is written in the
                         ; first two bytes of
                         ; table, which contains
                         ; 56 and zeroes for the
                         ; remaining 19 elements
```

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

3.3.4 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`, such as

```
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 BX register and then BX acts as a pointer to an element in `table1`. By manipulating the contents of the BX register, we can access different elements of `table1`. Remember that we use 16-bit segments where the offset into a segment is 16 bits long (see Chapter 2).

How do we get the starting address of `table1`? A statement like

```
mov     BX,table1
```

will not work because this statement copies the first element of `table1` into the BX register. Remember that the symbolic name `table1` refers to the offset of the first element of `table1`. The OFFSET directive should be used whenever the offset (i.e., the effective address) of a variable is needed. Thus,

```
mov     BX,OFFSET table1
```

copies the offset of `table1` into the BX register. The following code assigns 100 to the first element and 99 to the second element of `table1`. Note that BX is incremented by 2 because each element of `table1` requires two bytes.

```
mov     BX,OFFSET table1   ; copy address of table1 to BX
mov     [BX],100           ; table1[0] := 100
add     BX,2               ; BX := BX + 2
mov     [BX],99            ; table1[1] := 99
```

Chapter 5 discusses other memory addressing modes that can perform this task more efficiently. In summary, we have discussed four addressing modes:

addressing mode	valid example	invalid example
register	`mov EAX,EBX`	`mov AX,EBX`
immediate	`mov ECX,155`	`mov 155,ECX`
direct	`mov table1,DX`	`mov response,name1`
indirect	`mov [BX],EAX`	`mov [BX],[AX]`

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     BX,table1
```

can be used in place of the

```
mov     BX,OFFSET table1
```

instruction. The difference is that `lea` computes the offset values at run time, whereas `mov` with `OFFSET` resolves the offset value at assembly time. For this reason, we will try to use the latter whenever possible. However, `lea` offers more flexibility as to the types of `source` operands. For example, we can write

```
lea     BX,array[SI]
```

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

```
mov     BX,OFFSET array[SI]     ; illegal
```

3.4 Data Transfer Instructions

We now discuss some of the data transfer instructions supported by Pentium. Specifically, we describe `mov`, `xchg`, and `xlat` instructions. Other data transfer instructions such as `movsx` and `movzx` are discussed in Chapter 6.

3.4.1 The mov Instruction

We have already introduced the `mov` instruction, which requires two operands and has the syntax

```
mov    destination,source
```

The data is copied from source to destination and the source operand remains unchanged. Both operands should be of the same size. The mov instruction can take one of the following five forms:

```
mov    register,register
```

Restrictions:

- Destination register cannot be CS or (E)IP registers
- Both registers cannot be segment registers

```
mov    register,immediate
```

Restriction: Register cannot be a segment register

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

There is no move instruction to transfer data from memory to memory, as the Pentium processor does not allow it. However, as we will see in Chapter 9, memory to memory data transfer is possible when operating on strings.

Here are some example mov statements:

```
.DATA
response  DB   'Y'
table1    DW   20 DUP (0)
name1     DB   'Jim Ray'

 CODE
      mov    AL,response
      mov    DX,table1
      mov    response,'N'
      mov    name1+4,'K'
```

Some invalid mov statements are

```
mov    DL,CX     ;different operand sizes
mov    DS,175    ;immediate value cannot be moved
                 ; into a segment register
mov    CS,DX     ;destination register cannot be CS
mov    ES,DS     ;both registers cannot be segment
                 ; registers
mov    715,EAX   ;immediate value cannot be
                 ; destination operand
```

3.4.2 Ambiguous Moves: PTR Directive

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

```
mov     BX,OFFSET table1
mov     SI,OFFSET name1
mov     [BX],100
mov     [SI],100
```

it is not clear whether a word (2 bytes) or a byte equivalent of 100 is to be written in the memory. The PTR directive can be used to clarify. WORD PTR can be used to identify a word operation and BYTE PTR for a byte operation. Using the PTR directive, we can write

```
mov     WORD PTR [BX],100
mov     BYTE PTR [SI],100
```

WORD and BYTE are called *type specifiers*. Some of the type specifiers available are

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

3.4.3 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. Thus,

```
xchg    response,name1      ; illegal
```

is invalid.

The xchg instruction is convenient because we do not need a third register to hold a temporary value in order to swap two values. For example, we need three mov instructions

```
mov    ECX,EAX
mov    EAX,EDX
mov    EDX,ECX
```

to perform xchg EAX,EDX. Thus, xchg is the most efficient way to exchange two 8-, 16-, or 32-bit values. This instruction is especially useful in sorting applications. The xchg instruction is also useful in implementing semaphores for process synchronization. It is also useful to swap the two bytes of 16-bit data to perform conversions between little endian and big endian forms, as in the following example:

```
xchg    AL,AH
```

Pentium provides the bswap instruction to perform such conversions on a 32-bit data. The format is

```
bswap  32-bit register
```

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

3.4.4 The xlat Instruction

The xlat (translate) instruction can be used to perform character translation. For example, it can be used to translate character codes from ASCII to EBCDIC and vice versa. The xlat has the form

```
xlatb
```

To use the xlat instruction, the BX register must 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 BX and reads the byte at the resulting address. This byte replaces the index value in the AL register. Since the 8-bit AL register provides the index into the translation table, the number of entries in the table is limited to 256. An application of xlat is given in Example 3.3.

3.5 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.

3.5.1 Simple Arithmetic Instructions

The Pentium family provides several instructions to perform simple arithmetic operations. In this section, we will describe five instructions to perform addition and subtraction. We will defer a full discussion until Chapter 6.

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 of these 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    BX        ; increment 16-bit register
dec    DL        ; decrement 8-bit register
```

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

Consider the following program:

```
.DATA
count    DW    0
value    DB    25

.CODE
      inc     count               ;unambiguous
      dec     value               ;unambiguous
      move    BX,OFFSET count
      inc     [BX]                ;ambiguous
      mov     SI,OFFSET value
      dec     [SI]                ;ambiguous
```

In the above example,

```
inc    count
dec    value
```

are unambiguous because the assembler knows from the definition of count and value that they are WORD and BYTE operands. However,

```
inc    [BX]
dec    [SI]
```

are ambiguous because BX and SI registers merely point to an object in memory but the actual object type (whether a WORD or BYTE) is not clear. We have to resort to the PTR directive to clarify, as shown below:

```
inc    WORD PTR [BX]
dec    BYTE PTR [SI]
```

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)
```

As a result, destination loses its contents before the execution of add but the contents of source remain unchanged. The examples given below assume the following data definitions:

```
.DATA
value    DB    0F0H
count    DW    3746H
```

	Before add		After add
instruction	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 following instructions are invalid:

```
add     AX,BL          ;mismatched operands
add     [SI],[DI]      ;two memory operands
add     value,[BX]     ;two memory operands
```

The instruction

```
add     [BX],10
```

is ambiguous. It should be written as one of the following depending on the operand size:

```
add     BYTE PTR [BX],10      ; for 8-bit operand
add     WORD PTR [BX],10      ; for 16-bit operand
add     DWORD PTR [BX],10     ; for 32-bit operand
```

In general,

```
inc     EAX
```

is preferred to

```
add     EAX,1
```

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

The 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)
```

instruction	Before sub		After sub
	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 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 except that the result of subtraction is not saved. Thus, cmp does not disturb both destination and source 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 used in conjunction with conditional jump instructions for decision making. This is the topic of the next section.

3.5.2 Conditional Execution

The Pentium instruction set contains several branching and looping instructions to construct programs that require conditional execution. In this section, we will discuss a subset of these instructions. A detailed discussion can be found in Chapter 7.

Program execution, by default, proceeds in a sequential manner—execution of instructions is in the order present in the program. However, programs often require conditional and looping constructs.

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 a 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 will 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 when the specified condition is satisfied. The general format is

```
    j<cond>   label
```

where <cond> identifies the condition under which the target instruction at `label` should be executed. Usually, the condition being tested is the result of the last arithmetic/logic operation. For example, the 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 labeled instruction if and only if ZF = 1.

To cause the jump, Pentium loads the (E)IP register with the target instruction address. Recall that the (E)IP register always points to the next instruction to be executed. Therefore, when the character read is CR, instead of fetching the instruction

```
inc     CL
```

it will fetch the

```
mov     DL,AL
```

instruction. Here are some of the conditions tested by conditional jump instructions:

```
je        jump if equal
jg        jump if greater
jl        jump if less
jge       jump if greater or equal
jle       jump if less than or equal
jne       jump if not equal
```

Conditional jumps can also test the values of flags. Some examples are

```
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: Consider the following code. The following table shows the actions taken depending on `statement_1`.

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

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 the next instruction `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`

The 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 7, the six conditional jump instructions introduced here are sufficient for writing simple assembly language programs.

3.5.3 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
repeat:
                .
        <loop body>
                .
        dec     CL
        jnz     repeat      ;jumps back to repeat
                .           ; as long as dec  CL does
                .           ; not result in CL = 0
```

Pentium, however, provides 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 of the jump (`repeat` in the above example).

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

```
        mov    CX,50
repeat:
          . . .
        <loop body>
          . . .
        loop   repeat
          . . .
```

3.5.4 Logical Instructions

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

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

The and and or are binary operators and perform bitwise and and or logical operations. The not is a unary operator that performs bitwise complement operations.

The logical and operation sets the destination bit according to the truth table shown below, depending on the values of the corresponding bits in the source and destination operands.

Truth table for the 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

The logical or operation is analogous to the and operation except that it follows the truth table shown below:

Truth table for the 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

The logical not operation simply flips the bits (a 1 in input becomes a 0 in the output, and vice versa), as shown in the following truth table:

Truth table for the not operation

Input bit	Output bit
destination b_i	destination b_i
0	1
1	0

Here are some examples explaining their operation (all numbers are expressed in binary).

		and AL,BL	or AL,BL	not AL
AL	BL	AL	AL	AL
1010 1110	1111 0000	1010 0000	1111 1110	0101 0001
0110 0011	1001 1100	0000 0000	1111 1111	1001 1100
1100 0110	0000 0011	0000 0010	1100 0111	0011 1001
1111 0000	0000 1111	0000 0000	1111 1111	0000 1111

Logical instructions also set some of the flags and therefore can be used in conjunction with conditional jump instructions to implement decision making in assembly language programs. Until we fully discuss the flags in Chapter 6, the following usage should be sufficient to write and understand 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 also sets the zero flag (i.e., ZF = 1) because the result is zero. Recall that je tests the ZF and jumps to the target location if ZF = 1. In this example, it is more appropriate to use jz (jump if zero). Thus,

```
    jz    bit_is_zero
```

can replace the

```
    je    bit_is_zero
```

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

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 instruction set has a test instruction. The syntax is

```
    test   destination,source
```

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

```
    test   AL,01H
```

instead of

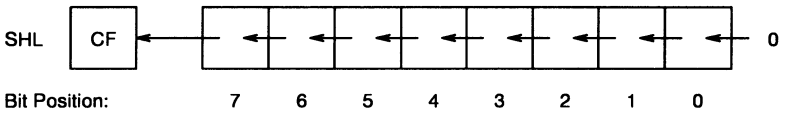
```
    and    AL,01H
```

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

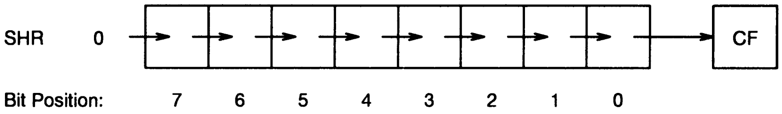
3.5.5 Shift Instructions

The Pentium instruction set provides several shift instructions. We will 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 carry flag (CF), and the vacated rightmost bit is filled with a zero. The bit that was in CF is lost as a result of the left shift operation.



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



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 can be used to specify the number of bit positions to be shifted. `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 `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.

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

3.5.6 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 provide this facility. These instructions can be divided into two types: rotate without involving the carry flag (CF), or through the carry flag. We will briefly discuss these two types of instructions next.

Rotate Without Carry

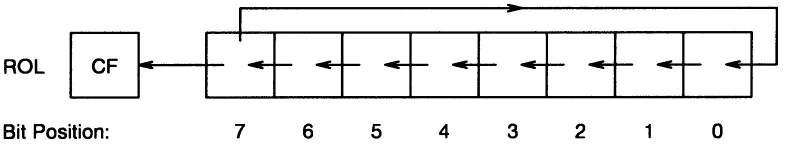
There are two instructions in this group:

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

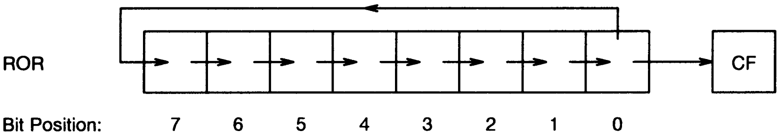
The format of these instructions is similar to the shift instructions and is given below:

```
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 will catch the last bit rotated out of destination. The following examples illustrate the rotate operation.

	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

As a further example, consider encryption of a byte by interchanging the upper and lower nibbles (i.e., 4 bits). This can be done either by

```
ror    AL,4
```

or by

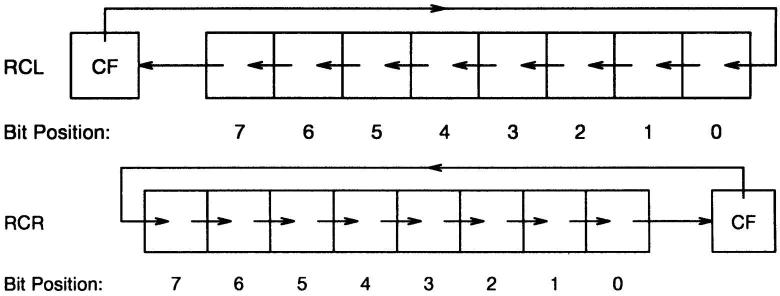
```
rol    AL,4
```

Rotate Through Carry

The instructions

rcl (Rotate through Carry Left)
rcr (Rotate through Carry Right)

include the carry flag in the rotation process. That is, the bit that is rotated out at one end goes into the carry flag and the bit that was in the carry flag is moved into the vacated bit, as shown below.



Some examples of rcl and rcr are given next.

	Before execution		After execution	
Instruction	AL or AX	CF	AL or AX	CF
`rcl AL,1`	1010 1110	0	0101 1100	1
`rcr AL,1`	1010 1110	1	1101 0111	0

`mov CL,3`
`rcl AL,CL`
	0110 1101	1	0110 1101	1

`mov CL,5`
`rcr AX,CL`
	1011 1101 0101 1001	0	1001 0101 1110 1010	1

The rcl and rcr instructions provide flexibility in bit rearranging. Furthermore, these are the only two instructions that take the carry flag bit as an input. This feature is useful in multiword shifts. 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 to the carry flag. The rcr instruction copies this carry flag value into the most significant bit of EAX during the rotation process. We will see in Chapter 8 that Pentium provides two double-precision shift instructions to facilitate shifting of 64-bit numbers.

3.6 Defining Constants

Assemblers provide two directives—EQU and =—to define constants, numeric as well as literal constants. The EQU directive can be used to define numeric constants and strings, whereas the = directive can be used to define numeric constants only.

3.6.1 The EQU Directive

The syntax of the EQU directive is

```
name     EQU    expression
```

which assigns the result of the `expression` to `name`. This directive serves the same purpose as `#define` in C. For example, we can use

```
NUM_OF_STUDENTS     EQU     90
```

to assign 90 to `NUM_OF_STUDENTS`. It is customary to use capital letters for these names in order to distinguish them from variable names. Then, we can write

```
                   .
                   .
mov     CX,NUM_OF_STUDENTS
                   .
                   .
cmp     AX,NUM_OF_STUDENTS
                   .
                   .
```

to move 90 into the CX register and to compare AX with 90. Defining constants this way has two advantages:

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

   ```
   mov     CX,NUM_OF_STUDENTS
   ```

   with

   ```
   mov     CX,90
   ```

   The first statement clearly indicates that we are moving the class size into the CX register.

2. Multiple occurrences of a constant can be changed from a single place. For example, if the class size has changed from 90 to 100, we need to change the value in the EQU statement only. If we didn't use the EQU statement, 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. Strings can be defined in a similar fashion as shown in the following example:

```
JUMP    EQU    jmp
```

Here JUMP is an alias for jmp. Thus, a statement like

```
JUMP    read_char
```

will be assembled as

```
jmp    read_char
```

The angle brackets (< and >) can be used to define strings that could potentially be interpreted as an expression. For example,

```
ARRAY_SIZE     EQU    <NUM_OF_ROWS * NUM_OF_COLS>
```

forces the assembler to treat

```
NUM_OF_ROWS * NUM_OF_COLS
```

as a string and will not evaluate it.

A Restriction: The symbols that have been assigned a value or a string cannot be reassigned another value or string in a given source module. If such redefinitions are required, you should use = directive, which is discussed next.

3.6.2 The = Directive

The = directive is similar to the EQU directive. The syntax, which is similar to that of the EQU directive, is

```
name = expression
```

There are two key differences:

1. A symbol that is defined by the = directive can be redefined. Therefore, the following code is valid.

   ```
   COUNT = 0
      .
      .
   COUNT = 99
   ```

2. The = directive cannot be used to assign strings or to redefine keywords or instruction mnemonics. For example,

   ```
   JUMP = jmp
   ```

   is not valid. For these purposes, you should use the EQU directive.

3.7 Illustrative Examples

This section presents five 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, 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.

Example 3.1 *Displays the ASCII value of the input key in binary*

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/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 program flow using the jz instruction. 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 {CX is used to store count}
    repeat
        if ((char AND mask) = 0)
        then
            write 0
        else
            write 1
        end if
        mask := mask/2 {can be done by shr}
        count := count − 1
     (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 3.4 follows the pseudocode in a straightforward way. Note that Pentium provides an instruction to perform integer division. However, shr is about 17 times faster than the divide instruction to divide a number by 2! More details about the division instructions are given in Chapter 6.

Program 3.4 Conversion of ASCII to binary representation

```
TITLE   Binary equivalent of characters   BINCHAR.ASM
COMMENT |
        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.
.MODEL SMALL
.STACK 100H
.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
INCLUDE io.mac
main    PROC
        .STARTUP
read_char:
        PutStr  char_prompt   ; request a char. input
        GetCh   AL            ; read input character
        nwln
        PutStr  out_msg1
        PutCh   AL
        PutStr  out_msg2
        mov     AH,80H        ; mask byte = 80H
        mov     CX,8          ; loop count to print 8 bits
print_bit:
        test    AL,AH         ; test does not modify AL
        jz      print_0       ; if tested bit is 0, print it
        PutCh   '1'           ; otherwise, print 1
        jmp     skip1
print_0:
        PutCh   '0'           ; print 0
skip1:
        shr     AH,1          ; right shift mask bit to test
                              ;  next bit of the ASCII code
        loop    print_bit
        nwln
        PutStr  query_msg     ; query user whether to terminate
        GetCh   AL            ; read response
        nwln
        cmp     AL,'Y'        ; if response is not 'Y'
        jne     read_char     ; read another character
done:                         ; otherwise, terminate program
        .EXIT
main    ENDP
        END   main
```

Example 3.2 *Displays the ASCII value of the input key in hexadecimal*

The objective of this example is to show how numbers can be converted to characters by using character manipulation. This and the next example are similar to the previous one except that the ASCII value is printed in hex. In order to get the least significant hex digit, we have to mask off the upper half of the byte and then perform integer to hex digit conversion. The example shown below assumes that the input character is L, whose ASCII value is 4CH.

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

Similarly, to get the most significant hex digit we have to isolate the upper half of the byte and move these 4 bits to the lower half, as shown below:

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

Notice that shifting right by 4 bit positions is equivalent to performing integer division by 16. The pseudocode of the program shown in Program 3.5 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 have to add the ASCII value for character 0 to convert it to its equivalent character. For instance, if the number is 5 (00000101B), it should be converted to character 5, whose ASCII value is 35H (00110101B). Therefore, we have to add 30H, which is the ASCII value of 0. This is done in Program 3.5 by

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

on line 34. If the number is between 10 and 15, we have to convert them to hex digits between A and F. You can verify that the required translation is achieved by

```
number - 10 + ASCII value for character A
```

In Program 3.5, this is done by

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

on line 37.

Program 3.5 Conversion to hexadecimal by character manipulation

```
 1:  TITLE   Hex equivalent of characters   HEX1CHAR.ASM
 2:  COMMENT |
 3:          Objective: To print the hex equivalent of
 4:                     ASCII character code.
 5:              Input: Requests a character from keyboard.
 6:             Output: Prints the ASCII code of the
 7:  |                  input character in hex.
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
11:  char_prompt     DB  'Please input a character: ',0
12:  out_msg1        DB  'The ASCII code of ''',0
13:  out_msg2        DB  ''' in hex is ',0
14:  query_msg       DB  'Do you want to quit (Y/N): ',0
15:
16:  .CODE
17:  .486
18:  INCLUDE io.mac
19:  main    PROC
20:          .STARTUP
21:  read_char:
22:          PutStr  char_prompt  ; request a char. input
23:          GetCh   AL           ; read input character
24:          nwln
25:          PutStr  out_msg1
26:          PutCh   AL
27:          PutStr  out_msg2
28:          mov     AH,AL        ; save input character in AH
29:          shr     AL,4         ; move upper 4 bits to lower half
30:          mov     CX,2         ; loop count - 2 hex digits to print
31:  print_digit:
32:          cmp     AL,9         ; if greater than 9
33:          jg      A_to_F       ; convert to A through F digits
34:          add     AL,'0'       ; otherwise, convert to 0 through 9
35:          jmp     skip
36:  A_to_F:
37:          add     AL,'A'-10    ; subtract 10 and add 'A'
```

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

Example 3.3 *Displays the ASCII value of the input key in hexadecimal using xlat instruction*

The objective of this example is to show how the use of `xlat` simplifies the solution of the last example. In this example, we use the `xlat` instruction to convert an integer value in the range between 0 and 15 to its equivalent hex digit. The program is shown in Program 3.6. To use `xlat` we have to construct a translation table, which is done by the following statement (line 17):

```
hex_table    DB    '0123456789ABCDEF'
```

We can then use the integer value as an index into the table. For example, an integer value of 10 points to `A`, which is the equivalent hex digit. In order to use the `xlat` instruction, BX should point to the base of the `hex_table` and AL should have the integer value between 0 and 15. The rest of the program is straightforward to follow.

Program 3.6 Conversion to hexadecimal by using the `xlat` instruction

```
 1:  TITLE   Hex equivalent of characters   HEX2CHAR.ASM
 2:  COMMENT |
 3:          Objective: To print the hex equivalent of
 4:                     ASCII character code. Demonstrates
 5:                     the use of xlat instruction.
```

```
             Input: Requests a character from keyboard.
           Output: Prints the ASCII code of the
|                  input character in hex.
.MODEL SMALL
.STACK 100H
.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
.486
INCLUDE io.mac
main    PROC
        .STARTUP
read_char:
        PutStr  char_prompt  ; request a char. input
        GetCh   AL           ; read input character
        nwln
        PutStr  out_msg1
        PutCh   AL
        PutStr  out_msg2
        mov     AH,AL        ; save input character in AH
        mov     BX,OFFSET hex_table  ; BX := 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
        PutStr  query_msg    ; query user whether to terminate
        GetCh   AL           ; read response
        nwln
        cmp     AL,'Y'       ; if response is not 'Y'
        jne     read_char    ; read another character
done:                        ; otherwise, terminate program
        .EXIT
main    ENDP
        END   main
```

Example 3.4 *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 3.7 is as follows:

```
main()
    display prompt message
    read input string
    index := 0
    char := string[index]
    while (char ≠ NULL)
        if ((char ≥ 'a') AND (char ≤ 'z'))
        then
            char := char + 'A' − 'a'
        end if
        display char
        index := index + 1
        char := string[index]
    end while
end main
```

You can see from Program 3.7 that the compound condition **if** requires two cmp instructions (lines 27 and 29). Also the program uses the BX register in indirect addressing mode and always holds the pointer value of the character to be processed. In Chapter 5 we will see a better way of accessing elements of an array. The end of the string is detected by

```
cmp    AL,0        ; check if AL is NULL
je     done
```

and is used to terminate the **while** loop (lines 25 and 26).

Program 3.7 Conversion to uppercase by character manipulation

```
TITLE   uppercase conversion of characters   TOUPPER.ASM
COMMENT |
        Objective: To convert lowercase letters to
                   corresponding uppercase letters.
            Input: Requests a character string from keyboard.
|          Output: Prints the input string in uppercase.
.MODEL SMALL
.STACK 100H
.DATA
name_prompt    DB  'Please type your name: ',0
out_msg        DB  'Your name in capitals is: ',0
in_name        DB  31 DUP (?)

.CODE
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  name_prompt  ; request character string
        GetStr  in_name,31   ; read input character string
        nwln
        PutStr  out_msg
        mov     BX,OFFSET in_name  ; BX := address of in_name
process_char:
        mov     AL,[BX]      ; 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     BX           ; BX points to next char.
        jmp     process_char ; go back to process next char.
        nwln
done:
        .EXIT
main    ENDP
        END   main
```

Example 3.5 *Sum of the individual digits of a number*

The last example shows how decimal digits can be converted from their character representations to equivalent binary. The program receives a number (maximum 10 digits) and displays the sum of the individual digits of the input number. For example, if the input number is 45213, the program displays 15. Since ASCII assigns a special set of contiguous values to the 10-digit characters, it is straightforward to get their numerical value (as discussed in Appendix A). All we have to do is to mask off the upper half of the byte, as is done in Program 3.8 by

```
and     AL,0FH
```

Alternatively, we can 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 3.8 as an exercise.

Program 3.8 Sum of individual digits of a number

```
TITLE   Add individual digits of a number   ADDIGITS.ASM
COMMENT |
        Objective: To find the sum of individual digits of
                   a given number. Shows character to binary
                   conversion of digits.
            Input: Requests a number from keyboard.
|          Output: Prints the sum of the individual digits.
.MODEL SMALL
.STACK 100H
.DATA
number_prompt  DB  'Please type a number (<11 digits): ',0
out_msg        DB  'The sum of individual digits is: ',0
number         DB  11 DUP (?)

.CODE
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  number_prompt  ; request an input number
        GetStr  number,11     ; read input number as a string
        nwln
        mov     BX,OFFSET number  ; BX := address of number
        sub     DX,DX         ; DX := 0 -- DL keeps the sum
repeat_add:
        mov     AL,[BX]      ; 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     BX           ; increment BX to point to next digit
        jmp     repeat_add   ;  and jump back
done:
        PutStr  out_msg
        PutInt  DX           ; write sum
        nwln
        .EXIT
main    ENDP
        END   main
```

3.8 Performance: When to Use the xlat Instruction

The xlat instruction is convenient to perform character conversions. Proper use of xlat will 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 3.2 and 3.3. This example is used in Experiment 2 to show the performance benefit that can be obtained from using the xlat instruction for the conversion.

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 3.4 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 75 characters.

We have 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 3.7.

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 BX to upper_table, the following code

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

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 3.1. In this plot, the x-axis gives the number of thousands of times the case conversion procedure is called to convert a lowercase string of 75 characters. The data shows that the use of xlat deteriorates the performance by about 30 percent! For reasons we have discussed previously, this is clearly a bad example for the use of the xlat instruction.

Experiment 2

We now use the hex conversion examples of Section 3.7 to show the benefits of the xlat instruction. As shown in Example 3.2, 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.

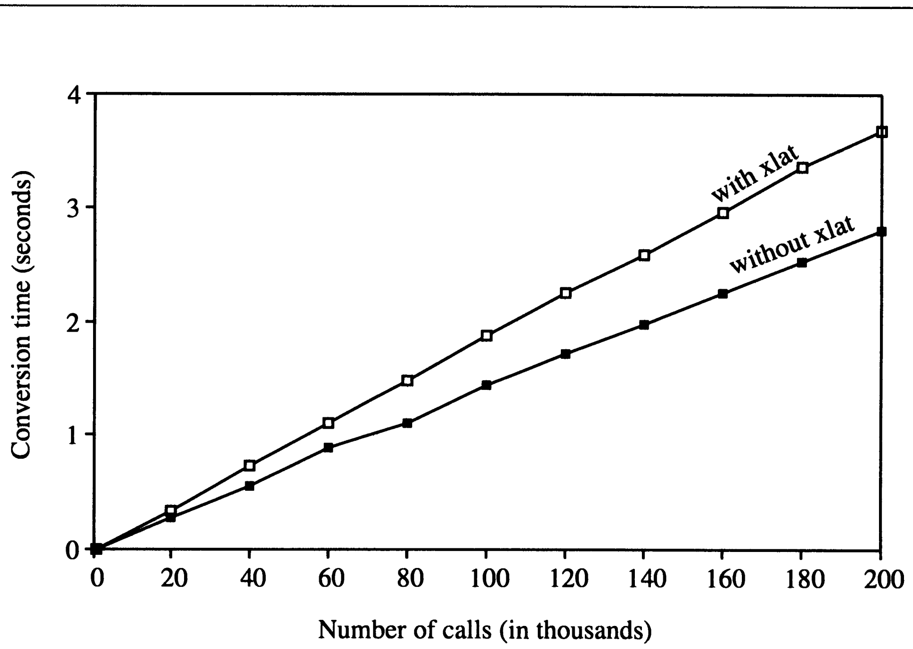


Figure 3.1 Performance of the case conversion program.

The two programs of Examples 3.2 and 3.3 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 75 characters.

The data presented in Figure 3.2 clearly demonstrates the benefit of using the `xlat` in this example. The procedure that uses `xlat` is about 80 percent faster!

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

3.9 Summary

The structure of the stand-alone assembly language program is described in Appendix B. In this chapter, we presented the basics of programming in the Pentium assembly language.

We discussed two types of assembly language statements:

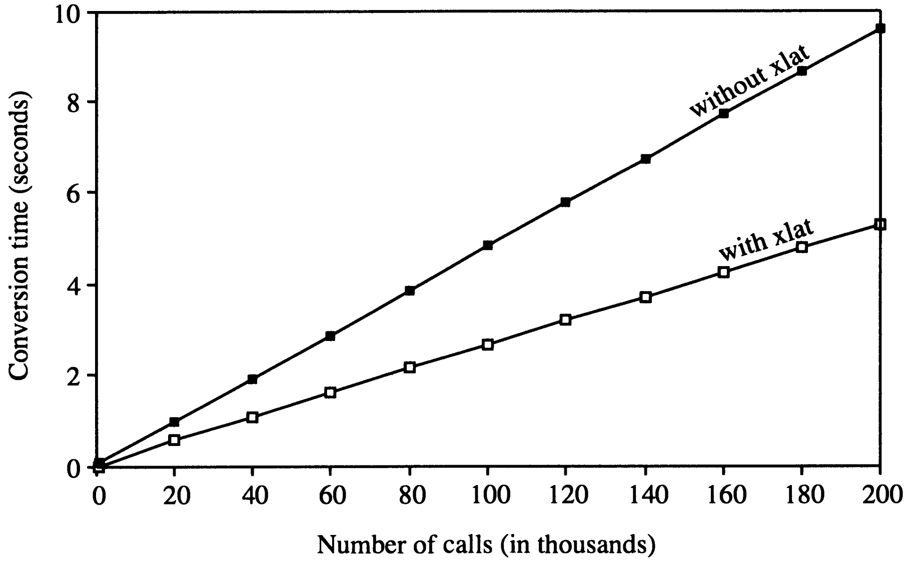


Figure 3.2 Performance of the hex conversion program.

1. Executable statements that instruct the CPU as to what to do
2. Assembler directives that facilitate the assembly process

Assembler directives to allocate storage space for data variables and to define numeric and string constants were discussed in detail. An overview of the Pentium instruction set was also presented. While we have 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.

3.10 Exercises

3–1 Why aren't assembler directives executed by the CPU?

3–2 What is the difference between the following two data definition statements?

```
int1    DB    2 DUP (?)
int2    DW    ?
```

3–3 For each of the following statements, what is the amount of storage space reserved (in bytes)? Also indicate the initialized data. Verify your answers using your assembler.

```
(a) table  DW  100 DUP (-1)
(b) nest1  DB  5 DUP (2 DUP ('%'), 3 DUP ('$'))
(c) nest2  DB  4 DUP (5 DUP (2 DUP (1),3 DUP (2)))
(d) value  DW  -2300
(e) count  DW  40000
(f) msg1   DB  'Finder''s fee is:',0
(g) msg2   DB  'Finder''s fee is:',0
(h) msg3   DB  'Sorry! Invalid input.',0DH,0AH,0
```

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

3–5 We discussed four addressing modes in this chapter. Which addressing mode is the most efficient one? Explain why.

3–6 Can we use the immediate addressing mode in the `inc` instruction? Justify your answer.

3–7 Discuss the pros and cons of using the `lea` instruction as opposed to using the `mov` instruction along with the `OFFSET` assembler directive.

3–8 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
```

3–9 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,DX               (l) sub     DX,CX
```

3–10 In each of the following code fragments, state whether `mov AX,10` or `mov BX,1` is executed:

```
(a)                              (b)
        mov     CX,5                     mov     CX,5
        sub     DX,DX                    mov     DX,10
        cmp     DX,CX                    shr     DX,1
        jge     jump1                    cmp     CX,DX
        mov     BX,1                     je      jump1
        jmp     skip1                    mov     BX,1
jump1:                                   jmp     skip1
        mov     AX,10            jump1:
skip1:                                   mov     AX,10
          . . .                  skip1:
                                           . . .
```

```
(c)                              (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:
                                           . . .
```

3–11 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
```

3–12 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
```

3.11 Progamming Exercises

3–P1 Modify the program of Example 3.1 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.

3–P2 Modify the program of Example 3.1 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.

3–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
```

3–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/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.

3–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 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.

3–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!)

3–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.

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

3–P9 This is the assembly language counterpart of the Chapter 2 exercise. Write a complete assembly language program to perform logical-address to physical-address translation. 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 16-bit segments and data.

In Chapter 2, you have written the same program in your favorite high-level language. The purpose is to compare the time required to write programs in assembly and high-level languages. Therefore, as you did for the Chapter 2 exercise, you should note the time spent on the exercise.

Compare and contrast your experience in using assembly and high-level languages. Make sure to include the debugging time as well in the comparison.

Part II

Basic Topics

Chapter 4

Procedures and the Stack

Objectives

- To introduce the stack and its implementation in Pentium
- To describe stack operations and the use of the stack
- To present procedures and parameter passing mechanisms
- To introduce separate assembly of source program modules
- To discuss procedure overheads

Chapter 3 gave an overview of assembly language programs. Starting with this chapter, we will discuss the constructs and instructions of the Pentium assembly language in detail. We start our discussion with procedures. A procedure is an important programming construct that facilitates modular programming. The stack plays an important role in procedure invocation and execution. Section 4.1 introduces the stack concept and the next section discusses how the stack is implemented in Pentium. Stack operations—push and pop—are discussed in Section 4.3. Section 4.4 discusses stack uses.

After a brief introduction to procedures (Section 4.5), assembly language directives for writing procedures are discussed in Section 4.6. Section 4.7 presents the Pentium instructions for procedure invocation and return. Parameter passing mechanisms are discussed in detail in Section 4.8. Stack plays an important role in parameter passing. Using the stack it is relatively straightforward to pass a variable number of parameters to a procedure (discussed in Section 4.9). The issue of local variable storage in procedures is discussed in Section 4.10.

While short assembly language programs can be stored in a single file, real application programs are likely to be broken into several files, called modules.

S. P. Dandamudi, *Introduction to Assembly Language Programming*

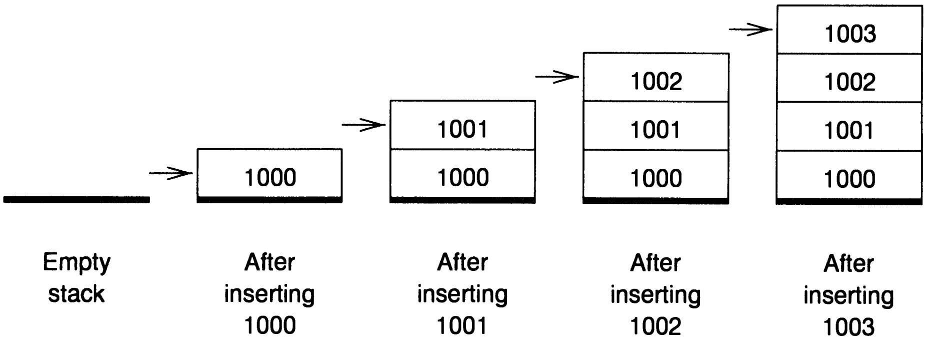


Figure 4.1 An example showing stack growth—numbers 1000 through 1003 are inserted in ascending order. The arrow points to the top-of-stack.

The issues involved in writing and assembling multiple source program modules are discussed in Section 4.11.

While modular programming encourages the use of procedures, there is a certain overhead associated with the use of procedures. This overhead is quantified in Section 4.12. The last section provides a summary of the chapter.

4.1 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 the stack of trays would be the last tray that had been placed on the stack. There are two operations associated with a stack—insertion and deletion. If we view the stack as a linear array of elements, both insertion and deletion operations are restricted to one end of the array. Thus, the only element that is directly accessible is the element at the top-of-stack (TOS). In stack terminology, insert and delete operations are referred to as *push* and *pop* operations, respectively.

There is another data structure—the *queue*—that you are familiar with in your day-to-day activities. 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 4.1. The arrow points to the top-of-stack. When

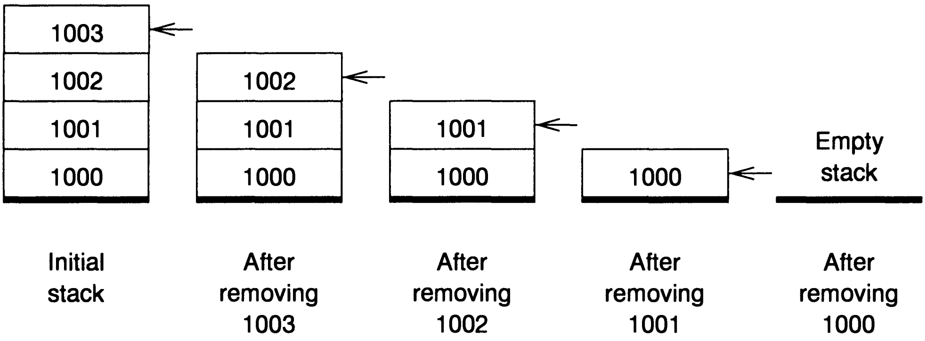


Figure 4.2 An example showing deletion of data items from a stack. The arrow points to the top-of-stack.

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 4.2).

In contrast, a queue maintains the order. Suppose that the numbers 1000 through 1003 are inserted into a queue as in the stack example. When removing the numbers from the queue, the first number to enter the queue would be the one to come out first. Thus, the numbers deleted from the queue would maintain their insertion order.

4.2 Pentium Implementation of the Stack

The set of memory locations reserved in the stack segment is used for implementing the stack. The registers SS and (E)SP are used to implement the Pentium stack. If 16-bit address size segments are used as we do in this book, SP is used as the stack pointer; ESP is used for 32-bit address size segments. In the rest of the chapter, we focus on 16-bit segments.

The top-of-stack, which points to the last item inserted into the stack, is indicated by SS:SP, with the SS register pointing to the beginning of the stack segment, and the SP register giving the offset value of the last item inserted relative to the beginning of the stack segment.

The Pentium stack implementation characteristics are:

- Only words (i.e., 16-bit data) or doublewords (i.e., 32-bit data) are saved on the stack, never a single byte.

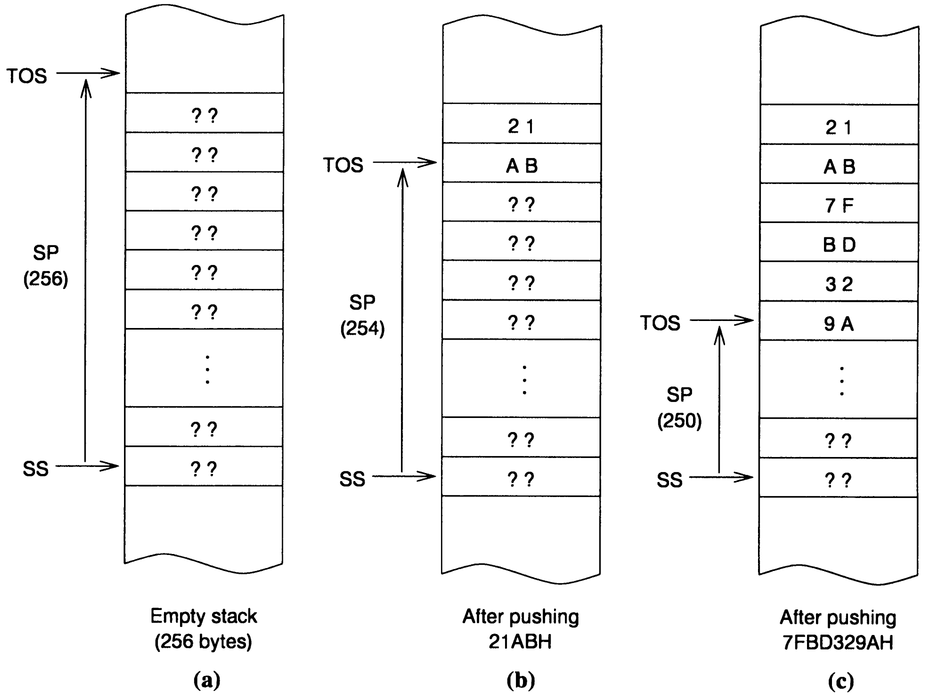


Figure 4.3 Stack implementation in Pentium—SS:SP points to the top-of-stack.

- The stack grows toward lower memory addresses. Since we graphically represent memory with addresses increasing from the bottom of a page to top, we say that the stack grows "downward."
- Top-of-stack (TOS) always points to the last data item placed on the stack. TOS, which is represented by SS:SP, always points to the lower byte of the last word data inserted into the stack.

The statement

```
.STACK 100H
```

creates an empty stack as shown in Figure 4.3*a*, and allocates 256 (i.e., 100H) bytes of memory for stack operations. When the stack is initialized, TOS points to a byte just outside the reserved stack area. It is an error to read from an empty stack as this causes a *stack underflow*.

When a data item is pushed onto the stack, the SP is first decremented by 2 and then the word data is stored at SS:SP. Since Pentium is a little endian, a

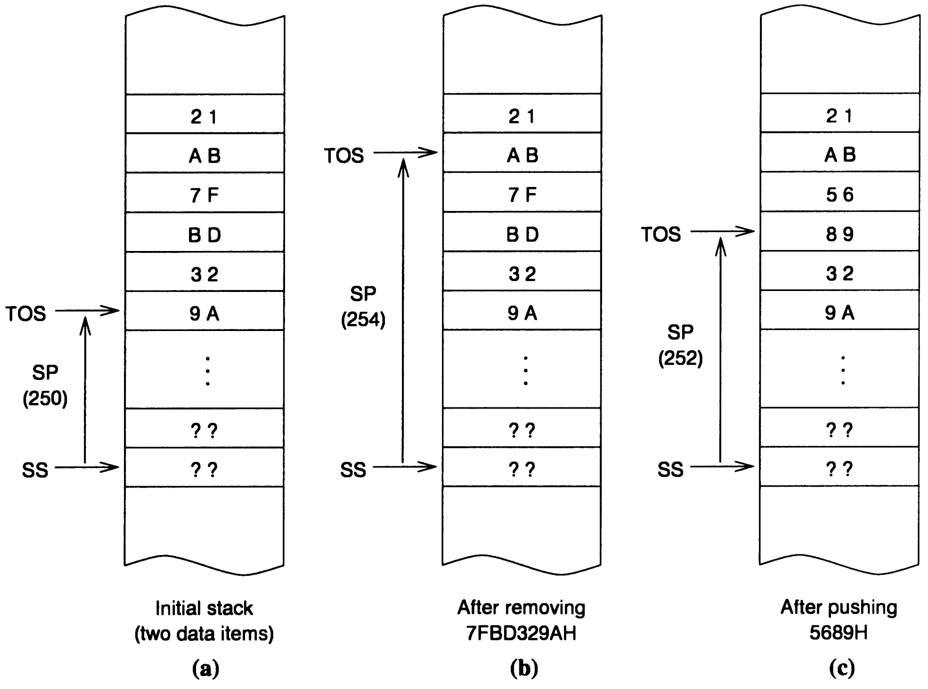


Figure 4.4 An example showing stack insert and delete operations.

higher-order byte is stored in the higher memory address. For instance, when we push the data word 21ABH, the stack expands by 2 bytes and the SP is decremented by 2 to point to the last data item, as shown in Figure 4.3*b*. The stack shown in Figure 4.3*c* results when we expand the stack further by 4 more bytes by pushing a doubleword 7FBD329AH onto the stack.

The stack full condition is indicated by the zero offset value (i.e., the SP register is 0000H). If we try to insert a data item into a full stack, *stack overflow* occurs. Both stack underflow and overflow are programming errors and should be handled with care.

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 4.4*a*, we get 7FBD329AH from the stack and SP is updated, as shown in Figure 4.4*b*. Notice that the four memory locations retain their values. However, since TOS is updated, these four locations will be used to store the next data value pushed onto the stack, as shown in Figure 4.4*c*.

Table 4.1 Stack operations on 16- and 32-bit data

`push source16`	SP := SP − 2 (SS:SP) := `source16`	SP is first decremented by 2 to modify TOS. Then the 16-bit data from `source16` is copied onto the stack at the new TOS. The stack expands by 2 bytes.
`push source32`	SP := SP − 4 (SS:SP) := `source32`	SP is first decremented by 4 to modify TOS. Then the 32-bit data from `source32` is copied onto the stack at the new TOS. The stack expands by 4 bytes.
`pop dest16`	`dest16` := (SS:SP) SP := SP + 2	The data item located at TOS is copied to `dest16`. Then SP is incremented by 2 to update TOS. The stack shrinks by 2 bytes.
`pop dest32`	`dest32` := (SS:SP) SP := SP + 4	The data item located at TOS is copied to `dest32`. Then SP is incremented by 4 to update TOS. The stack shrinks by 4 bytes.

4.3 Stack Operations

4.3.1 Basic Instructions

The stack data structure allows two basic operations: insertion of a data item into the stack (called the *push* operation), and deletion of a data item from the stack (called the *pop* operation). Pentium allows these two operations on word or doubleword data items. The syntax is

```
push   source
pop    destination
```

The operand of these two instructions can be a 16- or 32-bit general-purpose register, segment register, or a word or doubleword in memory. In addition, `source` for the push instruction can be an immediate operand of size 8, 16, or 32 bits. Table 4.1 summarizes the two stack operations.

On an empty stack created by

```
.STACK 100H
```

the following statements

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

would result in the stack shown in Figure 4.4*a*. Executing the following statement

```
pop     EBX
```

on this stack would result in the stack shown in Figure 4.4*b* with the register EBX receiving 7FBD329AH.

4.3.2 Additional Instructions

Pentium supports two special instructions for stack manipulation. These instructions can be used to save and restore the flags and the set of general-purpose registers.

Stack Operations on Flags

The push and pop operations cannot be used to save and retrieve the flags register. For this, Pentium provides two special versions of these instructions:

```
pushf   (push 16-bit flags)
popf    (pop 16-bit flags)
```

These instructions do not need any operands. For operating on 32-bit flags register (EFLAGS), we can use pushfd and popfd instructions.

Stack Operations on All General-Purpose Registers

Pentium also provides special pusha and popa instructions to save and restore the eight general-purpose registers. The pusha saves the 16-bit general-purpose registers AX, CX, DX, BX, SP, BP, SI, and DI. These registers are pushed in the order specified. The last register pushed is the DI register. The popa restores these registers except that it will not copy the SP value (i.e., the SP value is not loaded into the SP register as a part of popa instructions). The corresponding instructions for 32-bit registers are pushad and popad. These instructions are useful in procedure calls, as explained in Section 4.8.2.

4.4 Uses of the Stack

A stack is used for three main purposes: (i) as a scratch pad to temporarily store data, (ii) for transfer of program control, and (iii) for passing parameters during a procedure call.

4.4.1 Temporary Storage of Data

A stack can be used as a scratch pad to store data on a temporary basis. For example, consider exchanging 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 following 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. Here is a better solution that does not require any spare registers.

```
xchg    EAX,value1
xchg    EAX,value2
xchg    EAX,value1
```

Note that, even though the EAX register is used in this code, its value is preserved. This code, however, requires six memory operations—two for each xchg instruction. Using the stack, the exchange can be done by

```
push    value1
push    value2
pop     value1
pop     value2
```

Notice that the above code does not use any general-purpose registers and requires only four memory operations. Another point to note is that push and pop instructions allow movement of data from (data segment) memory to (stack segment) memory. Recall that, normally, the mov instruction does not allow data transfer from memory to memory. Stack operations are an exception!

The stack is also frequently used as a scratch pad 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 (discussed in Section 4.8). The following code shows an example that frees up EBX and ECX registers by saving their contents on the stack. After using the registers, their original values are restored from the stack.

```
          . . .
;save EBX and ECX registers on the stack
     push   EBX
     push   ECX
          . . .
     EBX and ECX registers
     can now be used
          . . .
;restore EBX and ECX registers from the stack
     pop    ECX
     pop    EBX
          . . .
```

Because the stack is a LIFO data structure, the sequence of pop instructions is a mirror image of the push instruction sequence.

It should be clear from these examples that the stack grows and shrinks during the course of a program execution. It is important to allocate enough storage space for the stack, as stack overflow and underflow could cause unpredictable results—often causing system errors.

4.4.2 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 that follows the procedure call instruction is stored on the stack so that the control can be transferred back to the calling program at the end of procedure execution. A detailed discussion on this topic is presented in Section 4.7.

4.4.3 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 4.8.

4.5 Procedures

A procedure is a logically self-contained unit of code designed to perform a particular task. These are sometimes referred to as *subprograms* and play an important role in modular program development. In high-level languages such as Pascal, there are two types of subprograms: *procedures* and *functions*. There is a strong similarity between a Pascal function and a mathematical function. Each function receives a list of arguments and performs a computation based on the arguments passed onto it and returns a single value. Procedures also receive a list of arguments just like functions. However, procedures, after performing their computation based on the values of the arguments, 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 use by the function. Thus, in this case, the values of these actual parameters are not changed in the called function—these values can only be used like in a mathematical function. In our example, the sum function is invoked by using the call-by-value mechanism, as we simply pass the values of number1 and number2 to the called function sum.

In the call-by-reference mechanism, the called function actually receives the addresses (i.e., pointers) of the parameters from the calling function. The function can change the contents of these parameters—and these changes will be seen by the calling function—by directly manipulating the actual parameter storage space. For instance, the following swap function

```
void swap (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

assumes that swap receives the addresses of the two parameters from the calling function. Thus, we are using the call-by-reference mechanism for parameter passing. Such a function can be invoked by

```
swap (&data1, &data2);
```

Often both types of parameter passing mechanisms are used in the same function. As an example, consider finding the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

The two roots are defined as

$$\text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\text{root2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

The roots are real if $b^2 \geq 4ac$, and imaginary otherwise.

Suppose that we want to write a function that receives a, b, and c and returns the values of the two roots (if real) and indicates whether the roots are real or imaginary.

```
int roots (double a, double b, double c,
           double *root1, double *root2)
{
    int root_type = 1;
    if (4 * a * c <= b * b){  /* roots are real */
       *root1 = (-b + sqrt(b*b - 4*a*c))/(2*a);
       *root2 = (-b - sqrt(b*b - 4*a*c))/(2*a);
    }
    else    /* roots are imaginary */
       root_type = 0;
    return (root_type);
}
```

The function roots receives parameters a, b, and c by the call-by-value mechanism, and the parameters root1 and root2 by the call-by-reference mechanism. A typical invocation of roots is

```
root_type = roots (a, b, c, &root1, &root2);
```

In summary, procedures receive a list of parameters, which may be passed either by the call-by-value or the call-by-reference mechanism. If more than one result is to be returned back by a called procedure, the call-by-reference parameter passing mechanism should be used.

4.6 Assembler Directives for Procedures

Assemblers provide two directives to define procedures in the assembly language: PROC and ENDP. The PROC directive (stands for PROCedure) signals the beginning of a procedure, and ENDP (END Procedure) indicates the end of a procedure. Both these directives take a label that is the name of the procedure. In addition, the PROC directive may optionally include NEAR or FAR to indicate whether the procedure is a NEAR procedure or a FAR procedure. The general format is

```
proc-name    PROC    NEAR
```

to define a near procedure, and

```
proc-name    PROC    FAR
```

to define a far procedure.

A procedure is said to be a near procedure if the calling and called procedures are both located in the same code segment. On the other hand, if the calling and called procedures are located in two different code segments, they are called far procedures. Near procedures involve intrasegment calls, while far procedures involve intersegment calls. Here we restrict our attention to near procedure calls.

When NEAR or FAR is not included in the PROC directive, the procedure definition defaults to NEAR procedure. Thus, the following two statements define a near procedure:

```
proc-name    PROC    NEAR
```

or

```
proc-name    PROC
```

A typical procedure definition is

```
proc-name PROC
          . . .
      <procedure body>
          . . .
proc-name ENDP
```

where proc-name in both PROC and ENDP statements must match.

4.7 Pentium Instructions for Procedures

Pentium provides call and ret (return) instructions to write procedures in assembly language. The call instruction can be used to invoke a procedure, and has the format

```
call    proc-name
```

where proc-name is the name of the procedure to be called. The assembler replaces proc-name by the offset value of the first instruction of the called procedure.

4.7.1 How Is Program Control Transferred?

The offset value provided as a part of 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. Look at the following example.

```
offset   machine code
(in hex)   (in hex)
                     main  PROC
                             . . .
cs:000A     E8000C           call   sum
cs:000D     8BD8             mov    BX,AX
                             . . .
                     main  ENDP

                     sum   PROC
cs:0019     55               push   BP
                             . . .
                             . . .
                     sum   ENDP

                     avg   PROC
                             . . .
                             . . .
cs:0028     E8FFEE           call   sum
cs:002B     8BD0             mov    DX,AX
                             . . .
                     avg   ENDP
```

The call instruction of main is located at CS:000AH and the next instruction at CS:000DH. The first instruction of the procedure sum is at CS:0019H in the code segment. After the call instruction has been fetched, the IP register points to the next instruction to be executed (see Chapter 2) i.e., IP = 000DH. This is the instruction that should be executed after completing the execution of sum. The processor makes a note of this by pushing the contents of the IP register onto the stack.

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

```
push BP
```

instruction in sum. To do this, the processor adds the 16-bit relative displacement found in the call instruction to the contents of the IP register. Proceeding with our example, the machine language encoding of the call instruction, which requires three bytes, is E8000CH. The first byte E8H identifies that the instruction is the call (i.e., opcode), and the next two bytes provide a (signed) relative displacement in bytes. In this example, it is the difference between 0019H (offset of the push BP instruction in sum) and 000DH (offset of the instruction mov BX,AX in main). Therefore, 0019H−000DH = 000CH. Adding this difference to the contents of the IP register (after fetching the call instruction) leaves the IP register pointing to the first instruction of sum.

Note that the procedure call in main is a forward call, and therefore the relative displacement is a positive number. As an example of a backward procedure call, let us look at the sum procedure call in the avg procedure. In this case, the program control has to be transferred back. That is, the displacement is a negative value. Following the explanation given in the last paragraph, the displacement is 0019H − 002BH = FFEEH. Since negative numbers are expressed in 2's complement notation, FFEEH corresponds to −12H (i.e., −18D), which is the displacement value in bytes.

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

```
SP := SP − 2          ; push return address onto the stack
(SS:SP) := IP
P := IP + relative displacement   ; update IP to point to the procedure
```

The relative displacement is a signed 16-bit number to accommodate both forward and backward procedure calls.

4.7.2 The ret Instruction

The ret (return) instruction is used to transfer control from the called procedure to the calling procedure. Return transfers control to the instruction following the call (instruction mov BX,AX in the example given on page 129). 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. As a part of executing the `ret` instruction, the return address from the stack is recovered. The actions taken during the execution of the `ret` instruction are:

```
IP := (SS:SP)     ; pop return address at TOS into IP
SP := SP + 2      ; update TOS by adding 2 to SP
```

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

```
ret 6
```

The details on this optional number are covered in Section 4.8.2, which discusses the stack-based parameter passing mechanism.

4.8 Parameter Passing

Parameter passing in assembly language is different and more complicated than that used in high-level languages. In assembly language, the calling procedure first places all the parameters needed by the called procedure in a mutually accessible storage area (usually registers or memory). Only then can the procedure be invoked. There are two common methods depending on the type of storage area used to pass parameters: *register method* or *stack method.* As their names imply, the register method uses general-purpose registers to pass parameters, while the stack is used in the other method.

4.8.1 Register Method

In the register method, the calling procedure places the necessary parameters in the general-purpose registers before invoking the procedure. Next, let us look at a couple of examples before considering the advantages and disadvantages of passing parameters using the register method.

Example 4.1 *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 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 4.9, requests two integers from the user and displays the sum on the screen.

Program 4.9 Parameter passing by call-by-value using registers

```
TITLE   Parameter passing via registers      PROCEX1.ASM
COMMENT |
        Objective: To show parameter passing via registers
            Input: Requests two integers from the user.
|          Output: Outputs the sum of the input integers.
.MODEL SMALL
.STACK 100H
.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
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg1    ; request first number
      GetInt  CX             ; CX := first number
      nwln
      PutStr  prompt_msg2    ; request second number
      GetInt  DX             ; DX := second number
      nwln
      call    sum            ; returns sum in AX
      PutStr  sum_msg        ; display sum
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP

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

Example 4.2 *Parameter passing by call-by-reference using registers*

This example shows how parameters can be passed by call-by-reference using the register method. The program requests a character string from the user and displays the number of characters in the string (i.e., string length). The string length is computed by the function `str_len`. This function receives a pointer to the string in the BX register and returns the string length in the AX register. The `main` procedure executes

```
mov     BX,OFFSET string
```

to place the address of `string` in BX (line 23) before invoking the procedure on line 24. The `str_len` procedure 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 program listing is given in Program 4.10. Note that even though the procedure modifies the BX register during its execution, it restores the original value of BX (pointing to the string) by saving its value initially on the stack (line 37) and restoring it (line 46) before returning to the `main` procedure.

Program 4.10 Parameter passing by call-by-reference using registers

```
 1:  TITLE   Parameter passing via registers      PROCEX2.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via registers
 4:              Input: Requests a character string from the user.
 5:  |          Output: Outputs the length of the input string.
 6:
 7:  BUF_LEN     EQU  41           ; string buffer length
 8:  .MODEL SMALL
 9:  .STACK 100H
10:  .DATA
```

```
string      DB  BUF_LEN DUP (?)  ;input string < BUF_LEN chars.
prompt_msg  DB  'Please input a string: ',0
length_msg  DB  'The string length is ',0

.CODE
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg     ; request string input
      GetStr  string,BUF_LEN ; read string from keyboard
      nwln
      mov     BX,OFFSET string  ; BX := string address
      call    str_len        ; returns string length in AX
      PutStr  length_msg     ; display string length
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP

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

Pros and Cons of the Register Method

The register method has its advantages and disadvantages. These are summarized here.

Advantages

1. The register method is convenient and easier to pass a small number of parameters.
2. This method is also faster because all the parameters are available in registers.

Disadvantages

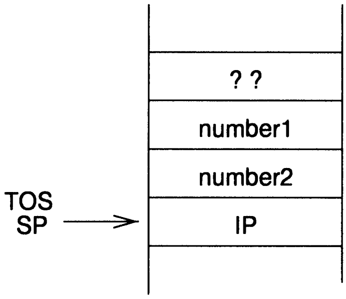
1. The main disadvantage is that only a few parameters can be passed by using registers, as there are a limited number of general-purpose registers available in the CPU.
2. Another problem is that the general-purpose registers are often used by the calling procedure for some other purpose. Thus, it is necessary to temporarily save the contents of these registers on the stack to free them for use in parameter passing before calling a procedure, and restore them after returning from the called procedure. In this case, it is difficult to realize the second advantage listed above, as the stack operations involve memory access.

4.8.2 Stack Method

In this method of parameter passing, all parameters required by a procedure are pushed onto the stack before the procedure is called. As an example, let us consider passing the two parameters required by the sum procedure shown in Program 4.9. This can be done by

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

The stack would look as shown below after executing the call instruction, which automatically pushes the IP contents onto the stack.



Since the parameter values are buried inside the stack, first we have to pop the IP value to read the required two parameters. This, for example, can be done by

```
pop     AX
pop     BX
pop     CX
```

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

```
push    AX
```

so that TOS is pointing to the return address.

The main problem with this code is that we need to set aside general-purpose registers to copy parameter values. This means that the calling procedure cannot use these registers for any other purpose. Worse still, what if you want to pass 10 parameters? One way to free up registers is to copy the parameters from the stack to local data variables, but this is impractical and inefficient!

The best way to get parameter values is to leave them on the stack and read them off the stack as needed. Since the stack is a sequence of memory locations, SP+2 points to `number2`, and SP+4 to `number1`. Unfortunately, in 16-bit addressing mode

```
mov     BX,[SP+2]
```

is not allowed. However, we can increment SP by 2 and use SP in indirect addressing mode, as shown below:

```
add     SP,2
mov     BX,[SP]
```

A problem with this solution is that it is very cumbersome, as we have to remember to update SP to point to the return address on the stack before the end of the procedure.

In 32-bit addressing mode, we can use ESP with a displacement to point to a parameter on the stack. For instance,

```
mov     BX,[ESP+2]
```

can be used, but this causes another problem. The stack pointer register is modified by push and pop stack instructions. As a result, the relative offset changes with the stack operations performed in the called procedure. This is not a desirable situation.

There is a better alternative—we can use the BP register instead of SP to specify an offset into the stack segment. For example, we can copy the value of number2 into the AX register by

```
mov     BP,SP
mov     AX,[BP+2]
```

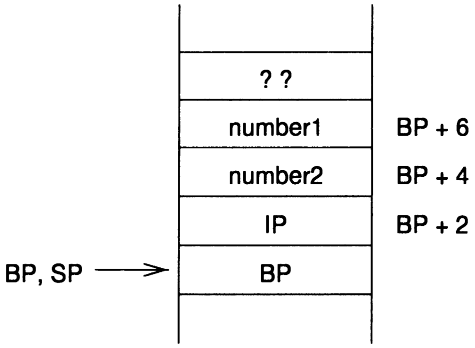
This is the usual way of pointing to the parameters on the stack. Since every procedure uses the BP register itself to access parameters, the BP register should be preserved. Therefore, we should save the contents of the BP register before executing the

```
mov     BP,SP
```

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

```
push    BP
mov     BP,SP
```

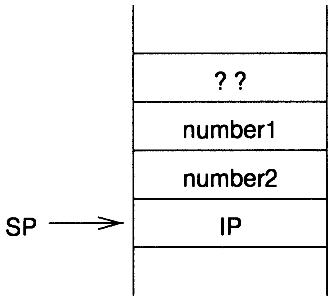
causes the parameter displacement to increase by 2 bytes, as shown below.



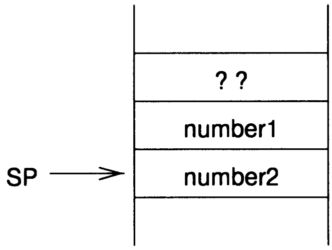
Before returning from the procedure

```
pop     BP
```

restores the original value of BP and the resulting stack state is shown below.



The `ret` statement discussed in Section 4.7.2 causes the return address to be placed in the IP register, and the stack after `ret` is shown below.



Now the problem is that the 4 bytes of the stack occupied by the 2 parameters are no longer useful. One way to free the 4 bytes of stack storage is to increment SP by 4 after the call statement, as shown below.

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

For example, the Turbo C compiler uses this method to clear parameters from the stack. The above assembly language code segment corresponds to the

```
sum(number2, number1);
```

function call in C.

Rather than adjusting the stack by the calling procedure, the called procedure can also clear the stack. Note that we cannot write

```
sum     PROC
          .
          .
```

```
        add SP,4
        ret
sum     ENDP
```

because when `ret` is executed, SP should point to the contents of IP on the stack.

The solution lies in the optional operand that can be specified in the `ret` statement. The format is

```
ret     optional-value
```

which results in the following sequence of actions:

IP := (SS:SP)
SP := SP + 2 + optional-value

The optional-value should be a number (i.e., immediate). Since the purpose of the optional-value is to discard the parameters pushed onto the stack before the execution of the corresponding call instruction, this operand usually takes a positive value.

Who Should Clean up the Stack?

We have discussed two 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, there is no apparent advantage of one method over the other. Mostly, it is a question of personal choice or conforming to an existing convention. In this book, we follow the convention that parameter discarding is done by the called procedure.

4.8.3 Preserving Calling Procedure State

It is important to preserve the contents of the registers across a procedure call. The necessity for this is illustrated by the following code.

```
                . . .
          mov    CX,count
repeat:
          call   compute
                . . .
                . . .
          loop   repeat
                . . .
```

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

Suppose, now, that the `compute` procedure uses the CX register during its computation. Then, when `compute` returns control back to the calling program, CX would have changed and the program logic would be incorrect.

Since there are a limited number of registers and registers should be used for writing efficient code, registers should be preserved. The stack is used to save registers temporarily.

4.8.4 Which Registers Should Be Saved?

The answer to this question is simple: Save those registers that are used by the calling procedure but changed by the called procedure. This leads to the following question: Which procedure, the calling or the called, should save the registers?

Usually, one or two registers are used to return a value by the called procedure. Therefore, such register(s) do not have to be saved. For example, in Turbo C, an integer C function returns the result in the AX register; if the function returns a `long int` data type result, which requires 32 bits, both the AX and DX registers are used to return the result.

In order to avoid the selection of the registers to be saved, we could save, blindly, all registers each time a procedure is invoked. For instance, we could use the `pusha` instruction (see page 123). But such an action may result in unnecessary overhead, as `pusha` takes five clocks to push all eight registers, whereas an individual register `push` instruction takes only one clock. Recall that producing efficient code is an important motivation for using 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 is 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 set of 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 back before returning to the calling procedure. This also conforms to modular program design principles.

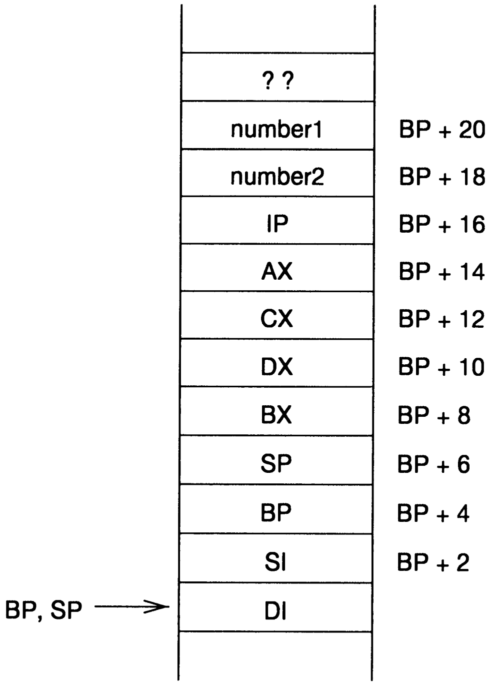
When to Use pusha

The pusha instruction is useful in certain instances, but not all! We identify two instances where pusha is not useful. First, what if some of the registers saved by pusha are used for returning results? For instance, Turbo C uses the AX register for returning a 16-bit result and the DX:AX register pair for a 32-bit result. In this case pusha is not really useful, as popa destroys the result to be returned to the calling procedure. Second, since pusha takes five clocks whereas a single push takes only a single clock, pusha is efficient only if you want to save more than four registers. In some instances where you want to save only one or two registers, it may be worthwhile to use a push instruction. Of course, the other side of the coin is that pusha improves readability of code and reduces memory required for the instructions.

When pusha is used to save registers, it modifies the offset of the parameters. Note that

```
pusha
mov     BP,SP
```

causes the stack state, shown below, to be different from that shown on page 137. You can see that the offset of number1 and number2 increases.



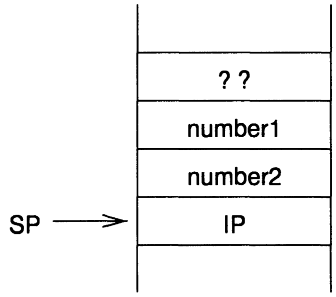
4.8.5 Illustrative Examples

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

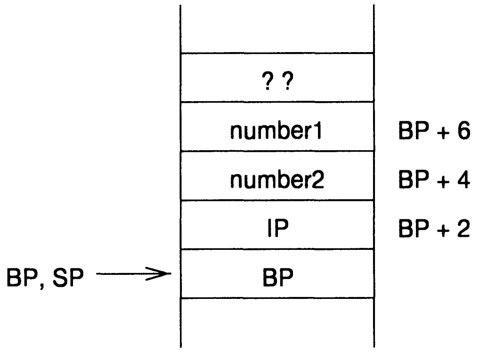
Example 4.3 *Parameter passing by call-by-value using the stack*

This is the stack counterpart of Example 4.1, which passes two integers to the procedure sum. The procedure returns the sum of these two integers in the AX register, as in Example 4.1. The program listing is given in Program 4.11.

The program requests two integers from the user. It reads the two numbers into the CX and DX registers using GetInt (lines 19 and 22). 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 24 and 25). When the control is transferred to sum, the state of the stack is:



As discussed in Section 4.8.2, the BP register is used to access the two parameters from the stack. Therefore, we have to save BP itself on the stack (line 39), which changes the stack to:



The original value of BP is restored at the end of the procedure (line 43). Accessing the two numbers follows the explanation given in Section 4.8.2. Note that the first number is at BP+6, and the second one at BP+4. As in Example 4.1, 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 4 bytes in the stack.

Program 4.11 Parameter passing by call-by-value using the stack

```
TITLE   Parameter passing via the stack      PROCEX3.ASM
COMMENT |
        Objective: To show parameter passing via the stack
            Input: Requests two integers from the user.
|          Output: Outputs the sum of the input integers.
.MODEL SMALL
.STACK 100H
.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
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg1    ; request first number
      GetInt  CX             ; CX := first number
      nwln
      PutStr  prompt_msg2    ; request second number
      GetInt  DX             ; DX := second number
      nwln
      push    CX             ; place first number on stack
      push    DX             ; place second number on stack
      call    sum            ; returns sum in AX
      PutStr  sum_msg        ; display sum
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP

```

```
34:  ;-----------------------------------------------------------
35:  ;Procedure sum receives two integers via the stack.
36:  ; The sum of the two integers is returned in AX.
37:  ;-----------------------------------------------------------
38:  sum   PROC
39:        push    BP              ; we will use BP, so save it
40:        mov     BP,SP
41:        mov     AX,[BP+6]       ; sum := first number
42:        add     AX,[BP+4]       ; sum := sum + second number
43:        pop     BP              ; restore BP
44:        ret     4               ; return and clear parameters
45:  sum   ENDP
46:        END     main
```

Example 4.4 *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 4.12, 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 24–27). The swap procedure, after saving the BP register as in the last example, can access the pointers of the two characters at BP+4 and BP+6. Since the procedure uses the BX register, we save it on the stack as well. Note that, once the BP is pushed onto the stack and the SP value is copied to BP, the two parameters (i.e., the two character pointers in this example) are available at BP+4 and BP+6, irrespective of the other stack push operations in the procedure.

Program 4.12 Parameter passing by call-by-reference using the stack

```
 1:  TITLE   Parameter passing via the stack      PROCSWAP.ASM
 2:  COMMENT |
 3:          Objective: To show parameter passing via the stack
 4:              Input: Requests a character string from the user.
 5:             Output: Outputs the input string with the first
 6:  |                  two characters swapped.
 7:
 8:  BUF_LEN     EQU  41           ; string buffer length
 9:  .MODEL SMALL
```

```
.STACK 100H
.DATA
string      DB  BUF_LEN DUP (?)  ;input string < BUF_LEN chars.
prompt_msg  DB  ’Please input a string: ’,0
output_msg  DB  ’The swapped string is: ’,0

.CODE
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg     ; request string input
      GetStr  string,BUF_LEN ; read string from the user
      nwln
      mov     AX,OFFSET string  ; AX := string[0] pointer
      push    AX             ; push string[0] pointer on stack
      inc     AX             ; AX := string[1] pointer
      push    AX             ; push string[1] pointer on stack
      call    swap           ; swaps the first two characters
      PutStr  output_msg     ; display the swapped string
      PutStr  string
      nwln
done:
      .EXIT
main  ENDP

;-----------------------------------------------------------
;Procedure swap receives two pointers (via the stack) to
; characters of a string. It exchanges these two characters.
;-----------------------------------------------------------
swap  PROC
      push    BP             ; save BP - procedure uses BP
      mov     BP,SP          ; copy SP to BP
      push    BX             ; save BX - procedure uses BX
      ; swap begins here. Because of xchg, AL is preserved.
      mov     BX,[BP+6]      ; BX := first character pointer
      xchg    AL,[BX]
      mov     BX,[BP+4]      ; BX := second character pointer
      xchg    AL,[BX]
      mov     BX,[BP+6]      ; BX := first character pointer
      xchg    AL,[BX]
      ; swap ends here
      pop     BX             ; restore registers
      pop     BP
```

```
        ret     4              ; return and clear parameters
swap    ENDP
        END     main
```

Example 4.5 *Bubble sort procedure*

This program requests a set of up to 20 nonzero integers from the user and displays them in sorted order. The input can be terminated earlier by typing a zero.

The logic of the main program is straightforward. The `read_loop` (lines 26–34) reads the input integers. Since the CX is initialized to `MAX_SIZE`, which is set to 20 in this program, the `read_loop` iterates a maximum of twenty times. The reading of input integers can also be terminated by typing a zero. The zero input condition is detected and the loop is terminated by statements on lines 29 and 30.

The `bubble_sort` procedure receives the size of the array to be sorted and a pointer to the array. These two parameters are pushed onto the stack (lines 36 and 37) before calling the `bubble_sort` procedure. The `print_loop` (lines 41–47) displays the sorted array.

Bubble sort: A detailed description of the bubble sort algorithm is given in Chapter 1. Here we present the pseudocode for the `bubble_sort` procedure:

```
bubble_sort (arrayPointer, arraySize)
    status := UNSORTED
    #comparisons := arraySize
    while (status = UNSORTED)
        #comparisons := #comparisons − 1
        status := SORTED
        for (i = 0 to #comparisons)
            if (array[i] > array[i+1])
                swap ith and (i + 1)th elements of the array
                status := UNSORTED
            end if
        end for
    end while
end bubble_sort
```

The CX register is used to keep track of the number of comparisons, and DX maintains the status information. The SI register points to the *i*th element of the input array.

The while loop condition is tested by lines 92–94. The for loop body corresponds to lines 79–90 and 96–101. The rest of the code follows the pseudocode. Note that the array pointer is available in the stack at BP+18 and its size at BP+20, as we use pusha to save all registers. Also note that this program uses only the 16-bit addressing modes.

Program 4.13 Bubble sort procedure

```
 1: COMMENT |         Bubble sort procedure     BBLSORT.ASM
 2:  Objective: To implement the bubble sort algorithm
 3:      Input: A set of non-zero integers to be sorted.
 4:     Input is terminated by entering zero.
 5: |          Output: Outputs the numbers in ascending order.
 6: CRLF       EQU   0DH,0AH
 7: MAX_SIZE   EQU   20
 8: .MODEL SMALL
 9: .STACK 100H
10: .DATA
11: array        DW  MAX_SIZE DUP (?)  ; input array for integers
12: prompt_msg  DB  'Enter non-zero integers to be sorted.',CRLF
13:      DB  'Enter zero to terminate the input.',0
14: output_msg  DB  'Input numbers in ascending order:',0
15:
16: .CODE
17: .486
18: INCLUDE   io.mac
19: main  PROC
20:       .STARTUP
21:       PutStr  prompt_msg     ; request input numbers
22:       nwln
23:       mov     BX,OFFSET array  ; BX := array pointer
24:       mov     CX,MAX_SIZE    ; CX := array size
25:       sub     DX,DX          ; number count := 0
26: read_loop:
27:       GetInt  AX             ; read input number
28:       nwln
29:       cmp     AX,0           ; if the number is zero
30:       je      stop_reading   ; no more numbers to read
31:       mov     [BX],AX        ; copy the number into array
32:       add     BX,2           ; BX points to the next element
33:       inc     DX             ; increment number count
34:       loop    read_loop      ; reads a max. of MAX_SIZE numbers
```

```
stop_reading:
      push    DX              ; push array size onto stack
      push    OFFSET array    ; place array pointer on stack
      call    bubble_sort
      PutStr  output_msg      ; display sorted input numbers
      nwln
      mov     BX,OFFSET array
      mov     CX,DX           ; CX := number count
print_loop:
      PutInt  [BX]
      nwln
      add     BX,2
      loop    print_loop
done:
      .EXIT
main  ENDP
;-----------------------------------------------------------
;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     PROC
      pusha
      mov     BP,SP

      ;CX serves the same purpose as the end_index variable
      ; in the C procedure. CX keeps the number of comparisons
      ; to be done in each pass. Note that CX is decremented
      ; by 1 after each pass.
      mov     CX, [BP+20]  ; load array size into CX
      mov     BX, [BP+18]  ; load array address into BX

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

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

      ;SI points to element X and SI+2 to the next element
      mov     SI,BX        ; load array address into SI
```

```
pass:
      ;This loop represents one pass of the algorithm.
      ;Each iteration compares elements at [SI] and [SI+2]
      ; and swaps them if ([SI]) < ([SI+2]).
      mov     AX,[SI]
      cmp     AX,[SI+2]
      jg      swap
increment:
      ;Increment SI by 2 to point to the next element
      add     SI,2
      dec     DI
      jnz     pass

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

swap:
      ; swap elements at [SI] and [SI+2]
      xchg    AX,[SI+2]
      mov     [SI],AX
      mov     DX,UNSORTED     ; set status to UNSORTED
      jmp     increment

sort_done:
      popa
      ret     4               ; return and clear parameters
bubble_sort    ENDP
      END    main
```

4.9 Handling a Variable Number of Parameters

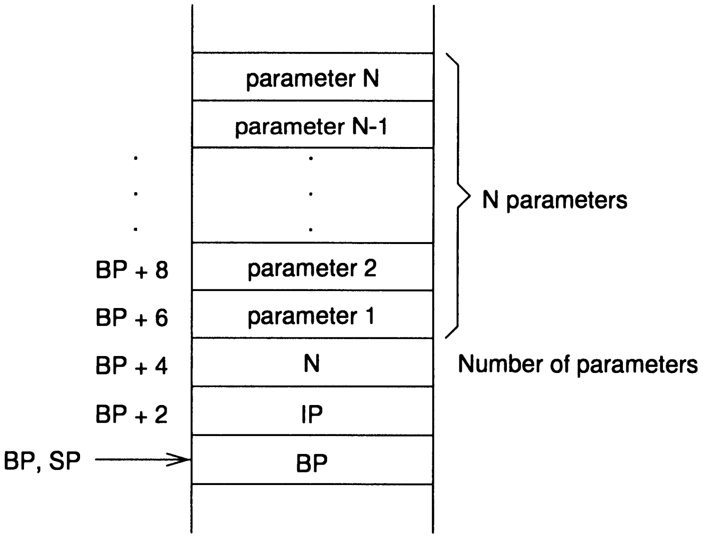
In the C language, some procedures can accept a variable number of parameters. The input and output functions `scanf` and `printf` are the two common procedures with a variable number of parameters. In this case, the called procedure does not know the number of parameters passed onto it. Usually, the first parameter in the parameter list specifies the number of parameters passed. This parameter should be pushed onto the stack last so that it is just below the return address independent of the number of parameters passed.

In assembly language procedures, a variable number of parameters can be easily handled by the stack method of parameter passing. Only the stack size imposes a limit on the number of parameters that can be passed. The next example illustrates the use of the stack in passing a variable number of parameters in assembly language programs.

Example 4.6 *Passing a variable number of parameters via the stack*

In this example, the procedure `variable_sum` receives a variable number of integers via the stack. The actual number of integers passed is the last parameter pushed onto the stack before calling the procedure. The procedure finds the sum of the integers and returns this value in the AX register.

The `main` procedure in Program 4.14 requests input from the user. Only nonzero numbers are accepted as valid input (entering a zero terminates the input). The `read_number` loop (lines 25–32) reads input numbers using `GetInt` and pushes them onto the stack. The CX register keeps a count of the number of input values, which is passed as the last parameter (line 34) before calling the `variable_sum` procedure. The state of the stack at line 56 (after pushing BP) is shown below:



The `variable_sum` procedure first reads the number of parameters passed onto it from the stack at BP+4 into the CX register. The `add_loop` (lines 63–66) successively reads each integer from the stack and computes their sum in AX. Note that, on line 64, we used a segment override prefix. If we write

```
add     AX,[BX]
```

the contents of BX is 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 BX is relative to SS (and not DS). The segment override prefixes—CS:, DS:, ES:, FS:, GS:, and SS:—can be placed in front of a memory operand to indicate accessing a segment other than the default segment.

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

```
        add     BP,6
        sub     AX,AX
add_loop:
        add     AX,[BP]
        add     BP,2
        loop    add_loop
```

can replace the code at lines 60–66. A disadvantage of this modified code is that, since we have modified BP, we no longer can access, for example, the parameter count value in the stack. For this example, however, this method works fine. A better way is to use an index register to represent the offset relative to BP. We defer this discussion until Chapter 5, which discusses the addressing modes of Pentium.

Another interesting feature is that the parameter space on the stack is cleared by main. Since a variable number of parameters is passed, we cannot use ret to clear the parameter space. This is done in main by lines 37–39. The CX is first incremented to include the parameter count (line 37). The byte count of the parameter space is computed on line 38. This value is added to SP to clear the parameter space (line 39).

Program 4.14 Program to illustrate passing a variable number of parameters

```
 1: TITLE   Variable # of parameters passed via stack   VARPARA.ASM
 2: COMMENT |
 3:         Objective: To show how variable number of parameters
 4:                    can be passed via the stack
 5:             Input: Requests variable number of non-zero integers.
 6:                    A zero terminates the input.
 7: |          Output: Outputs the sum of input numbers.
 8: CRLF    EQU    0DH,0AH    ; carriage return and line feed
 9: .MODEL SMALL
10: .STACK 100H
11: .DATA
```

```
prompt_msg  DB  'Please input a set of non-zero integers.',CRLF
            DB  'You must enter at least one integer.',CRLF
            DB  'Enter zero to terminate the input.',0
sum_msg     DB  'The sum of the input numbers is: ',0

.CODE
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg     ; request input numbers
      nwln
      sub     CX,CX          ; CX keeps number count
read_number:
      GetInt  AX             ; read input number
      nwln
      cmp     AX,0           ; if the number is zero
      je      stop_reading   ; no more numbers to read
      push    AX             ; place the number on stack
      inc     CX             ; increment number count
      jmp     read_number
stop_reading:
      push    CX             ; place number count on stack
      call    variable_sum   ; returns sum in AX
      ; clear parameter space on the stack
      inc     CX             ; increment CX to include count
      add     CX,CX          ; CX := CX * 2 (space in bytes)
      add     SP,CX          ; update SP to clear parameter
                             ; space on the stack
      PutStr  sum_msg        ; display the sum
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP

;-----------------------------------------------------------
;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 AX.
;-----------------------------------------------------------
variable_sum  PROC
      push    BP             ; save BP - procedure uses BP
      mov     BP,SP          ; copy SP to BP
```

```
        push    BX              ; save BX and CX
        push    CX

        mov     CX,[BP+4]       ; CX := # of integers to be added
        mov     BX,BP
        add     BX,6            ; BX := pointer to first number
        sub     AX,AX           ; sum := 0
add_loop:
        add     AX,SS:[BX]      ; sum := sum + next number
        add     BX,2            ; BX points to the next integer
        loop    add_loop        ; repeat count in CX

        pop     CX              ; restore registers
        pop     BX
        pop     BP
        ret                     ; parameter space cleared by main
variable_sum    ENDP
        END     main
```

4.10 Local Variables

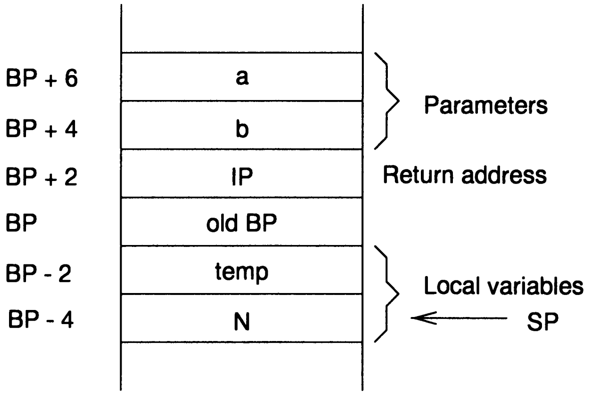
So far in our discussion we have not considered how local variables can be used in a procedure. To focus our discussion, consider the following C code.

```
int compute(int a, int b)
{
   int   temp, N;
         . . .
         . . .
}
```

The variables `temp` and `N` are local variables that come into existence when the procedure `compute` is invoked and disappear when the procedure terminates. Thus, these local variables are dynamic. We could reserve space for the local variables in a data segment. However, such space allocation is not desirable for two reasons:

1. Space allocation done in the data segment is static and remains active even when the procedure is not.
2. More importantly, is does not work with recursive procedures (i.e., procedures that call themselves).

For these reasons, space for local variables is reserved on the stack. For the C function, the stack may look like:



The information stored in the stack—parameters, return address, old BP value, and local variables—is collectively called the *stack frame*. In high-level languages, it is also referred to as the *activation record* (because each procedure activation requires all this information). The BP value is referred to as the *frame pointer* (FP). Once the BP value is known, we can access all items of data present in the stack frame. For example, parameters a and b can be accessed at BP+6 and BP+4, respectively. Local variables temp and N, for example, can be accessed at BP−2 and BP−4, respectively.

To aid program readability, we can use the EQU directive to name the stack locations. Thus, we can write

```
mov     BX,a
mov     temp,AX
```

instead of

```
mov     BX,[BP+6]
mov     [BP-2],AX
```

after establishing temp and a labels by using the EQU directive, as shown below.

```
a       EQU   WORD PTR [BP+6]
temp    EQU   WORD PTR [BP-2]
```

We will now look at two examples—both compute Fibonacci numbers. However, one example uses registers for local variables, while the other uses the stack. Section 4.12 compares the performance of these two versions.

Example 4.7 *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,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$$

is the Fibonacci sequence of numbers.

In this and the next example, we will write a procedure to compute the largest Fibonacci number that is less than or equal to a given input number. The `main` procedure requests this number and passes it on to the `fibonacci` procedure.

The `fibonacci` procedure keeps the last two Fibonacci numbers in local variables. These are mapped to registers AX and BX. The higher of the two Fibonacci numbers is kept in BX. The `fib_loop` successively computes the Fibonacci number until it is greater than or equal to the input number. Then, the Fibonacci number in AX is returned to the `main` procedure.

Program 4.15 Fibonacci number computation with local variables mapped to registers

```
TITLE   Fibonacci numbers (register version)    PROCFIB1.ASM
COMMENT |
        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.

.MODEL SMALL
.STACK 100H
.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
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg     ; request input number
      GetInt  DX             ; DX := input number
      nwln
      call    fibonacci
      PutStr  output_msg1    ; display Fibonacci number
      PutInt  DX
      PutStr  output_msg2
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP

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

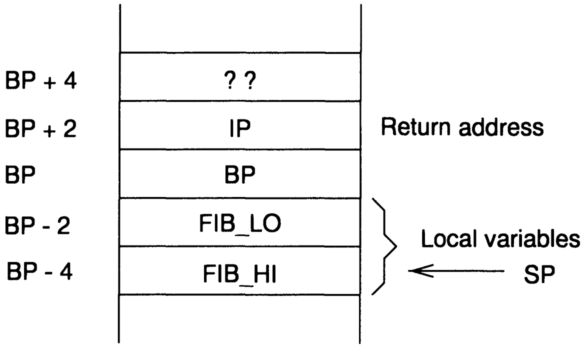
Example 4.8 *Fibonacci number computation using the stack for local variables*

In this example, we use the stack for storing the two Fibonacci numbers. The variable `fib_lo` corresponds to fib($n-1$), and `fib_hi` to fib(n).

The code

```
push   BP
mov    BP,SP
sub    SP,4
```

saves the BP value and copies the SP value into BP as usual. It also decrements SP by 4, thus creating 4 bytes of storage space for the two local variables `fib_lo` and `fib_hi`. At this point, the stack allocation is:



The two local variables can be accessed at BP−2 and BP−4. The two EQU statements, on lines 39 and 40, conveniently establish labels to these two locations. The space allocated for the local variables is cleared by

```
mov    SP,BP
```

on line 60. The rest of the code follows the logic of Example 4.7.

Program 4.16 Fibonacci number computation with local variables mapped to the stack

```
1: TITLE   Fibonacci numbers (stack version)    PROCFIB2.ASM
2: COMMENT |
3:         Objective: To compute Fibonacci number using the stack
4:                    for local variables.
5:             Input: Requests a positive integer from the user.
6:            Output: Outputs the largest Fibonacci number that
```

```
|                   is less than or equal to the input number.
.MODEL SMALL
.STACK 100H
.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
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  prompt_msg     ; request input number
      GetInt  DX             ; DX := input number
      nwln
      call    fibonacci
      PutStr  output_msg1    ; print Fibonacci number
      PutInt  DX
      PutStr  output_msg2
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP

;-----------------------------------------------------------
;Procedure fibonacci receives an integer in DX and computes
; the largest Fibonacci number that is less than the input
; number. The Fibonacci number is returned in AX.
;-----------------------------------------------------------
FIB_LO   EQU  WORD PTR [BP-2]
FIB_HI   EQU  WORD PTR [BP-4]
fibonacci  PROC
      push    BP
      mov     BP,SP
      sub     SP,4           ; space for local variables
      push    BX
      ; 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     AX,FIB_HI      ; compute next Fibonacci number
      mov     BX,FIB_LO
      add     BX,AX
      mov     FIB_LO,AX
      mov     FIB_HI,BX
      cmp     BX,DX          ; compare with input number in DX
      jle     fib_loop       ; if not greater, find next number
      ; AX contains the required Fibonacci number
      pop     BX
      mov     SP,BP          ; clear local variable space
      pop     BP
      ret
fibonacci  ENDP
      END     main
```

4.11 Multiple Source Program Modules

In the program examples we have seen so far, the entire assembly language source 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 massive source programs 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 short files.
- It is safer to edit a short file; any unintended modifications to the source file are limited to a single short file.

If we want to separately assemble modules, we have to precisely specify the intermodule interface. For example, if a procedure is called in the current module but is defined in another module, we have to state that fact so that the assembler will not flag such procedure calls as errors. Assemblers provide two directives—PUBLIC and EXTRN—to facilitate separate assembly of source modules. These two directives are discussed in the following sections. A simple example follows this discussion.

4.11.1 PUBLIC Directive

The PUBLIC directive makes the associated label(s) public and therefore available for other modules of the program. The format is

```
PUBLIC    label1, label2, ...
```

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

```
                    . . .
PUBLIC  error_msg, total, sample
                    . . .
.DATA
error_msg      DB   ''Out of range!'',0
total          DW   0
                    . . .
.CODE
                    . . .
sample   PROC
                    . . .
sample   ENDP
                    . . .
```

Note that when you make a label public, it is not necessary to specify the type of label!

4.11.2 EXTRN Directive

The EXTRN directive can be used to tell the assembler that certain labels are not defined in the current source file (i.e., module), but are defined in other modules. Thus the assembler leaves "holes" in the corresponding `.obj` file that the linker will fill in later on. The format is

```
EXTRN    label:type
```

where `label` is a label that is made public by a PUBLIC directive in some other module. The `type` specifies the type of label, some of which are listed below:

Type	Description
UNKNOWN	Undetermined or unknown type
BYTE	Data variable (size is 8 bits)
WORD	Data variable (size is 16 bits)
DWORD	Data variable (size is 32 bits)
QWORD	Data variable (size is 64 bits)
FWORD	Data variable (size is 6 bytes)
TBYTE	Data variable (size is 10 bytes)
PROC	A procedure name (Near or Far according to .MODEL)
NEAR	A near procedure name
FAR	A far procedure name

The PROC type should be used for procedure names if simplified segment directives (.MODEL, .STACK, ...) are used. In this case, appropriate procedure type is automatically included. For example, when the .MODEL is SMALL, the PROC type defaults to NEAR type. Assuming the labels `error_msg`, `total`, and `sample` are made public, as in the last example, the following example code makes them available in the current module.

```
.MODEL SMALL
          . . .
EXTRN     error_msg:BYTE, total:WORD
EXTRN     sample:PROC
          . . .
```

Note that the directive is spelled EXTRN (not EXTERN)!

Example 4.9 *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 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 4.17. Notice that on line 16, we declare `string_length` as an externally defined procedure by using the EXTRN directive.

Program 4.17 The `main` procedure defined in `mudule1.asm` calls the `sum` procedure defined in `module2.asm`

```
TITLE   Multimodule program for string length   MODULE1.ASM
COMMENT |
        Objective: To show parameter passing via registers
            Input: Requests two integers from keyboard.
|          Output: Outputs the sum of the input integers.
BUF_SIZE  EQU  41   ; string buffer size
.MODEL SMALL
.STACK 100H
.DATA
prompt_msg   DB   'Please input a string: ',0
length_msg   DB   'String length is: ',0
string1      DB   BUF_SIZE DUP (?)

.CODE
INCLUDE io.mac
EXTRN   string_length:PROC
main  PROC
      .STARTUP
      PutStr  prompt_msg     ; request a string
      GetStr  string1,BUF_SIZE  ; read string input
      nwln
      mov     BX,OFFSET string1 ; BX := string pointer
      call    string_length  ; returns string length in AX
      PutStr  length_msg     ; display string length
      PutInt  AX
      nwln
done:
      .EXIT
main  ENDP
      END     main
```

Program 4.18 This module defines the sum procedure called by `main`

```
TITLE         String length procedure         MODULE2.ASM
COMMENT |
        Objective: To write a procedure to compute string
                   length of a NULL terminated string.
            Input: String pointer in BX register.
|          Output: Returns string length in AX.
.MODEL SMALL
.CODE
```

```
 9: PUBLIC string_length
10: string_length PROC
11:       ; all registers except AX are preserved
12:       push    SI                ; save SI
13:       mov     SI,BX             ; SI := string pointer
14: repeat:
15:       cmp     BYTE PTR [SI],0   ; is it NULL?
16:       je      done              ; if so, done
17:       inc     SI                ; else, move to next character
18:       jmp     repeat            ;       and repeat
19: done:
20:       sub     SI,BX             ; compute string length
21:       mov     AX,SI             ; return string length in AX
22:       pop     SI                ; restore SI
23:       ret
24: string_length ENDP
25:       END
```

Program 4.18 shows a listing of module2.asm. This module consists of a single procedure. By using the PUBLIC directive, we make this procedure public (line 9) so that other modules can use this procedure. The string_length procedure receives a pointer to a NULL-terminated string in BX and returns the length of the string in AX. The procedure preserves all registers except for AX. Note that the END statement (last statement) of this module does not have a label, as this is not the procedure that begins the program execution.

We can assemble each source code module separately producing the corresponding .obj files. We can then link the .obj files together to produce a single .exe file. For example, using Turbo Assembler, the following sequence of commands will produce the executable file.

```
TASM    module1    ← Produces module1.obj
TASM    module2    ← Produces module2.obj
TLINK   module1+module2+io    ← Produces module1.exe
```

TASM, by default, assumes the .asm extension and TLINK assumes the .obj extension. The above sequence assumes that you have io.obj in your current directory. If you are using Microsoft Assembler, replace TASM with MASM and TLINK with LINK.

4.12 Performance: Procedure Overheads

As we have seen, 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 will quantify these overheads using the bubble sort and Fibonacci examples.

4.12.1 Stack Versus Registers

Figure 4.5 shows the performance of the assembly language bubble sort procedure (AL-original line) discussed in this chapter, as well as in Chapter 1. In this procedure, which does not use a separate swap procedure, swapping is done by

```
; AX contains the element pointed to by SI
xchg   AX,[SI+2]
mov    [SI],AX
```

where SI and SI+2 point to the elements of the array to be interchanged. This code requires a total of three memory accesses (two for the xchg and one for the mov instruction). It is straightforward to see that exchange of two values in memory requires four memory accesses (two reads and two writes). Here we are accomplishing the exchange using only three memory accesses by exploiting the fact that one of the elements is in the AX register. The code does not preserve the contents of the AX register. This is fine here because AX does not have anything useful. However, when we write a general swap procedure, it is good programming practice to preserve the registers. Let us, then, suppose that we want to preserve the contents of AX. Then we have to modify the two lines of code as

```
xchg   AX,[SI+2]
xchg   AX,[SI]
xchg   AX,[SI+2]
```

The above code interchanges the two elements while preserving the contents of AX. The code, however, requires six memory accesses. Thus, compared to the original code, the modified code requires three additional memory accesses. The performance impact of the additional memory accesses is significant, as shown in Figure 4.5 (AL-modified line).

The two versions that we have discussed so far have not used a separate swap procedure. When a procedure is used to swap two elements, performance deteriorates further (see AL-register and AL-stack lines). The AL-register and AL-stack lines represent the performance of the assembly language version in which registers and stack are used for parameter passing, respectively. The performance difference between AL-modified and AL-register lines closely approximates the call/return overhead associated with the swap procedure call.

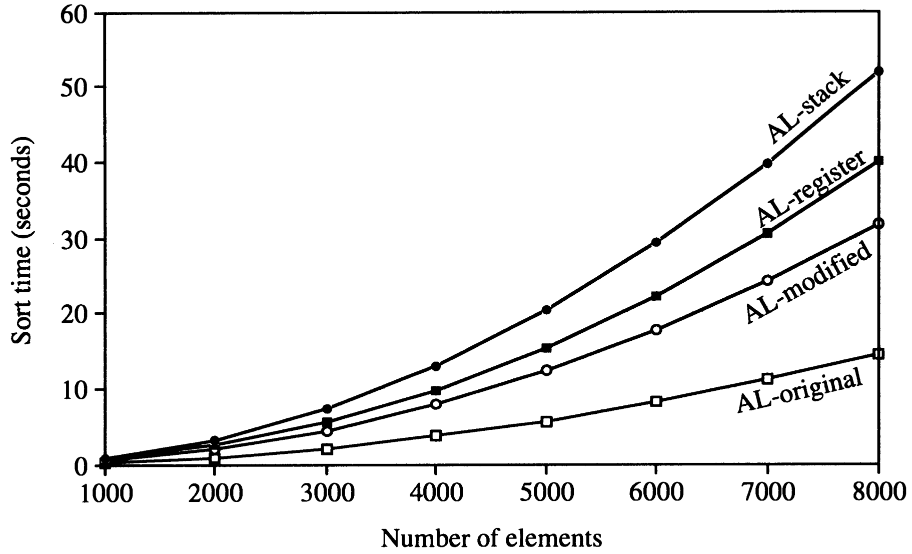


Figure 4.5 Performance of the four assembly language versions of the bubble sort.

The performance difference between the AL-stack and AL-register quantifies the additional overhead to pass parameters via the stack.

4.12.2 Comparison of C and Assembly Language Versions

The question that naturally arises is: How does the assembly language version compare with the C version? Figure 4.6 shows the performance of the C and assembly language versions of the bubble sort. All three versions use a swap procedure to exchange two elements. The AL-register line represents performance of an optimized version of the code used in Figure 4.5. In this version, the contents of the AX register are not preserved. This factor alone contributes significant improvement in the execution time (compare AL-register lines in Figures 4.5 and 4.6).

The stack version of the assembly language code is also optimized by not preserving the AX register. The stack code is further optimized by using the ESP register to access the parameters from the stack. The AL-stack line represents performance of the assembly language program that uses the following swap procedure:

```
swap_proc    PROC
```

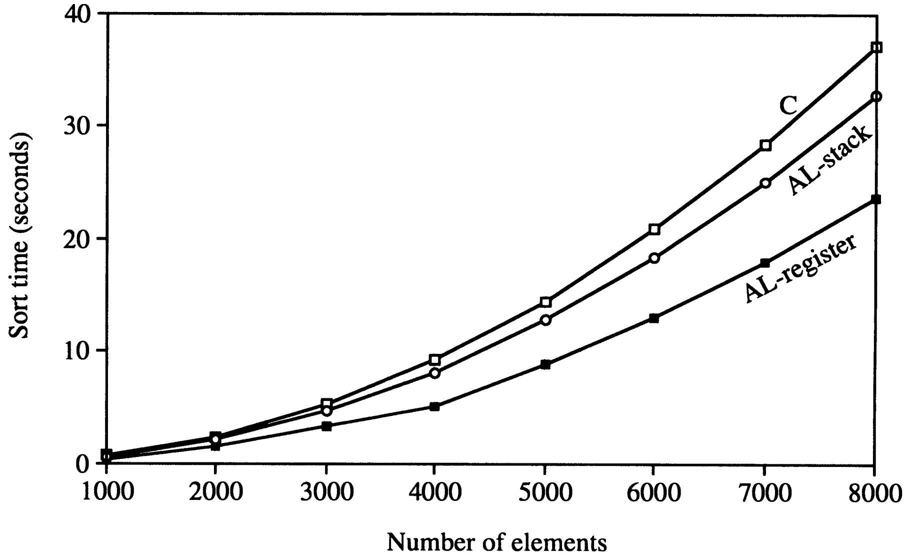


Figure 4.6 Performance comparison of C and assembly language versions of the bubble sort.

```
          push    SI
          push    DI
          mov     SI,[ESP+6]
          mov     DI,[ESP+8]
          mov     AX,[DI]
          xchg    AX,[SI]
          mov     [DI],AX
          pop     DI
          pop     SI
          ret     4
swap_proc         ENDP
```

Note that to use the stack pointer register to access the parameters, we have to use a 32-bit addressing mode (i.e., cannot use the SP register). It is also interesting to note that if we use the BP register to access the parameters as we have done in several examples, the performance is similar to that of the C version.

As you can see from the data presented in Figures 4.5 and 4.6 (compare the AL-register and AL-stack lines of Figure 4.5 with the C line of Figure 4.6), if

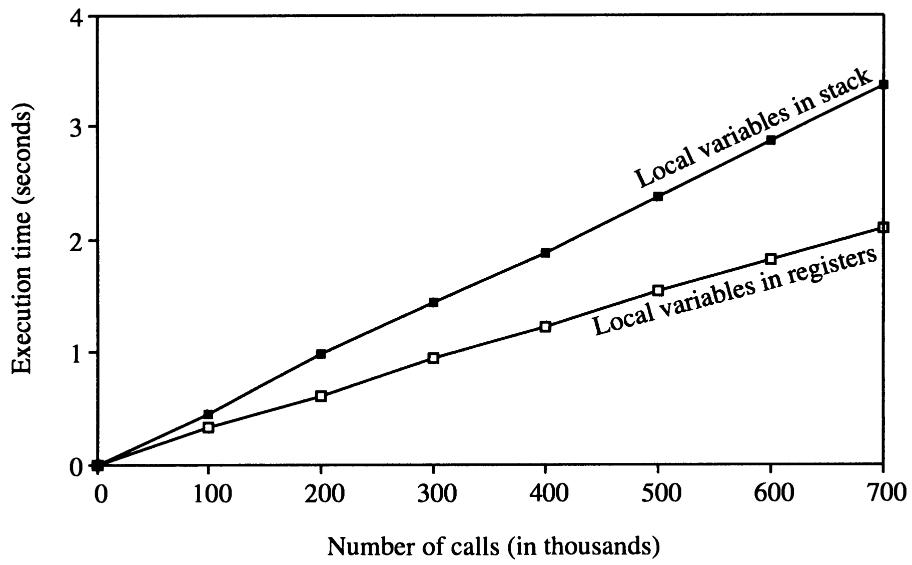


Figure 4.7 Local variable overhead: registers versus stack.

we don't write assembly language programs carefully, we may end up with a program that is worse than its high-level language counterpart!

4.12.3 Local Variable Overhead

Last, 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 4.15 and 4.16 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 4.7. The x-axis represents the number of calls to the procedure (varied from 1 to 700,000). The execution time increases by approximately 60 percent when the local variable storage is moved from the registers to the stack. Because of this performance impact, compilers always try to keep the local variables that are most frequently accessed by a procedure in registers.

4.13 Summary

The stack is a last-in-first-out (LIFO) 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 stack segment is used for implementing the stack. The top-of-stack is represented by SS:SP. In the Pentium implementation, the stack grows toward lower memory addresses (i.e., grows downward).

The stack serves three main purposes: temporary storage of data, transfer of control during a procedure call and return, and parameter passing.

When writing procedures in assembly language, parameter passing has to be explicitly handled. Parameter passing can be done by using registers or via the stack. While the register method is efficient, the stack-based method is more general. Also, when parameters are passed via the stack, it is straightforward to handle a variable number of parameters. Using the bubble sort example, we demonstrated the various overheads associated with procedure invocations and return and parameter passing.

As with parameter passing, local variables of a procedure can be stored either in registers or in the stack. Due to the limited number of registers available, only a few local variables can be mapped to registers. While stack avoids this limitation of the registers, it is slow. We demonstrated using the Fibonacci examples the advantage of mapping local variables to registers.

Real application programs are unlikely to be short to keep them in a single file. It is advantageous to break large source programs into more manageable chunks of code and keep them in several files (i.e., modules) rather than in one large file. We discussed how such multimodule programs can be written and assembled into a single executable file.

4.14 Exercises

4–1 What are the defining characteristics of a stack?

4–2 Discuss the differences between a queue and a stack.

4–3 What is top-of-stack? How is it represented in Pentium?

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

4–5 What is stack overflow? Which stack operation can cause stack overflow?

4–6 What are the main uses of the stack?

4–7 Can we invoke a procedure through the call instruction without the presence of a stack segment? Explain.

4–8 What is the main difference between a near procedure and a far procedure?

4–9 What are the two most common methods of parameter passing? Identify circumstances under which each method is preferred.

4–10 What are the disadvantages of passing parameters via the stack?

4–11 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.

4–12 In passing a variable number of parameters via the stack, why is it necessary to push the parameter count last?

4–13 Why are local variables of a procedure not mapped to a data segment?

4–14 How is storage space for local variables created in the stack?

4–15 A swap procedure can exchange two elements (pointed to by SI and DI) of an array using

```
xchg   AX,[DI]
xchg   AX,[SI]
xchg   AX,[DI]
```

The above code preserves the contents of the AX register. This code requires six memory accesses. Can we do better than this in terms of the number of memory accesses if we save and restore AX using `push` and `pop` stack operations?

4–16 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.

4–17 Verify that the following procedure is equivalent to the `string_length` procedure given in Section 4.11. Which procedure is better and why?

```
string_length1  PROC
        push    BX
        sub     AX,AX
repeat:
        cmp     BYTE PTR [BX],0
        je      done
        inc     AX
        inc     BX
        jmp     repeat
done:
        pop     BX
        ret
string_length1  ENDP
```

4.15 Progamming Exercises

4–P1 Write an assembly language program that reads a set of integers from the keyboard and displays their sum on the screen. Your program should read up to twenty integers (except zero) from the user. The input can be terminated by entering a zero or by entering twenty 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.

4–P2 Write a procedure `max` that receives three integers from `main` and returns the maximum of the three in AX. The `main` procedure requests the three integers from the user and displays the maximum number returned by the `max` procedure.

4–P3 Extend the last exercise to return both maximum and minimum of the three integers received by your procedure `minmax`. In order to return the minimum and maximum values, your procedure `minmax` also receives two pointers from `main` to variables `min_int` and `max_int`.

4–P4 Extend the last exercise to handle a variable number of integers passed 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 two values returned by the procedure are displayed by `main`.

4–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.

4–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 from the user 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.

4–P7 Write a procedure that receives a string via the stack (i.e., 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.

4–P8 Write a procedure that receives a string via the stack (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 passed is (␣ indicates a blank character)

␣ ␣ ␣ ␣ ␣Read␣ ␣ ␣my␣ ␣ ␣ ␣ ␣lips.

it will be modified by removing all leading and duplicate blanks, as

Read␣my␣lips.

4–P9 Write a program to read a number (consisting of up to 28 digits) and display the sum of the individual digits. Do not use `GetInt` to read the input number—read it as a sequence of characters. A sample input and output of the program is:

input: `123456789`
output: `45`

4–P10 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.

4–P11 Modify the last exercise to work on an input that can contain multiple spaces between the names. Also, display the name as in the last exercise but with the last name in all capital letters.

Chapter 5

Addressing Modes

Objectives

- To discuss in detail the various addressing modes supported by Pentium
- To illustrate the usefulness of these addressing modes in supporting high-level language features
- To describe how arrays are implemented and manipulated in assembly language
- To demonstrate the effectiveness of the advanced addressing modes

In assembly language, specification of data required by instructions can be done in a variety of ways. In Chapter 3 we discussed four different ways of specifying the operands. These are the register, immediate, and direct and indirect addressing modes. The last two addressing modes specify operands in memory. However, operands located in memory can be specified in several other ways. Section 5.1 describes the register and immediate addressing modes. Section 5.2 provides a detailed discussion of the various memory addressing modes supported by Pentium. Section 5.3 gives examples to illustrate the use of these addressing modes.

Arrays are important to organize a collection of related data. While one-dimensional arrays are straightforward to implement, multidimensional arrays are more involved. Section 5.4 discusses these issues in detail. Section 5.4.3 gives some examples to illustrate the use of addressing modes in processing one- and two-dimensional arrays.

Section 5.5 demonstrates the usefulness of the advanced addressing modes. The chapter ends with a summary.

S. P. Dandamudi, *Introduction to Assembly Language Programming*

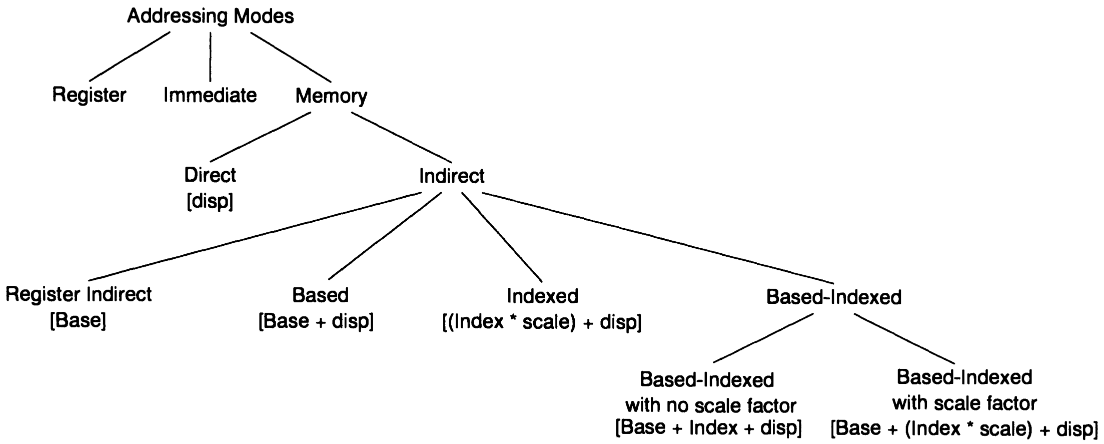


Figure 5.1 Addressing modes of Pentium for 32-bit addresses.

5.1 Simple Addressing Modes

The majority of assembly language instructions require specification of operands to be used as input data for the instruction and specification of the location where the result should be placed. The specification of the location of data is called the *data addressing mode*. Pentium provides several data addressing modes. A brief discussion of some basic addressing modes is given in Chapter 3. As discussed in Chapter 3, there are three fundamental addressing modes: register mode, immediate mode, and memory mode (see Figure 5.1).

Specification of operands located in memory can be done in a variety of ways, as shown in Figure 5.1. In previous chapters, we used only the direct and register indirect memory addressing modes to specify memory data. The remainder of this section discusses the register and immediate addressing modes. Memory addressing modes supported by Pentium are discussed in Section 5.2.

5.1.1 Register Addressing Mode

An instruction is said to be using the register addressing mode if both source and destination operands are located in the CPU registers. Here are some example instructions that use the register addressing mode:

```
mov     EAX,EDX
add     AL,CH
inc     BX
```

The register addressing mode is the most efficient way of specifying source and destination operands for two reasons:

- The operands are in the registers and no memory access is required.
- Instructions using the register mode tend to be shorter, as only 3 bits are needed to identify a register. In contrast, we need at least 16 bits to identify a memory location.

As a consequence, good compilers attempt to place frequently accessed data items in registers. As an example, consider the following pseudocode:

```
total := 0
for (i = 1 to 400)
    total = total + marks[i]
end for
```

Even if the compiler allocates memory for variables `i` and `total`, it should move these variables to registers for the duration of the `for` loop. At the end of the loop, the values from the assigned registers can be written back to the memory. This arrangement is more efficient than accessing variables `i` and `total` directly from memory during each iteration.

5.1.2 Immediate Addressing Mode

In this addressing mode, the operand is stored as part of the instruction. As suggested in our previous discussion (see Chapter 3), this mode imposes several restrictions:

- This addressing mode is typically used with instructions that require at least two operands to manipulate. There are exceptions, however! For instance, the `push` instruction, which takes only a single operand, allows specification of an immediate value.
- This addressing mode can only be used to specify the source operand.
- Another addressing mode is required to specify the destination operand.

Also, the immediate operand, which is stored along with the instruction, resides in the code segment—not in the data segment. Here are some examples:

```
mov     AL,55
mov     EDX,12345
```

This addressing mode is also faster to execute an instruction because the operand "immediately" follows the instruction in memory. Consequently, the operand is fetched into the instruction queue along with the instruction during the instruction fetch cycle. This prefetch, therefore, reduces the time required to get the operand from memory.

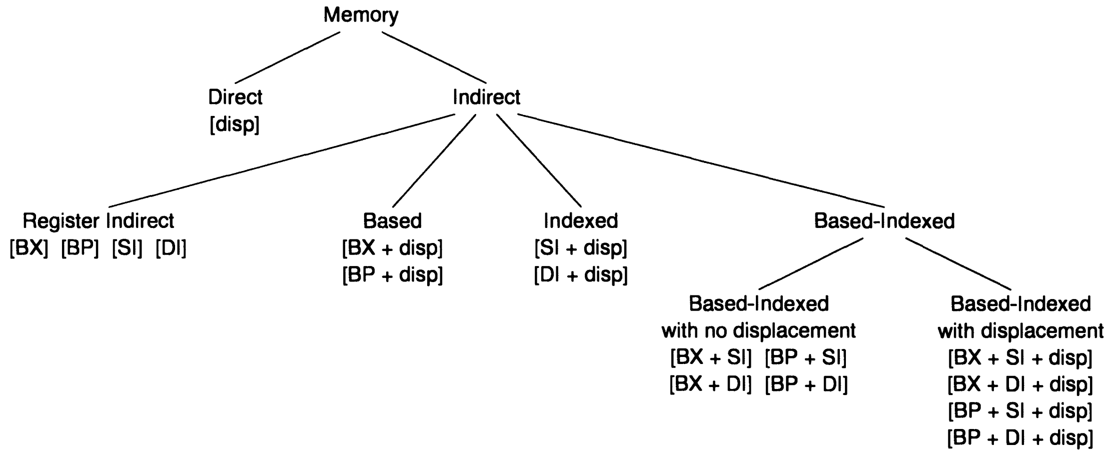


Figure 5.2 Memory addressing modes for 16-bit addresses.

5.2 Memory Addressing Modes

Pentium offers several addressing modes to access operands located in memory. 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). Address size of 16 bits support segments of 64 KB, while 4 GB segments can be used with 32-bit address size. The different memory addressing modes available for 16-bit address size are the same as those supported by the 8086. Figure 5.2 shows the default memory addressing modes available for 16-bit address size. Pentium supports a more flexible set of addressing modes for 32-bit addresses. These addressing modes are shown in Figure 5.1 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			

Table 5.1 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

The main differences between 16-bit and 32-bit addressing are summarized in Table 5.1. How does the processor know whether to use 16-bit or 32-bit addressing? It uses the D bit in the CS segment descriptor to determine if the address is 16 bits or 32 bits long. As discussed in Chapter 2, if the D bit is 0, the default size for operands as well as addresses is 16 bits. The D bit is 1 for 32 bit operands and addresses. It is, however, possible to override these defaults. Pentium provides two size override prefixes:

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 16-bit data and addresses. This, however, does not restrict us from using 32-bit data and addresses. For example, when we write

```
mov     AX,123
```

the assembler generates the following machine language code:

```
B8 007B
```

However, when we use a 32-bit operand, as in

```
mov     EAX,123
```

the following code is generated by the assembler:

```
66 | B8 0000007B
```

Notice that the operand size override prefix (66H) is automatically inserted by the assembler.

The greatest benefit of the address size override prefix is that we can use all the addressing modes provided for 32-bit addresses with 16-bit addresses. For instance, we can use a scale factor, as in the following example:

```
mov     AX,[EBX+ESI*2]
```

The assembler automatically inserts the address size override prefix (67H), as shown below:

```
67 | 8B 04 73
```

It is also possible to mix both override prefixes as demonstrated by the following example. The assembly language statement

```
mov     EAX,[EBX+ESI*2]
```

causes the assembler to insert both operand and address size override prefixes, as shown here:

```
66 | 67 | 8B 04 73
```

Remember that with 16-bit addresses, our segments are limited to 64 KB. Even though we have used 32-bit registers EBX and ESI in the last two examples, offsets into the segment are still limited to 64 KB (i.e., offset should be less than or equal to FFFFH). The processor generates a general protection fault if this value is exceeded. In summary, the address size prefix only allows us to use the additional addressing modes of Pentium with 16-bit addresses.

5.2.1 Direct Addressing

This is the simplest of the addressing modes available to access data from memory. Recall that accessing a data item that is located in the memory involves specifying the current data segment base address and an offset within the segment. The offset is often referred to as the *effective address*. By default, the DS register identifies the data segment. The various data addressing modes to access a memory data item specify the offset value of the data item.

In the direct addressing mode, effective address of a data item is specified as part of the instruction. The Pentium instruction set does not allow the specification of both operands in direct addressing mode. As a result, instructions in high-level languages involving more than one variable could require a sequence of assembly language instructions. For example, the following C code

```
total_marks = assign_marks + test_marks + exam_marks
```

could be translated by a compiler into a sequence of four directly addressed assembly language instructions:

```
mov     EAX,assign_marks
add     EAX,test_marks
add     EAX,exam_marks
mov     total_marks,EAX
```

Even though we are using variable names in the above examples, the assembler will actually replace these variables by their offset values during the assembly process.

In general, direct addressing can be used to access simple variables. The main drawback of this addressing mode is that it is not useful to access complex data structures like arrays and records that are permitted in high-level languages such as C.

5.2.2 Register Indirect Addressing

When an operand is specified by the register indirect addressing mode, the effective address of the operand is placed in a general-purpose register. For 16-bit segments, only BX, BP, SI, and DI registers are allowed to hold an effective address. For 32-bit segments, all eight 32-bit registers (i.e., EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP) can be used. This mode is called register indirect mode because the effective address of the operand is not found directly from the instruction but indirectly through a register. The fact that a register is holding the offset is indicated by enclosing it within square brackets [ ], as in

```
add     AX,[BX]
```

This is different from

```
add     AX,BX
```

which implies that the second operand is in BX. While 16-bit addressing restricts us to use only four 16-bit registers (i.e., BX, BP, SI, and DI), all eight 32-bit registers can be used by using the address size override prefix. For instance

```
mov     AX,[ECX]
```

is valid, but not

```
mov     AX,[CX]   ; not valid
```

Default Segments Referenced

The register indirect addressing mode can be used to specify data items that are located either in the data segment or in the stack segment.

16-bit Addresses: By default, effective address in registers BX, SI, or DI is taken as the offset value into the data segment (i.e., relative to the DS segment register). On the other hand, if the BP register is used, by default, the offset is used to access a data item from the stack segment (i.e., relative to the SS segment register). As we have seen in Chapter 4, BP is frequently used in the register indirect addressing mode to access parameter values from the stack in procedure calls.

32-bit Addresses: By default, effective address in registers EAX, EBX, ECX, EDX, ESI, and EDI is relative to the DS data segment. The SS stack segment is used if EBP and ESP registers are used.

In both cases, stack operations such as push and pop refer to the stack segment. In addition, the destination of string instructions uses the ES data segment register (see Chapter 9 for details on string instructions).

Overriding Default Segments

These default segment assignments can be overridden by a segment override prefix. For example,

```
add     AX,SS:[BX]
```

can be used to access a data item from the stack whose offset relative to the SS register is given in the BX register. In a similar manner, the BP register can be used as an offset into the data segment by

```
add     AX,DS:[BP]
```

The CS, ES, FS, and GS segment registers can also be used to override the default association, even though the CS register is not used frequently. (See Chapter 7 for an example that uses the CS overriding prefix.) To summarize, Pentium provides the following segment override prefixes:

2EH	CS segment override prefix
36H	SS segment override prefix
3EH	DS segment override prefix
26H	ES segment override prefix
64H	FS segment override prefix
65H	GS segment override prefix

You cannot use these override prefixes to affect the default segment association in the following cases:

- Destination of string instructions always uses the ES segment.

- Stack push and pop instructions always use the SS segment.
- Instruction fetches always use the CS segment.

The register indirect addressing mode is useful in accessing variables that contain more than one element such as an array. Furthermore, this addressing mode also provides a fundamental capability by which data can be accessed from a pointer passed on to a procedure. The remaining three addressing modes provide further refinements that are useful in supporting some high-level language features.

5.2.3 Based Addressing

In the based addressing mode, one of the registers (as described in the last subsection) acts as a 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 a part of the instruction. For 16-bit addresses, the signed displacement is either an 8-bit or a 16-bit number. For 32-bit addresses, it is either an 8-bit or a 32-bit number.

The same default segment associations discussed in the last subsection apply. Of course, a segment override prefix can be used to specify some other segment association.

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 spaces left in a particular course. We can let the BX register point to the base address of the corresponding course record and use displacement to read the number of students registered and the enrollment limit for the course to compute the desired answer. This is illustrated in Figure 5.3.

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

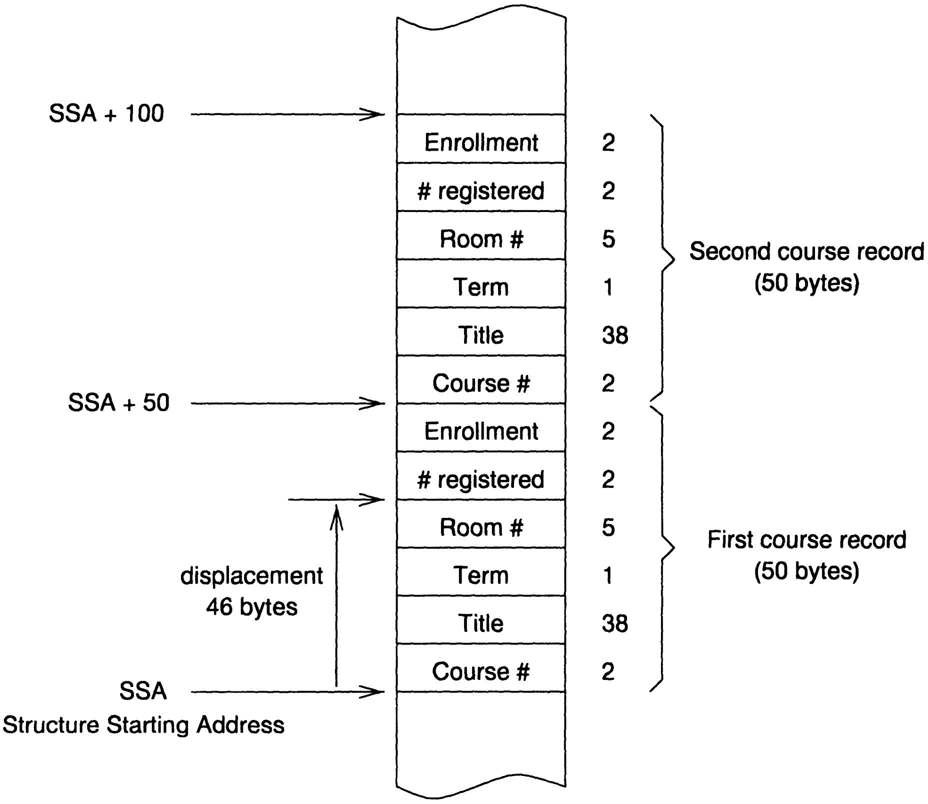


Figure 5.3 Course record layout in memory.

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.

5.2.4 Indexed Addressing

In this addressing mode, the effective address is computed as

(Index * scale factor) + signed displacement

For 16-bit addresses, no scaling factor is allowed (see Table 5.1 on page 177). For 32-bit addresses, a scale factor of 2, 4, or 8 can be specified. Of course, we can use a scale factor in the 16-bit addressing mode by using an address size override prefix.

The indexed addressing mode is often used to access elements of an array. The beginning of the array is given by the displacement, and the value of the index register selects an element within the array. The scale factor is particularly useful to access arrays of elements whose size is 2, 4, or 8 bytes.

The following are valid instructions using the indexed addressing mode to specify one of the operands.

```
add     AX,[DI+20]
mov     AX,marks_table[ESI*4]
add     AX,table1[SI]
```

In the second instruction, the assembler would supply a constant displacement that represents the offset of `marks_table` in the data segment. When using this notation, which is typical, you should note that ESI represents an index into the array. If no scale factor is used as in the last instruction, SI should hold a value that represents the difference in *bytes* between the beginning of the array and the offset of the element being accessed. For example, if `table1` is an array of integers, where each integer requires 2 bytes of storage, to refer to the tenth element, the SI register should have 18. When using a scale factor, we avoid such byte counting!

5.2.5 Based-Indexed Addressing

Based-Indexed with No Scale Factor

In this addressing mode, the effective address is computed as

Base + Index + signed displacement

The displacement can be a signed 8-bit or 16-bit number for 16-bit addresses; it can be a signed 8-bit 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, we can use this addressing mode to access arrays passed on to a procedure. In this case, the base register could point to the beginning of the array, while an index register can be used to store the offset to a specific element.

Assuming that BX points to `table1`, we can use the code

```
mov     AX,[BX+SI]
cmp     AX,[BX+SI+2]
```

to compare two successive elements of `table1`. This type of code is particularly useful if the `table1` pointer is passed as a parameter.

Based-Indexed with Scale Factor

In this addressing mode, the effective address is computed as

Base + (Index * scale factor) + signed displacement

This addressing mode provides an efficient indexing mechanism into a two-dimensional array when the element size is 2, 4, or 8 bytes.

5.3 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 data value in a sorted array. Example 5.1 uses only the 16-bit addressing modes (see Figure 5.2), while Example 5.2 uses both 16-bit and 32-bit addressing modes. The insertion sort procedure that uses 16-bit as well as 32-bit addressing modes is given in Section 5.5 (see Program 5.23 on page 200).

Example 5.1 *Sorting an integer array using the insertion sort*

This example requests a set of integers from the user and displays these numbers in sorted order. The main procedure reads a maximum of MAX_SIZE integers (lines 23–30). It accepts only non-negative numbers. Entering a negative number terminates the input (lines 26 and 27).

The main procedure passes the array size and pointer (lines 32–36) to the insertion sort procedure. The remainder of the main procedure displays the sorted array returned by the sort procedure. Note that the main procedure uses the indirect addressing mode on lines 28 and 43.

There are several sorting algorithms to sort an array of numbers. We have used the bubble sort algorithm in Chapter 1. Here we will use another sorting algorithm—the insertion sort. 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, 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 as in C.

```
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 5.19, follows the pseudocode.

Program 5.19 Insertion sort

```
TITLE    Sorting an array by insertion sort    INS_SORT.ASM
COMMENT |
        Objective: To sort an integer array using insertion sort.
            Input: Requests numbers to fill array.
|          Output: Displays sorted array.
.MODEL SMALL
.STACK 100H
.DATA
MAX_SIZE       EQU 100
array          DW  MAX_SIZE DUP (?)
input_prompt   DB  'Please enter input array: '
               DB  '(negative number terminates input)',0
out_msg        DB  'The sorted array is:',0

.CODE
.486
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  input_prompt ; request input array
        mov     BX,OFFSET array
        mov     CX,MAX_SIZE
array_loop:
        GetInt  AX           ; read an array number
        nwln
        cmp     AX,0         ; negative number?
        jl      exit_loop    ; if so, stop reading numbers
        mov     [BX],AX      ; otherwise, copy into array
        add     BX,2         ; increment array address
        loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
        mov     DX,BX        ; DX keeps the actual array size
        sub     DX,OFFSET array  ; DX := array size in bytes
        shr     DX,1         ; divide by 2 to get array size
        push    DX           ; push array size & array pointer
        push    OFFSET array
        call    insertion_sort
        PutStr  out_msg      ; display sorted array
        nwln
        mov     CX,DX
        mov     BX,OFFSET array
display_loop:
        PutInt  [BX]
        nwln
        add     BX,2
        loop    display_loop
done:
        .EXIT
main    ENDP

;-----------------------------------------------------------
; 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.
;-----------------------------------------------------------
SORT_ARRAY  EQU  [BX]
insertion_sort PROC
       pusha                 ; save registers
       mov     BP,SP
       mov     BX,[BP+18]    ; copy array pointer
       mov     CX,[BP+20]    ; copy array size
       mov     SI,2          ; array left of SI is sorted
for_loop:
       ; variables of the algorithm are mapped as follws:
       ; DX = temp, SI = i, and DI = j
       mov     DX,SORT_ARRAY[SI] ; temp := array[i]
       mov     DI,SI         ; j := i-1
       sub     DI,2
while_loop:
       cmp     DX,SORT_ARRAY[DI]  ; temp < array[j]
```

```
71:        jge     exit_while_loop
72:        ; array[j+1] := array[j]
73:        mov     AX,SORT_ARRAY[DI]
74:        mov     SORT_ARRAY[DI+2],AX
75:        sub     DI,2           ; j := j-1
76:        cmp     DI,0           ; j >= 0
77:        jge     while_loop
78: exit_while_loop:
79:        ; array[j+1] := temp
80:        mov     SORT_ARRAY[DI+2],DX
81:        add     SI,2           ; i := i+1
82:        dec     CX
83:        cmp     CX,1           ; if CX = 1, we are done
84:        jne     for_loop
85: sort_done:
86:        popa                   ; restore registers
87:        ret     4
88: insertion_sort ENDP
89:        END     main
```

Since the sort procedure does not return any value back to the main program in registers, we can use pusha (line 58) and popa (line 86) to save and restore registers. As pusha saves all eight 16-bit registers on the stack, the offset is appropriately adjusted to access the array size and array pointer parameters (lines 60 and 61).

The `while` loop is implemented by lines 69–78, while the `for` loop is implemented by lines 63–84. Note that the array pointer is copied to BX (line 60), and line 56 assigns a convenient label to this. We have used the based-indexed addressing mode on lines 66, 70, and 73 without any displacement and on lines 74 and 80 with displacement. Based addressing is used on lines 60 and 61 to access parameters from the stack.

Example 5.2 *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 searching 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 shown below.

```
binary_search (array, size, number)
    lower := 0
    upper := size − 1
    while (lower ≤ upper)
        middle := (lower + upper)/2
        if (number = array[middle])
        then
            return (middle)
        else
            if (number < array[middle])
            then
                upper := middle − 1
            else
                lower := middle + 1
            end if
        end if
    end while
    return (0)      {number not found}
    end binary_search
```

The listing of the binary search program is given in Program 5.20. The main procedure is similar to that in the last example. The lower and upper index variables are mapped to the AX and CX registers. The number to be searched is stored in DX and the array pointer is in the BX register. Register SI keeps the middle index value.

Program 5.20 Binary Search

```
TITLE   Binary search of a sorted integer array   BIN_SRCH.ASM
COMMENT |
        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.
.MODEL SMALL
.STACK 100H
.DATA
```

```
MAX_SIZE        EQU 100
array           DW  MAX_SIZE DUP (?)
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

.CODE
.486
INCLUDE io.mac
main    PROC
        .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
        nwln
        cmp     AX,0            ; negative number?
        jl      exit_loop       ; if so, stop reading numbers
        mov     array[ESI*2],AX ; otherwise, copy into array
        inc     SI           ; 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
        nwln
        push    AX           ; push number, size & array pointer
        push    SI
        push    OFFSET 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
        nwln
        cmp     AL,'Y'        ; if response is not 'Y'
        jne     read_input    ; repeat the loop
done:                         ; otherwise, terminate program
        .EXIT
main    ENDP

;-----------------------------------------------------------
; 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  PROC
       push    BP             ; save registers
       mov     BP,SP
       push    EBX
       push    ESI
       push    CX
       push    DX
       sub     EBX,EBX        ; EBX := 0
       mov     BX,[BP+4]      ; copy array pointer
       mov     CX,[BP+6]      ; copy array size
       mov     DX,[BP+8]      ; copy number to be searched
       sub     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
```

```
101:         jmp     while_loop
102: upper_half:
103:         inc     SI              ; middle := middle+1
104:         mov     AX,SI           ; lower := middle+1
105:         jmp     while_loop
106: end_while:
107:         sub     AX,AX           ; number not found (clear AX)
108:         jmp     skip1
109: search_done:
110:         inc     SI              ; position := index+1
111:         mov     AX,SI           ; return position
112: skip1:
113:         pop     DX              ; restore registers
114:         pop     CX
115:         pop     ESI
116:         pop     EBX
117:         pop     BP
118:         ret     6
119: binary_search  ENDP
120:         END     main
```

Since the binary search procedure returns a value in the AX register, we cannot use the pusha instruction as in the last example. This example also demonstrates how some of the 32-bit addressing modes can be used with 16-bit segments. For example, on line 95, we use a scale factor of 2 to convert the index value in SI to byte count. Also, a single comparison (line 95) 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 111).

5.4 Arrays

Arrays are useful in organizing a collection of related data items, such as test marks of a class, salaries of employees, etc. 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 5.4.2.

5.4.1 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]`.

In Pascal, however, we can specify a lower bound value for the subscript. An example declaration of test marks in Pascal is

```
test_marks: ARRAY[1..10] OF Integers;
```

Array declaration in high-level languages specifies the following four attributes:

- Name of the array (`test_marks`)
- Number of the elements (10)
- Element size (2 bytes)
- Index range (0–9 in C version, 1–10 in Pascal version)

From this information, the amount of storage space required for the array can be easily calculated. Storage space in bytes is given by

Storage space = number of elements * element size in bytes

In our example, it is equal to 10*2 = 20 bytes. In assembly language, arrays are implemented by allocating the required amount of storage space. For example, `test_marks` can be declared as

```
test_marks    DW    10 DUP (?)
```

An array name can be assigned to this storage space. But that is all the support you get in assembly language! It is up to you as a programmer to "properly" access the array taking into account the element size and the range of subscripts.

You need to know how the array is stored in memory in order to access elements of the array. For one-dimensional arrays, representation of array in memory is rather direct—array elements are stored linearly in the same order as shown in Figure 5.4. In the remainder of this section, we will use the convention used for arrays in C, i.e., subscripts are assumed to begin with 0.

To access an element of an array, you need to know its displacement value in bytes relative to the beginning of the array. Since you know the element size in bytes, it is rather straightforward to compute the displacement from the subscript value.

displacement = subscript * element size in bytes

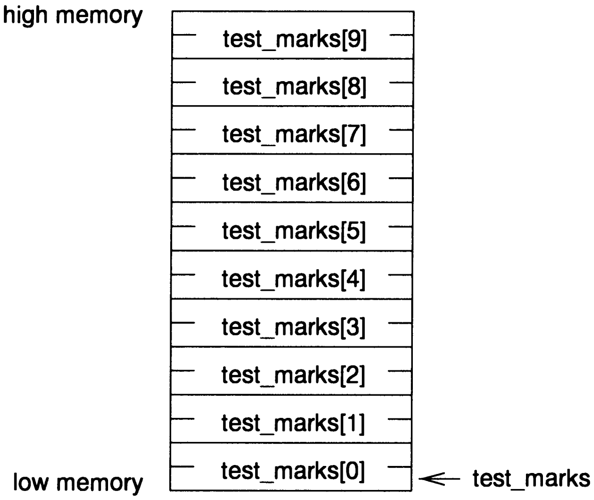


Figure 5.4 One-dimensional array storage representation.

For example, to access the sixth student's mark (i.e., subscript is 5), you have to use 5*2=10 as the displacement value into the `test_marks` array. Section 5.4.3 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 a scale factor to avoid computing displacement in bytes.

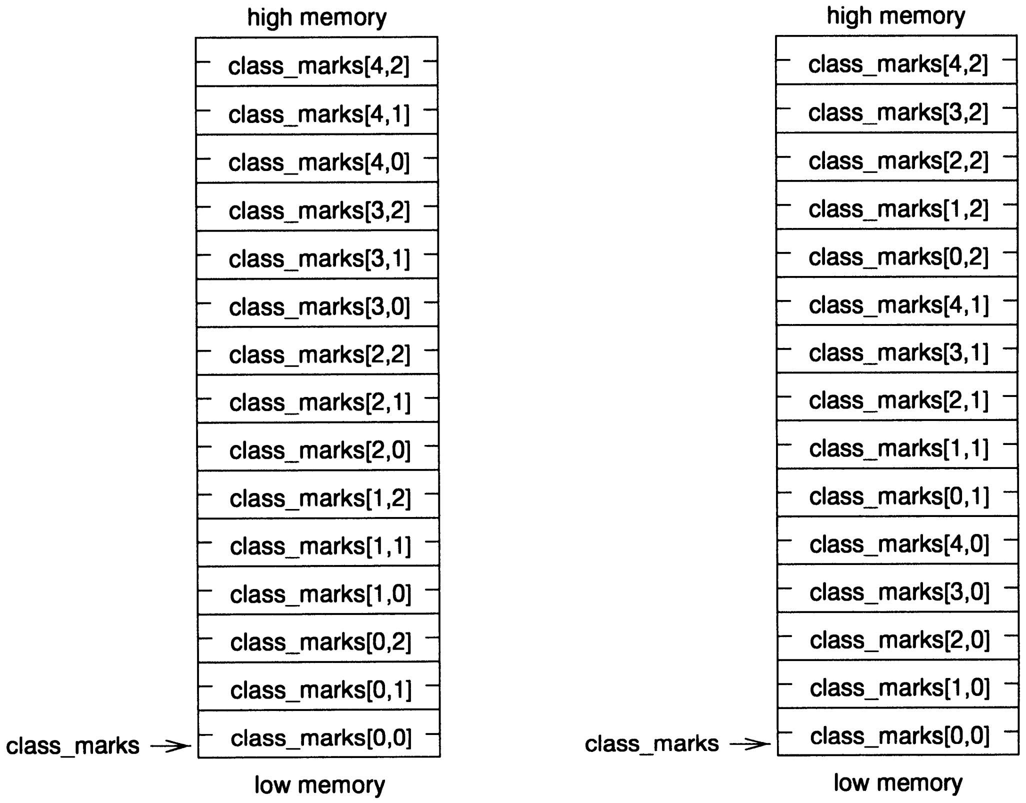
5.4.2 Multidimensional Arrays

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 fifty students getting three tests during a semester. For most programs, arrays of up to three dimensions are adequate. In this section, we will discuss how two-dimensional arrays are represented and manipulated in assembly language. Our discussion can be generalized to higher dimension arrays.

For example, a 5×3 array to store test marks can be declared in C as

```
int     class_marks[5][3];   /* 5 rows and 3 columns */
```

Storage representation of such arrays is not as direct as that for one-dimensional arrays. Since the memory is one-dimensional (i.e., linear array of bytes), we need to transform the two-dimensional structure to a one-dimensional structure. This transformation can be done in one of two common ways:



(a) Row-major order **(b) Column-major order**

Figure 5.5 Two-dimensional array storage representation.

- Order the elements of the array row by row starting with the first row
- Order the elements of the array column by column starting with the first column.

The first method, called the *row-major ordering*, is shown in Figure 5.5*a*. Row-major ordering is used in most high-level languages including C and Pascal. The other method, called the *column-major ordering*, is shown in Figure 5.5*b*. Column-major ordering is used in Fortran. In the remainder of this section, we focus on the row-major ordering scheme.

Why do we need to know the underlying storage representation? When you are using a high-level language, you really do not have to bother about the storage representation of the arrays. Access to arrays is provided by subscripts—one subscript for each dimension of the array. However, when using assembly language, you need to know the storage representation in order to access individual elements of the array for reasons discussed next.

In assembly language, we can allocate storage space for the `class_marks` array as

```
class_marks    DW    5*3 DUP (?)
```

This statement simply allocates 30 bytes required to store the array. Now we need a formula to translate row and column subscripts to the corresponding displacement. In C language, which uses row-major ordering and subscripts start with zero, we can express displacement of an element at row *i* and column *j* as

$$\text{displacement} = (i * \text{COLUMNS} + j) * \text{ELEMENT\_SIZE}$$

where COLUMNS is the number of columns in the array and ELEMENT_SIZE is the number of bytes required to store an element. For example, the displacement of `class_marks[3,1]` is (3*3+1)*2 = 20. The next section gives some examples to illustrate how two-dimensional arrays are manipulated.

5.4.3 Examples of Arrays

This section presents two examples to illustrate array manipulation of one- and two-dimensional arrays. These examples also demonstrate the use of advanced addressing modes in accessing multidimensional arrays.

Example 5.3 *Finding the sum of a one-dimensional array*

This example shows how one-dimensional arrays can be manipulated. Program 5.21 finds the sum of the `test_marks` array and displays the result.

Program 5.21 The sum of a one-dimensional array

```
TITLE       Sum of a long integer array       ARAY_SUM.ASM
COMMENT |
        Objective: To find sum of all elements of an array.
            Input: None
|          Output: Displays the sum.
.MODEL SMALL
.STACK 100H
```

```
 8:  .DATA
 9:  test_marks      DD  90,50,70,94,81,40,67,55,60,73
10:  NO_STUDENTS     EQU ($-test_marks)/4     ; number of students
11:  sum_msg         DB  'The sum of test marks is: ',0
12:
13:  .CODE
14:  .486
15:  INCLUDE io.mac
16:  main    PROC
17:          .STARTUP
18:          mov     CX,NO_STUDENTS    ; loop iteration count
19:          sub     EAX,EAX           ; sum := 0
20:          sub     ESI,ESI           ; array index := 0
21:  add_loop:
22:          mov     EBX,test_marks[ESI*4]
23:          PutLInt EBX
24:          nwln
25:          add     EAX,test_marks[ESI*4]
26:          inc     ESI
27:          loop    add_loop
28:
29:          PutStr  sum_msg
30:          PutLInt EAX
31:          nwln
32:          .EXIT
33:  main    ENDP
34:          END     main
```

Each element of the `test_marks` array, declared on line 9, requires 4 bytes. The array size NO_STUDENTS is computed on line 10 using the predefined location counter symbol $. The predefined symbol $ is always set to the current offset in the segment. Thus, on line 10, $ points to the byte after the array storage space. Therefore, `($-test_marks)` gives the storage space in bytes and dividing this by four gives the number of elements in the array. We are using the indexed addressing mode on lines 22 and 25 where a scale factor of 4 is used. Remember that scale factor is only allowed in the 32-bit mode. As a result, we have to use ESI rather than the SI register.

Example 5.4 *Finding the sum of a column in a two-dimensional array*

Consider the `class_marks` array representing test scores of a class. For simplicity, assume that there are only five students in the class. Also, assume that the class is given three tests. As we have discussed before, we can use a 5×3 array to store the marks. Each row represents the three test marks of a student in the class. The first column represents the marks of the first test, the second column represents the marks of the second test, and so on. The objective of this example is to find the sum of the last test marks for the class. The program listing is given in Program 5.22.

Program 5.22 The sum of a column in a two-dimensional array

```
TITLE  Sum of a column in a 2-dimensional array  TEST_SUM.ASM
COMMENT |
        Objective: To demonstrate array index manipulation
                   in a two-dimensional array of integers.
            Input: None
|          Output: Displays the sum.
.MODEL SMALL
.STACK 100H
.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
.486
INCLUDE io.mac
main    PROC
        .STARTUP
        mov     CX,NO_ROWS    ; loop iteration count
        sub     AX,AX         ; sum := 0
        ; ESI := index of class_marks[0,2]
        sub     EBX,EBX
        mov     ESI,NO_COLUMNS-1
sum_loop:
        add     AX,class_marks[EBX+ESI*2]
        add     EBX,NO_ROW_BYTES
```

```
34:          loop    sum_loop
35:
36:          PutStr  sum_msg
37:          PutInt  AX
38:          nwln
39:  done:
40:          .EXIT
41:  main    ENDP
42:          END     main
```

To access individual test marks, we use based-indexed addressing with a displacement on line 32. Note that even though we have used

```
class_marks[EBX+ESI*2]
```

it is translated by the assembler as

```
[EBX+(ESI*2)+constant]
```

where the constant is the offset of class_marks. For this to work, EBX should store the offset of the row in which we are interested. For this reason, after initializing EBX to zero to point to the first row (line 29), NO_ROW_BYTES is added in the loop body (line 33). The ESI register is used as a column index. This works for row-major ordering.

5.5 Performance: Usefulness of Addressing Modes

The objectives of this section are to show the usefulness of the various memory addressing modes and the pitfalls of mixing 16-bit and 32-bit addressing modes.

Experiment 1

The objective of this experiment is to show the performance advantage of the various 16-bit addressing modes. In Program 5.19, we have used based-indexed addressing with and without displacement. For example, the two statements on lines 73 and 74 are equivalent to

```
mov    AX,[BX+DI]
mov    [BX+DI+2],AX
```

By using a displacement value of 2, we could use the same two registers to access the next element in the array. We will not have this kind of flexibility if

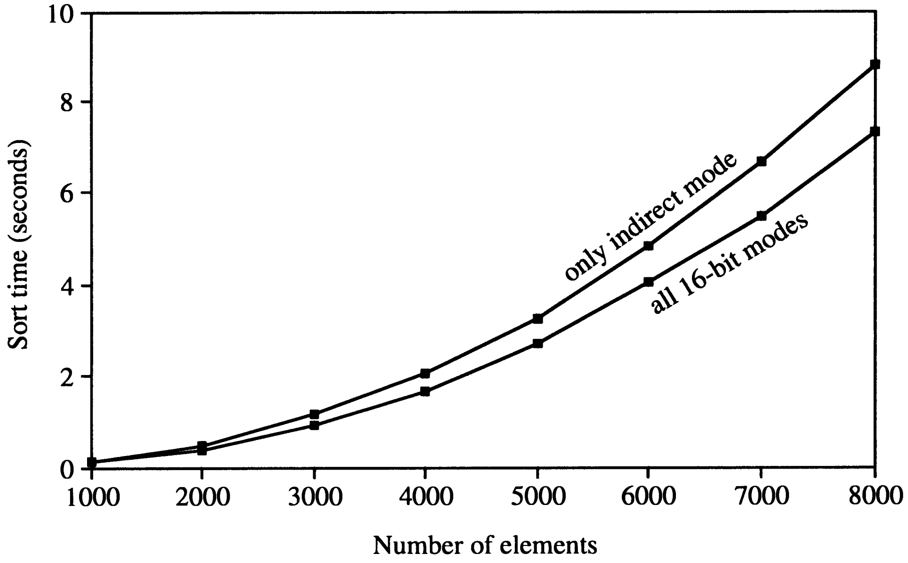


Figure 5.6 Performance impact of 16-bit addressing modes on the insertion sort.

we use only the indirect addressing mode. To see the relative performance, we have rewritten the insertion sort procedure discussed in Section 5.3 with only direct and register indirect addressing modes. We, however, have not modified how the parameters are accessed from the stack.

The performance implications of these changes is shown in Figure 5.6. The x-axis shows the size of the array and the y-axis gives the corresponding sort time. The performance of the procedure given in Program 5.19 is represented by the "all 16-bit modes" line, while the other line represents the performance of the modified version as discussed 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 20 percent!

Experiment 2

The goal of this experiment is to demonstrate the overheads associated with mixing 16-bit and 32-bit addressing modes. Remember that, by default, we use 16-bit operands and addresses. Thus, using 32-bit operands or addresses involves using size override prefixes—an overhead. To quantify this overhead,

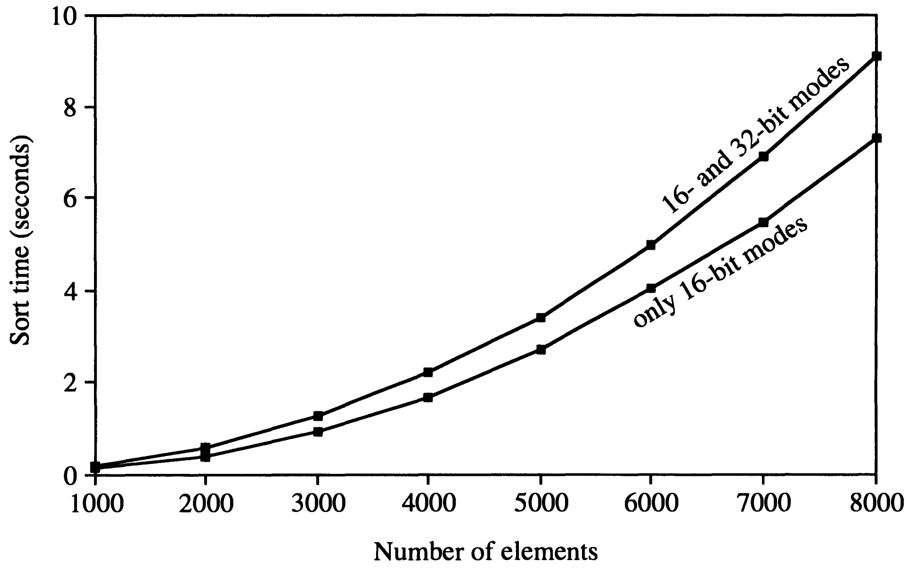


Figure 5.7 Performance impact of mixing 16-bit and 32-bit addressing modes on the insertion sort.

we have written the insertion sort procedure using both 16-bit and 32-bit addressing modes (see Program 5.23).

As shown in Program 5.23, all instructions of the `while_loop`, except for the `jge` instructions, have either an operand override prefix or an address size override prefix. Since decoding a prefix takes a clock cycle, the performance suffers for this application, as shown in Figure 5.7. Because of such overheads, Intel suggests using a 16-bit operand and addresses for 16-bit segments, and 32-bit operands and addresses for 32-bit segments as much as possible! In this book, for pedagogical reasons, we write programs that use 16-bit as well as 32-bit operands and addressing modes.

Program 5.23 Insertion sort procedure with 32-bit addressing modes

```
;-----------------------------------------------------------
; 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.
;-----------------------------------------------------------
```

```
SORT_ARRAY  EQU  [EBX]
insertion_sort PROC
       pushad                  ; save registers
       mov     BP,SP
       sub     EBX,EBX
       mov     BX,[BP+34]      ; copy array pointer
       mov     CX,[BP+36]      ; 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     DX,SORT_ARRAY[ESI*2] ; temp := array[i]
       mov     EDI,ESI         ; j := i-1
       dec     EDI
while_loop:
       cmp     DX,SORT_ARRAY[EDI*2]  ; temp < array[j]
       jge     exit_while_loop
       ; array[j+1] := array[j]
       mov     AX,SORT_ARRAY[EDI*2]
       mov     SORT_ARRAY[EDI*2+2],AX
       dec     EDI             ; j := j-1
       cmp     EDI,0           ; j >= 0
       jge     while_loop
exit_while_loop:
       ; array[j+1] := temp
       mov     SORT_ARRAY[EDI*2+2],DX
       inc     ESI             ; i := i+1
       dec     CX
       cmp     CX,1            ; if CX = 1, we are done
       jne     for_loop
sort_done:
       popad                   ; restore registers
       ret     4
insertion_sort ENDP

```

5.6 Summary

Addressing mode refers to the specification of data or operands required by an assembly language instruction. We discussed the various addressing modes supported by Pentium. There are a variety of ways in which a memory operand can be specified. We showed by means of examples how the various 16-bit and 32-bit addressing modes are useful in supporting features of high-level languages. We demonstrated by means of insertion sort that proper use of the advanced addressing modes can result in a better program—both in terms of improved readability and reduced execution time. We also showed the pitfalls in mixing 16-bit and 32-bit operands and addressing modes.

Arrays are useful to represent 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, it is necessary to know how the arrays are stored in memory. This is so because accessing individual elements of an array involves computing the corresponding displacement value. While there are two common ways of storing a multidimensional array—row-major or column-major order—most high-level languages, including C and Pascal, use the row-major order. We presented examples to illustrate the manipulation of one- and two-dimensional arrays.

5.7 Exercises

5–1 Explain why the register addressing mode is the most efficient of all the addressing modes supported by Pentium.

5–2 Discuss the restrictions imposed by the immediate addressing mode.

5–3 Where (i.e., which segment) is the data, specified by the immediate addressing mode, stored?

5–4 Describe all of the 16-bit addressing modes that you can use to specify an operand that is located in memory.

5–5 Describe all of the 32-bit addressing modes that you can use to specify an operand that is located in memory.

5–6 What are the differences between direct and register indirect addressing modes?

5–7 Which registers can be used in register indirect addressing mode? Also specify the default segments associated with each register used in this mode.

5–8 When is it necessary to use segment override prefix?

5–9 When is it necessary to use operand size override prefix?

5–10 When is it necessary to use address size override prefix?

5–11 Is there a fundamental difference between based and indexed addressing modes?

5–12 What additional flexibility does the based-indexed addressing mode have over based or indexed addressing modes?

5–13 Given the following declaration of table1

```
table1    DW    10 DUP (0)
```

fill in the blanks in the following code:

```
mov    SI, _______ ; SI := displacement of 5th element
                   ; (i.e., table1[4] in C)
mov    AX,table1[SI]
cmp    AX, _______ ; compare 5th and 4th elements
```

5–14 What is the difference between row-major and column-major orders for storing multidimensional arrays in memory?

5–15 In manipulating multidimensional arrays in assembly language, why is it necessary to know their underlying storage representation?

5–16 How is class_marks in Program 5.22 stored in memory—row-major or column-major order?

5–17 How would you change the class_marks declaration in order to store it in column-major order?

5–18 Assuming that subscripts begin with 0, derive a formula for the displacement (in bytes) of the element in row i and column j in a two-dimensional array stored in column-major order.

5–19 Suppose that the array **A** is a two-dimensional array stored in row-major order. As in Pascal, assume that a low value can be specified for each subscript. Derive a formula to express the displacement (in bytes) of **A**$[i,j]$.

5.8 Progamming Exercises

5–P1 What modifications would you make to the insertion sort procedure discussed in Section 5.3 to sort the array in descending order? Make the necessary modifications to the program and test it for correctness.

5–P2 Modify Program 5.21 to read array input data from the user. Your program should be able to accept up to twenty five nonzero numbers from the user. A zero terminates the input. An error should be reported if more than twenty five numbers are given.

5–P3 Modify Program 5.22 to read marks from the user. The first number of the input indicate the number of students in the 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 twenty students and five tests. Report error when exceeding these limits.

5–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 size of the matrices. Note that both **A** and **B** should have the same size. The `main` procedure calls another procedure to display **C**.

5–P5 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.

5–P6 Modify the program of the last exercise to work on matrices stored in column-major order.

5–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

$$\begin{bmatrix} 12 & 23 & 34 & 45 \\ 34 & 45 & 56 & 67 \\ 56 & 67 & 78 & 89 \\ 78 & 89 & 90 & 10 \end{bmatrix}$$

5–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].

5–P9 Write a program to read a matrix of integers and 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

$$\begin{bmatrix} 34 & 56 & 78 & 12 \\ 45 & 67 & 89 & 23 \\ 56 & 78 & 90 & 34 \\ 67 & 89 & 10 & 45 \end{bmatrix}$$

5–P10 Generalize the last exercise to cyclically permute by a user-specified number of elements.

5–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

5–P12 Modify the program for the last exercise to also produce 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

5–P13 If we are given a square matrix (i.e., a matrix with the number of rows equal to the number of columns), we can classify it as the diagonal matrix if only its diagonal elements are nonzero; as upper triangular matrix if all the elements below the diagonal are 0; as 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 6

Arithmetic Flags and Instructions

Objectives

- To discuss the significance of the six status flags
- To describe in detail the arithmetic instructions of the Pentium instruction set
- To illustrate how multiword arithmetic operations are done
- To demonstrate the performance implications of various ways of performing multiplication

The focus of this chapter is to discuss the arithmetic instructions of Pentium. All arithmetic operations affect some of the flags. In particular, there are six flags—called status flags—that are updated by most arithmetic instructions. These flags monitor the outcome of an arithmetic operation. For example, the zero flag is set if the outcome of an arithmetic operation is zero. Arithmetic flags are discussed in Section 6.1.

Pentium provides arithmetic instructions to perform addition, subtraction, multiplication, and division. While add and subtract instructions work on both signed and unsigned data, multiplication and division require separate instructions for signed and unsigned data. Details about the arithmetic instructions are provided in Section 6.2. Some example applications are presented in Section 6.3.

The arithmetic instructions can only be used on 8-, 16-, or 32-bit operands. Section 6.4 discusses how the basic four arithmetic operations can be performed

S. P. Dandamudi, *Introduction to Assembly Language Programming*

for numbers that use multiple words (for example, 64 bits). Performance issues are discussed in Section 6.5 and the chapter concludes with a summary.

6.1 Status Flags

Six flags in the flags register, described in Chapter 2, are used to monitor the outcome of the arithmetic, logical, and related operations. By now you are familiar with the purpose of some of these flags. The six flags are the zero flag (ZF), carry flag (CF), overflow flag (OF), sign flag (SF), auxiliary flag (AF), and parity flag (PF). These six flags are referred to as 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, while other instructions affect none of the flags. And there are other instructions that affect only a subset of the flags. For example, the arithmetic instructions `add` and `sub` affect all six flags, while `inc` and `dec` instructions affect all but the carry flag. The `mov`, `push` and `pop` instructions, on the other hand, do not affect any of the flags.

Here is an example illustrating how the zero flag changes with instruction execution.

```
;initially, assume that ZF is 0
mov     AL,55H  ; ZF is still 0
sub     AL,55H  ; result is zero
                ; Thus, ZF is set (ZF = 1)
push    BX      ; ZF remains 1
mov     BX,AX   ; ZF remains 1
pop     DX      ; ZF remains 1
mov     CX,0    ; ZF remains 1
inc     CX      ; result is 1
                ; Thus, ZF is cleared (ZF = 0)
```

As you have seen before, these flags are tested either singly or in combination to affect the flow control of a program. Chapter 7 discusses the conditional jump instructions in more detail.

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.

6.1.1 The Zero Flag

The purpose of the zero flag is to indicate whether the execution of the last instruction that affected the zero flag has produced a zero result. If the result was zero, ZF = 1; otherwise, ZF = 0. This is slightly confusing! You may want to take a moment to see through the confusion.

While it is fairly intuitive to understand how the sub instruction can affect the zero flag, it is not so obvious with other instructions. The following examples show some typical cases.

The following code

```
mov    AL,0FH
add    AL,0F1H
```

sets the zero flag (i.e., ZF = 1). This is because, after executing the add instruction, AL would contain zero (all eight bits zero). In a similar fashion, the following code

```
mov    AX,0FFFFH
inc    AX
```

also sets the zero flag. The same is true of the following code segment:

```
mov    AX,1
dec    AX
```

Related Instructions

jz	jump if zero (jump if ZF = 1)
jnz	jump if not zero (jump if ZF = 0)

Usage

There are two main uses of the zero flag: (i) to test equality, and (ii) to count to a preset value.

Testing 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 subtraction. Instead, the subtract operation is performed only to set the status flags.

Here are some examples:

```
cmp    char,'$'      ; ZF = 1 if char is $
```

Similarly, two registers can be compared to see if they both have the same value.

```
cmp     AX,BX
```

Counting to a Preset Value

Another important use of the zero flag is shown below. Consider the following code:

```
sum := 0
for (i = 1 to M)
    for (j = 1 to N)
        sum := sum + 1
    end for
end for
```

The equivalent in the assembly language can be written as follows (assuming that both *M* and *N* are ≥ 1):

```
        sub     AX,AX       ; AX:= 0 (AX stores sum)
        mov     DX,M
outer_loop:
        mov     CX,N
inner_loop:
        inc     AX
        loop    inner_loop
        dec     DX
        jnz     outer_loop
exit_loops:
        mov     sum,AX
```

In the above example, inner loop count is placed in the CX register so that we can use the loop instruction to iterate. The loop instruction is equivalent to

```
dec     CX
jnz     inner_loop
```

The above two instruction sequences are surprisingly more efficient than the loop instruction! More on this topic in Chapter 7. Incidentally, the loop instruction does not affect any of the flags! So the loop instruction is not strictly equivalent to the dec and jnz instruction sequence given above.

Since we have two nested loops to handle, we are forced to use another register to keep count of the outer loop iterations, and we use the dec instruction and the zero flag to see if the outer loop has been repeated *M* times. Again, this code is more efficient than initializing the DX register to one and using the following code

```
inc     DX
cmp     DX,M
jle     outer_loop
```

instead of

```
dec     DX
jnz     outer_loop
```

6.1.2 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 register or memory location. Consider the example

```
mov     AL,0FH
add     AL,0F1H
```

The addition of 0FH and F1H would produce a result of 100H that requires 9 bits to store, as shown below.

```
   00001111B   (0FH = 15D)
   11110001B   (F1H = 241D)
 1 00000000B   (100H = 256D)
```

Since the destination register AL is only 8 bits long, the carry flag would be set to indicate that the result is too big to be held in the AL register.

To understand when the carry flag is set, it is helpful to remember the range of unsigned numbers that can be represented. The range is given below for easy reference.

size	range
8 bits	0 to 255
16 bits	0 to 65,535
32 bits	0 to 4,294,967,295

Any operation that produces a result that is outside this range sets the carry flag to indicate an underflow or overflow condition. It is obvious that any negative result is out of range, as illustrated by the following example.

```
mov     AX,12AEH    ;AX := 4782D
sub     AX,12AFH    ;AX := 4782D - 4783D
```

Executing the above code will set the carry flag because 12AFH − 12AFH produces a negative result (i.e., the subtract operation generates a borrow), which is too small to be represented using unsigned numbers. Thus, the carry flag indicates this underflow condition.

Executing the code

```
mov    AL,0FFH
inc    AL
```

or the code

```
mov    AX,0
dec    AX
```

does not set the carry flag as we might expect because the `inc` and `dec` instructions do not affect the carry flag.

Related Instructions

Conditional jumps:

`jc`	jump if carry (jump if CF = 1)
`jnc`	jump if not carry (jump if CF = 0)

In addition to the conditional jump instructions that test the carry flag, there are three special instructions that directly manipulate the carry flag. There are:

`stc`	set carry flag	sets CF to 1
`clc`	clear carry flag	clears CF to 0
`cmc`	complement carry flag	inverts CF value

All three instructions affect only the carry flag and have no effect on the other flags. These three instructions are useful in conjunction with the rotate instructions `rcl` and `rcr`. An example that uses `stc` and `clc` instructions is given on page 221.

Usage

The carry flag is useful in several situations:

- To propagate carry or borrow in multiword addition or subtraction operations.
- To detect the overflow/underflow condition.
- 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 that are more than 32 bits each are to be added, the addition has to proceed in steps by adding 32 bits at a time. The following example shows 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 first add the least significant (lower half) 32 bits of the two operands. This produces the lower half of the result. This addition operation could potentially produce a carry that should be added to the upper 32 bits of the input. The second add operation performs the addition of the most significant (upper half) 32 bits and any carry generated by the previous addition. This operation produces the upper half of the 64-bit result. Section 6.4.1 gives an example to add two 64-bit numbers.

Similarly, adding two 128-bit numbers involves a four-step process, where each step adds 32 bits. The `sub` and other operations also require multiple steps when the data size is more than 32 bits.

To Detect an Overflow/Underflow Condition

In the previous example of x+y, if the second addition has produced 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 3), 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 left shift or rotate), or the least significant bit (in the case of right shift or rotate). Once the bit is in the carry flag, conditional execution of code is possible using the 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 two-fold:

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 the add instead of the inc instruction? 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 will normally expect the carry flag to be set to 1. But inc does not affect the carry flag. 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.

6.1.3 The Overflow Flag

The overflow flag, in some respects, is the carry flag counterpart for the signed number arithmetic. The main purpose of the overflow flag is to indicate whether or not 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 convenience, the range of the numbers is given below:

size	range
8 bits	−128 to +127
16 bits	−32,768 to +32,767
32 bits	−2,147,483,648 to +2,147,483,647

Executing the code

```
mov     AL,72H  ; 72H = 114D
add     AL,0EH  ; 0EH = 14D
```

will set the overflow flag to indicate that the result 80H (128D) is too big to be represented as a signed number using only 8 bits. The AL register will contain 80H, the correct result if the two 8-bit operands were treated as unsigned numbers. But AL contains an incorrect answer for 8-bit signed numbers (80H represents −128 in signed representation, not +128 as is required).

Here is another example using the sub instruction. The AX register is initialized to −5, which is FFFBH in 2's complement representation using 16 bits.

```
mov    AX,0FFFBH    ; AX := -5
sub    AX,7FFDH     ; subtract 32,765 from AX
```

Execution of the above code will set the overflow flag as the result

$$(-5)-(32{,}765) = -32{,}770$$

which is too small to be represented as a 16-bit signed number.

Note that the result will not be out of range (and hence the overflow flag will not be set) when we are adding two signed numbers of opposite sign or subtracting two numbers of the same sign.

Signed or Unsigned: How Does the System Know?

The values of the carry and overflow flags depend on whether the operands are unsigned or signed numbers. Given that a bit pattern can both be treated 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 whether a given bit pattern represents a signed or an unsigned number. It is up to your 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 your program logic to take whatever 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, your program logic can disregard the carry flag value. Since the overflow flag is 1, your program logic will have to handle the overflow condition.

```
        mov     AL,72H
        add     AL,0EH
        jo      overflow
no_overflow:
        (no overflow code here)
                .
                .
overflow:
        (overflow code here)
                .
                .
```

Related Instructions

Conditional jumps:

`jo`	jump on overflow (jump if OF = 1)
`jno`	jump on no overflow (jump if OF = 0)

In addition, a special software interrupt instruction

`into` interrupt on overflow

is provided that tests the overflow flag. Interrupts are discussed in Chapter 12.

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.

6.1.4 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 if positive and 1 if negative. The sign flag gets a copy of the sign bit of the result produced by an arithmetic or a related operation. The following sequence of instructions

```
mov    AL,15
add    AL,97
```

will clear the sign flag (i.e., SF = 0) because the result produced by add is a positive number: 112D (which in binary is 01110000, where the leftmost bit representing the sign is zero).

The result produced by

```
mov    AL,15
sub    AL,97
```

is a negative number and will set the sign flag to indicate this fact. Remember that negative numbers are represented in 2's 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 2's complement form. Therefore, after executing the above two instructions, the AL register will contain AEH, as shown below.

```
  00001111B     (8-bit signed form of 15)
+ 10011111B     (8-bit signed form of −97)
  ---------
  10101110B
```

Since the sign bit of the result is 1, the result is negative and is in 2's complement form. You can easily verify that AEH is the 8-bit signed form of −82, which is the correct answer.

Related Instructions

Conditional jumps:

```
js   jump on sign (jump if SF = 1)
jns  jump on no sign (jump 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 non-negative.

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 cmp instruction as follows:

```
        mov     CX,M
for_loop:
            .
        <loop body>
            .
        dec     CX
        jns     for_loop
```

If we do not use the jns instruction, we have to use in its place

```
        cmp     CX,0
        jl      for_loop
```

From the user point of view, a sign bit of a number can easily be tested by using either a logical instruction or a 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 Chapter 7).

6.1.5 The Auxiliary Flag

The auxiliary flag indicates whether an operation has produced a result that has generated a carry out of or 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 following example

```
mov     AL,43
add     AL,94
```

the auxiliary flag is set to 1 because there is a carry out of bit 3, as shown below:

```
                1 ← carry generated from lower to upper nibble
  43D  = 00101011B
  94D  = 01011110B
 ----    ---------
 137D  = 10001001B
```

You can verify that executing the following code will clear the auxiliary flag.

```
mov    AL,43
add    AL,85
```

Since the following instruction sequence

```
mov    AL,43
sub    AL,92
```

generates a borrow into the low-order 4 bits, the auxiliary flag is set. On the other hand, the instructions

```
mov    AL,43
sub    AL,87
```

clear the auxiliary flag.

Related Instructions

There are no conditional jump instructions that test the auxiliary flag. However, arithmetic operations on numbers expressed in decimal form or binary coded decimal form (instead of the standard binary form) use the auxiliary flag. Some related instructions are

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

More details on these instructions are given in Chapter 11.

Usage

The main use of this flag is in performing arithmetic operations on BCD numbers (see Chapter 11 for details on the BCD number representation). Also, see Chapter 11 for details on related instructions such as daa, das, etc.

6.1.6 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 low-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 is 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 1's (four 1 bits), as shown below.

$$\begin{array}{rcl} 53D & = & 00110101B \\ 89D & = & 01011001B \\ \hline 142D & = & 10001110B \end{array}$$

The instruction sequence

```
mov    AX,23994
sub    AX,9182
```

on the other hand, clears the parity flag, as the low-order 8 bits contain an odd number of 1's (five 1 bits), as shown below.

$$\begin{array}{rrcl} & 23994D & = & 01011101\ 10111010B \\ + & -9182D & = & 11011100\ 00100010B \\ \hline & 14813D & = & 00111001\ 11011100B \end{array}$$

Related Instructions

Conditional jumps:

`jp`	jump on parity (jump if PF = 1)
`jnp`	jump on no parity (jump 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 for a total of 8 bits. 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 1's 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 1's, 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 are an even number of 1's. The parity bit is shown as the most significant bit in the following examples.

```
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., eighth bit from the right) is assumed to be zero.

```
parity_encode PROC
      shl    AL
      jp     parity_zero
      stc
      jmp    move_parity_bit
parity_zero:
      clc
move_parity_bit:
      rcr    AL
parity_encode ENDP
```

6.1.7 Flag Examples

Table 6.1 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 is an even number of 1's in the result. In the rest of this section, we will focus on the carry and overflow flags.

Example 1 performs −5−123. Noting that −5 is represented internally as FBH (=251D), subtracting 123 (=7BH) leaves 80H (=128D) in AL. Since the result

Table 6.1 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

is within the range of unsigned 8-bit numbers, CF is cleared. For the OF, the operands are interpreted as signed numbers. Since the result is −128D, OF is also cleared.

Example 2 subtracts 124D from −5. For reasons discussed in the previous example, the CF is cleared. The OF, however, is set because the result is −129D, which is outside the range of signed 8-bit numbers.

In Example 3, the first add statement adds 132 to −5. However, when treating as unsigned numbers, 132 is actually added to 251D, which results in a number that is greater than 255D. Therefore, CF is set. When treating as signed numbers, 132D is internally represented as 84H (=−124D). Therefore, the result −129D is smaller than −128D. Therefore, the OF is also set. As a result of executing the first add instruction, AL will contain 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 1's 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.

6.2 Arithmetic Instructions

Pentium provides several instructions to perform 8-, 16-, and 32-bit addition, subtraction, multiplication, and division. Here is a look at the set of assembly language arithmetic instructions that we will be discussing next.

Addition: `add, adc, inc`
Subtraction: `sub, sbb, dec, neg, cmp`
Multiplication: `mul, imul`
Division: `div, idiv`
Related instructions: `cbw, cwd, cdq, cwde, movsx, movzx`

There are a few other arithmetic instructions that operate on decimal numbers. These are covered in Chapter 11.

6.2.1 Addition Instructions

The basic add instruction has the format

```
add    destination, source
```

which performs the following action:

```
destination := destination + source
```

The add instruction can take one of the five different forms depending on how the source and destination operands are specified. Recall that most two-operand instructions like add can be written in one of the five forms (see Chapter 3):

```
add    register, register
add    register, immediate
add    memory, immediate
add    register, memory
add    memory, register
```

As stated in Chapter 3, there are some restrictions on which registers can be used to specify the operands. Since these restrictions are common to all two-operand instructions that we discuss, we will not specifically mention them with each instruction.

Addition is a commutative operation. That is, it does not matter whether you write a+b or b+a. Therefore, the instruction

```
add    AX,BX
```

is equivalent to the instruction

```
add    BX,AX
```

except for the fact that the result is stored in a different register. In this sense, the add instruction itself is not "commutative."

The add instruction works with both signed and unsigned numbers and affects all six status flags. Table 6.1 on page 222 gives some examples.

The second version of the addition instruction is the adc (add with carry) instruction. This instruction has the general format

```
adc    destination, source
```

and the semantics of this instruction are

```
destination := destination + source + CF
```

The only difference between add and adc is that the adc instruction adds the contents of the carry flag (CF). The adc instruction is useful in performing addition of long multiword numbers (i.e., numbers that take more than 32 bits). The three instructions that manipulate the carry flag—stc, clc, and cmc—are useful in conjunction with instructions like adc.

The last instruction that performs addition is the single-operand inc instruction with the following format:

```
inc destination
```

where the destination operand can be either in a register or in memory. The operand of the inc instruction is treated as an unsigned number. For the sake of completeness, we state the action taken by inc as

```
destination := destination + 1
```

In general,

```
inc    BX
```

is preferred to

```
add    BX,1
```

because the inc instruction requires less memory space, even though both take one clock cycle to execute. The two instructions, however, are not exactly equivalent even though they both produce the same result. The chief difference is that the inc instruction does not affect the carry flag, whereas the add instruction does (see page 214).

As an example, consider the following two code fragments:

```
            add version                 inc version

            clc                         clc
            mov     CX,0FFFFH           mov     CX,0FFFFH
            add     CX,1                inc     CX
            jc      add_one             jz      add_one
                .                           .
                .                           .
        add_one:                    add_one:
            inc     DX                  inc     DX
                .                           .
                .                           .
```

The add version can either use jc or jz to detect overflow, whereas the inc version should use jz.

6.2.2 Subtraction Instructions

There are three subtract instructions corresponding to the add, adc, and inc instructions discussed in the last section. These are the sub, sbb, and dec instructions. The general format of the sub instruction is

```
sub    destination,source
```

which performs the following action:

```
destination := destination - source
```

The actual subtract operation is implemented by negating the source operand and then adding it to the destination operand. Thus, the actions performed by the sub instruction are

```
destination := destination + (-source)
```

Since the subtract operation is noncommutative, proper specification of source and destination operands is important even for correct operation. Like the add instruction, the sub instruction works with both the signed and unsigned integers. It also affects all six status flags (see Table 6.1 on page 222).

The second subtract instruction sbb (subtract with borrow) is the adc counterpart. The syntax is

```
sbb    destination, source
```

and the semantics are

```
destination := destination - source  - CF
```

The second subtract operation is done only if CF is 1. As in the sub instruction, the subtract operation is replaced by addition of the corresponding negative operands. This instruction works both on unsigned and signed binary numbers and updates all six status flags. This instruction is useful in performing subtract operation on numbers that are longer than 32 bits.

The third instruction dec (decrement) is a single-operand instruction with the following syntax:

```
dec    destination
```

It subtracts one from the destination

```
destination := destination - 1
```

The operand is treated as an unsigned number. This instruction, like the inc instruction, updates all status flags except the carry flag.

The next instruction that we discuss is the neg (negate) instruction, which is also a single-operand instruction. The instruction

```
neg    destination
```

subtracts the destination operand from 0. Thus, this instruction effectively reverses the sign of an integer and is meaningful only with signed numbers.

```
destination := 0 - destination
```

neg updates all six status flags. The carry flag is always set except when the operand is zero, in which case it is cleared.

There is a slight problem when negating the smallest number that can be represented. Recall that, with 8 bits we can represent signed integers in the range −128 to +127. What happens if we try to negate −128, as in the following example:

```
mov    AL,-128
neg    AL
```

Since +128 is out of range, the overflow flag will be set and there will be no change in the operand value (it remains −128). A similar situation arises with 16- and 32-bit operands.

With add and neg instructions, we can implement

```
sub    destination, source
```

as a sequence of

```
neg    source
add    destination, source
```

However, the former is more efficient and convenient. Furthermore, it aids in program readability.

The last instruction cmp (compare) has the format

```
cmp    destination, source
```

which effectively subtracts the source operand from the destination operand but does not affect any of the two operands, as shown below:

```
destination - source
```

The flags are updated as if the sub operation has been performed. The main purpose of the cmp instruction is to update the flags so that a subsequent conditional jump instruction can test these flags. More will be said about this instruction in Chapter 7, which discusses the branching and iterative instructions available in the Pentium assembly language.

6.2.3 Multiplication Instructions

Multiplication is more complicated than the addition and subtraction operations for two reasons:

1. First, multiplication produces double-length results. For example, if the two operands are 8 bits each, the result requires 16 bits. To convince you 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.

$$\begin{array}{ccccc} 11111111 & \times & 11111111 & = & 11111110\,11111111 \\ (255D) & & (255D) & & (65025D) \end{array}$$

   Similarly, you can verify that multiplication of two 16-bit numbers requires 32 bits to store the result, and two 32-bit numbers require 64 bits for the result.

2. Second, unlike the addition and subtraction operations, multiplication of signed numbers should be treated differently from that of unsigned numbers. This is because the resulting bit pattern depends on the type of input, as illustrated by the following example.

   We have already seen that treating FFH as the unsigned number results in multiplying 255D × 255D.

11111111 × 11111111 = 11111110 11111111

Now, what if FFH is representing a signed number? In this case, FFH is representing −1D and the result should be 1, as shown below.

11111111 × 11111111 = 00000000 00000001

As you can see, the resulting bit patterns are different for the two cases.

Thus, Pentium 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

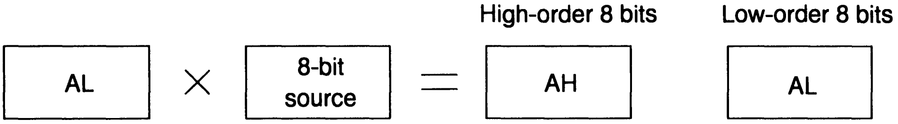
```
mul    source
```

The source operand can be in a general-purpose register or in memory. Immediate operand specification is not allowed. Thus,

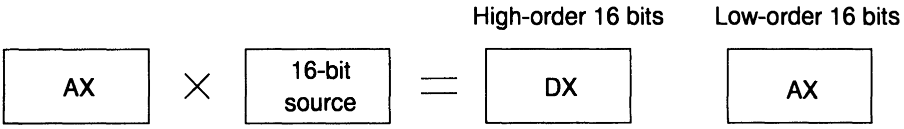
```
mul    10         ; invalid
```

is an invalid instruction. The mul instruction works on 8-, 16-, and 32-bit unsigned numbers.

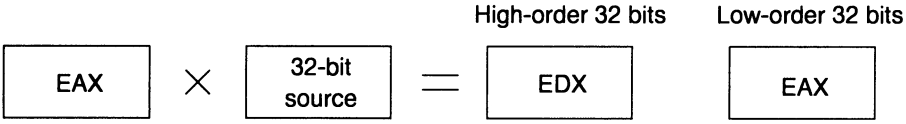
If the source operand is a byte, it is multiplied by the contents of the AL register. The 16 bit result is placed in the AX register, as shown below.



If the source operand is a word, it is multiplied by the contents of the AX register and the doubleword result is placed in the DX and AX register pair, with the AX register holding the lower-order 16 bits, as shown below.



If the source operand is a double word, it is multiplied by the contents of the EAX register and the 64-bit result is placed in the EDX and EAX register pair, 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 following code

```
mov    AL,10
mov    DL,25
mul    DL
```

will clear both the carry and the overflow flags, as the result of the `mul` is 250D, which can be stored in the AL register (and the AH register contains 00000000). On the other hand, executing

```
mov    AL,10
mov    DL,26
mul    DL
```

will set the carry and overflow flags indicating that the result is more than 255D.

The multiplication instruction that works on signed numbers is `imul` (Integer multiplication) and has the same format[1] as the `mul` instruction

```
imul    source
```

The behaviour 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 the sign extension in signed numbers, consider the following example.

We know that −66D 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

[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* for details.

1111111110111110

The upper 8 bits are simply sign-extended (i.e., the sign bit is copied into these bits), and doing so does not change the magnitude.

Following the same logic, the positive number 66, represented using 8 bits as

01000010

can be sign-extended to 16 bits by

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 to store.

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 CF and OF are cleared, as the AH is the sign extension of AL. This is also the case of the following code:

```
mov    DL,0FFH    ; DL := -1
mov    AL,0BEH    ; AL := -66
imul   DL
```

which produces the result

0000000001000010 (+66D)

in the AX register. Since the AH register is the sign extension of the AL register, 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 (−250D)

6.2.4 Division Instructions

The division operation is even more complicated than the multiplication for two reasons:

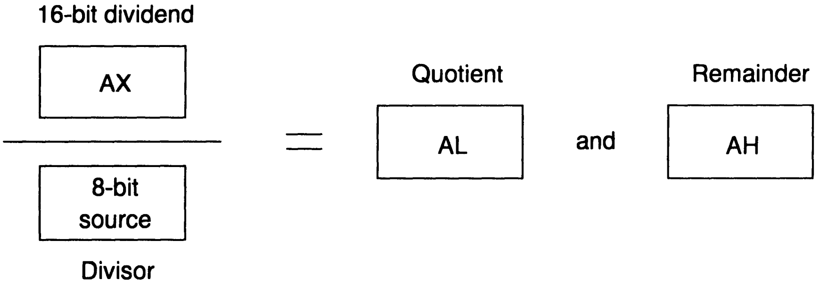
1. Division generates two result components—quotient and remainder.
2. In multiplication, by using double-length registers, no overflow occurs. In division, divide overflow is a real possibility and Pentium generates a special software interrupt when a divide overflow occurs.

As with multiplication, two versions of the divide instruction are provided to work on unsigned and signed numbers.

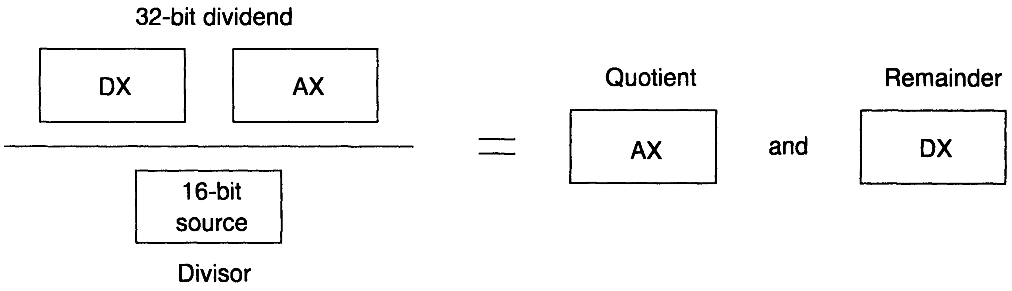
```
div    source  (unsigned)
idiv   source  (signed)
```

The source operand specified in the instruction is used as the divisor. Both instructions can work on 8-, 16-, or 32-bit numbers. All six status flags are affected and are undefined. None of the flags are updated. We will first consider the unsigned version.

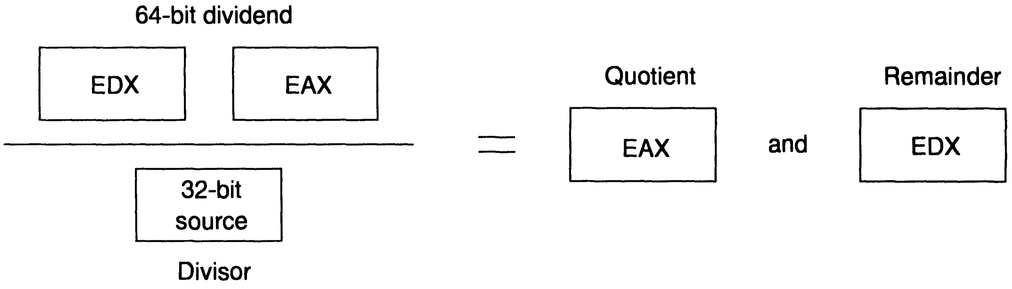
If the source operand is a byte, the dividend is assumed to be in the AX register and 16 bits long. After division, the quotient is returned in the AL register and the remainder in the AH register, as shown below.



For a word-size operand, the dividend is assumed to be 32 bits long and in the DX and AX registers (upper 16 bits in DX). After the division, the 16-bit quotient will be in AX and the 16-bit remainder in DX, as shown below.



For a 32-bit operand, the dividend is assumed to be 64 bits long and in the EDX and EAX registers (upper 32 bits in EDX). After the division, the 32-bit quotient will be in EAX and the 32-bit remainder in EDX, as shown below.



Example 6.1 *8-bit division*

This example considers 8-bit division of 251/12 = 20 with 11 as the remainder. The following code

```
mov    AX,00FBH    ; AX := 251D
mov    CL,0CH      ; DL := 12D
div    CL
```

will leave 14H (20D) in the AL register and 0BH (11D) as the remainder in the AH register. □□□□□□

Example 6.2 *16-bit division*

Now, let us look at 16-bit division. Consider 5147/300 = 17, with 47 as the remainder. The following code

```
mov    DX,0        ; clear DX
mov    AX,141BH    ; AX := 5147D
mov    CX,012CH    ; CX := 300D
div    CX
```

will leave 0012H (17D) in AX and 002FH (47D) in DX as the remainder of the division. □□□□□□

Now let us turn our attention to the signed division operation. The `idiv` instruction has the same format and behaviour 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 a 16-bit division of −251 by 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 DX were 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. Also, for a positive dividend, the DX should have 0000H.

To aid sign extension in instructions like the `idiv` operation, Pentium provides several instructions.

`cbw`	(convert byte to word)
`cwd`	(convert word to double word)
`cdq`	(convert double word to quad word)

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 AX into the DX register and is useful with the 16-bit `idiv` instruction. The `cdq` instruction sign extends EAX into 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 registers.

For completeness, we mention three other related instructions. The `cwde` instruction sign extends AX into EAX much like 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.

Pentium also provides the following two move instructions:

`movsx`	`dest,src`	(move sign-extended `src` to `dest`)
`movzx`	`dest,src`	(move zero-extended `src` to `dest`)

In both these instructions, `dest` has to be a register, while the `src` operand can be in either 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 along with `cbw` and `cwd` instructions.

Example 6.3 *Signed 8-bit division*

The following sequence of instructions will perform a signed 8-bit division of −95 by 12.

```
mov     AL,0A1H    ; AL := -95
cbw                ; AH := FFH
mov     CL,0CH     ; CL := 12
idiv    CL
```

The `idiv` instruction will leave F9H (−7D) in AL and F5H (−11D) in the AH register as the remainder of the division. □□□□□□

Example 6.4 *Signed 16-bit division*

This example considers 16-bit division. Suppose that we want to divide −5147D by 300D. The following sequence

```
mov     AX,0EBE5H    ; AX := -5147D
cwd                  ; DX := FFFFH
mov     CX,012CH     ; CX := 300D
idiv    CX
```

will perform this division and leaves FFEFH (−17D) in AX and FFD1H (−47D) in DX as the remainder. □□□□□□

6.3 Application Examples

To demonstrate the application of the arithmetic instructions and flags, we write two procedures to input and output signed 8-bit integers in the range of −128 to +127. These procedures are:

`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.

6.3.1 PutInt8 Procedure

Our objective here is to write a procedure that displays the signed 8-bit integer in the AL register. In order to do this, we have to separate individual digits of the number to be displayed and convert them to its ASCII representation. The steps involved are illustrated by the following example, which assumes that AL = 108D.

separate 1 → convert to ASCII → 31H → display
separate 0 → convert to ASCII → 30H → display
separate 8 → convert to ASCII → 38H → display

As you can see, separating individual digits is the heart of the procedure. This step is surprisingly simple! All you have to do is repeatedly divide 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 as follows:

```
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
     (number = 0)
    repeat
        index := index − 1
        display digit at buffer[index]
     (index = 0)
end PutInt8
```

Program 6.24 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 PROC
        push    BP
        mov     BP,SP
        sub     SP,3           ; local buffer space
        push    AX
        push    BX
        push    SI
        test    AL,80H         ; negative number?
        jz      positive
negative:
        PutCh   ’-’            ; sign for negative numbers
        neg     AL             ; convert to magnitude
positive:
        mov     BL,10          ; divisor  = 10
        sub     SI,SI          ; SI := 0 (SI points to buffer)
repeat:
        sub     AH,AH          ; AH := 0 (AX is the dividend)
        div     BL
        ; AX/BL leaves AL:= quotient & AH := remainder
        add     AH,’0’         ; convert remainder to ASCII
        mov     [BP+SI-3],AH ; copy into the buffer
        inc     SI
        cmp     AL,0           ; quotient = zero?
        jne     repeat         ; if so, display the number
display_digit:
        dec     SI
        mov     AL,[BP+SI-3] ; display digit pointed by SI
        PutCh   AL
        jnz     display_digit ; if SI<0, done displaying
display_done:
        pop     SI             ; restore registers
        pop     BX
        pop     AX
        mov     SP,BP          ; clear local variable space
        pop     BP
        ret
```

```
41:  PutInt8 ENDP
```

The PutInt8 procedure shown in Program 6.24 follows the logic of the pseudocode. Some points to note are:

- Buffer is considered as a local variable. Thus, we reserve 3 bytes in the stack (see line 8).
- The code

  ```
  test   AL,80H
  jz     positive
  ```

  tests whether the number is negative or positive. Remember that the sign bit (the leftmost bit) is 1 for a negative number.
- Reversing sign is done by the

  ```
  neg    AL
  ```

  instruction on line 16.
- Note that we have to initialize AH with 0 (line 21), as the div instruction assumes a 16-bit dividend in the AX register when the divisor is an 8-bit number.
- Conversion to ASCII character representation is done on line 24 using

  ```
  add    AH,'0'
  ```

- SI is used as the index into the buffer, which starts at [BP−3]. Thus, [BP+SI−3] points to the current byte in the buffer (line 31).
- The repeat while condition (index = 0) is tested by

  ```
  jnz    display_digit
  ```

  on line 33.

6.3.2 GetInt8 Procedure

The GetInt8 procedure reads a signed integer and returns the number in the AL register. Since only 8 bits are used to represent the number, the range is limited to −128 to +127 (both inclusive). The key part of the procedure converts a sequence of input digits received in character form to its binary equivalent. The conversion process, which involves repeated multiplication by 10, is illustrated for the input number 158.

input digit	numeric value	number := number * 10 + numeric value
initial value	–	0
'1' (31H)	1	0*10+1 = 1
'5' (35H)	5	1*10+5 = 15
'8' (38H)	8	15*10+8 = 158

The pseudocode of the GetInt8 procedure is as follows:

```
GetInt8()
    read input character into char
    if ((char = '−') OR (char = '+'))
    then
        sign := char
        read the next character into char
    end if
    number := char − '0' {convert to numeric value}
    count := 2 {number of remaining digits to read}
repeat
    read the next character into char
    if (char ≠ carriage return)
    then
        number := number * 10 + (char − '0')
    else
        goto convert_done
    end if
    count := count − 1
 (count = 0)
convert_done:
    {check for out-of-range error}
    if ((number > 128) OR ((number = 128) AND (sign ≠ '−')))
    then
        out of range error
        set carry flag
    else {number is OK}
        clear carry flag
    end if
    if (sign = '−')
    then
        number = −number {reverse sign}
    end if
end GetInt8
```

Program 6.25 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.
;-----------------------------------------------------------
CR     EQU   0DH

GetInt8 PROC
        push    BX           ; save registers
        push    CX
        push    DX
        sub     DX,DX        ; DX := 0
        sub     BX,BX        ; BX := 0
get_next_char:
        GetCh   DL           ; read input from keyboard
        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
        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     CX,2         ; maximum two more digits to read
convert_loop:
        GetCh   DL
        cmp     DL,CR        ; carraige return?
        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:
```

```
41:         cmp    AX,128
42:         ja     out_of_range ; if AX > 128, number out of range
43:         jb     number_OK    ; if AX < 128, number is valid
44:         cmp    BH,'-'       ; AX = 128. Must be a negative;
45:         jne    out_of_range ; otherwise, an invalid number
46: number_OK:
47:         cmp    BH,'-'       ; number negative?
48:         jne    number_done  ; if not, we are done
49:         neg    AL           ; else, convert to 2's complement
50: number_done:
51:         clc                 ; CF := 0 (no error)
52:         jmp    done
53: out_of_range:
54:         stc                 ; CF := 1 (range error)
55: done:
56:         pop    DX           ; restore registers
57:         pop    CX
58:         pop    BX
59:         ret
60: GetInt8 ENDP
```

The assembly language code for the GetInt8 procedure is given in Program 6.25. The procedure uses GetCh to read input digits into the DL register.

- The character input digits are converted to their numeric equivalent by subtracting '0' on line 29.
- The multiplication is done on line 37, which produces a 16-bit result in AX. Note that the numeric value of the current digit (in DX) is added (line 38) to detect the overflow condition rather than the 8-bit value in DL.
- When the conversion is done, AX will have the absolute value of the input number. Lines 41–45 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

- If the input is a negative number, the value in AL is converted to 2's complement representation by using the neg instruction (line 49).
- The clc (clear CF) and stc (set CF) instructions are used to indicate the error condition (lines 51 and 54).

6.4 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.

6.4.1 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 6.26 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/undeflow
;conditions are indicated by setting the carry flag.
;Other registers are not disturbed.
;------------------------------------------------------------
add64    PROC
      add    EAX,ECX
      adc    EBX,EDX
      ret
add64    ENDP
```

The 64-bit subtraction is also simple and similar to the 64-bit addition. For 64-bit subtraction, substitute sub for add and sbb for adc in the add64 procedure.

6.4.2 Multiplication

Multiplication of multiword operands is not as straightforward as addition and subtraction. In this section, we will 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 bit tested 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 A and B 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		
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

For more information on multiword arithmetic operations, see the excellent appendix by Goldberg in *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson. This algorithm and the division algorithm given in the next section are based on the procedures given in this text.

Program 6.27 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.
;-----------------------------------------------------------
COUNT   EQU   WORD PTR [BP-2]    ; local variable

mult64   PROC
        push    BP
        mov     BP,SP
        sub     SP,2            ; local variable
        push    ESI
        push    EDI
        mov     ESI,EDX         ; SI:DI := B
        mov     EDI,ECX
        sub     EDX,EDX         ; P := 0
        sub     ECX,ECX
        mov     COUNT,64        ; count = 64 (64-bit number)
step:
        test    AX,1            ; LSB of A is 1?
        jz      shift1          ; if not, skip add
        add     ECX,EDI         ; Otherwise, P := P+B
        adc     EDX,ESI
shift1:                         ; shift right P and A
        rcr     EDX,1
        rcr     ECX,1
```

```
29:         rcr     EBX,1
30:         rcr     EAX,1
31:
32:         dec     COUNT           ; if COUNT is not zero
33:         jnz     step            ;  repeat the process
34:         ; restore registers
35:         pop     EDI
36:         pop     ESI
37:         mov     SP,BP           ; clear local variable space
38:         pop     BP
39:         ret
40: mult64    ENDP
```

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 [BP−2]. The EQU statement on line 8 establishes a convenient label to refer to it.
- The while loop is implemented by lines 21–33. The if condition is implemented by the test instruction on line 22.
- The 64-bit addition of P+B is done by lines 24–25. These two statements are similar to the code given in the add64 procedure.
- Right shift of CF:P:A is done by the four 16-bit rcr statements (lines 27–30). Note that the test instruction (line 22) 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 will 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 zeroes on the right}
result := result + temp
temp := A0 × B1
temp := left shift temp value by 32 bits
{the shift operation replaces zeroes on the right}
result := result + temp
temp := A1 × B1
temp := left shift temp value by 64 bits
{the shift operation replaces zeroes on the right}
result := result + temp
```

The procedure `mult64w` follows the above algorithm in a straightforward fashion. The 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 EQU statements on lines 9–12 assign labels to them.

Program 6.28 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: RESULT3  EQU  DWORD PTR [BP-4] ; most significant 32 bits of result
10: RESULT2  EQU  DWORD PTR [BP-8]
11: RESULT1  EQU  DWORD PTR [BP-12]
12: RESULT0  EQU  DWORD PTR [BP-16]; least significant 32 bits of result
13:
14: mult64w   PROC
15:       push    BP
16:       mov     BP,SP
```

```
      sub     SP,16          ; local variables for the result
      push    ESI
      push    EDI
      mov     EDI,EAX        ; ESI:EDI := A
      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
      mov     SP,BP          ; clear local variable space
      pop     BP
      ret
mult64w   ENDP
```

6.4.3 Division

There are several division algorithms to perform n-bit unsigned integer division. Here we describe and implement what is called the "non-restoring" 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 by 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 the sign of P, either adding or subtracting 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 is 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 6.29. The 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 in div64 to SIGN:EDX:ECX, where SIGN is a local variable that is used to store the sign of P. The code on lines 20–23 checks if the divisor is zero. If zero, the carry flag is set (line 24) 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. The rest of the code follows the algorithm.

Program 6.29 Division of two 64-bit numbers

```
 1: ;-----------------------------------------------------------
 2: ;Divides two 64-bit unsigned numbers A and B (i.e., A/B).
 3: ;The number A is received in EBX:EAX and B in EDX:ECX registers.
 4: ;The 64-bit quotient is returned in EBX:EAX registers and
 5: ;the remainder is retuned in EDX:ECX registers.
 6: ;Divide by zero error is indicated by setting
 7: ;the carry flag; carry flag is cleared otherwise.
 8: ;Preserves all registers except EAX, EBX, ECX, and EDX.
 9: ;-----------------------------------------------------------
10: ; local variables
11: SIGN       EQU  BYTE PTR [BP-1]
12: BIT_COUNT  EQU  BYTE PTR [BP-2]
13: div64  PROC
14:       push    BP
15:       mov     BP,SP
16:       sub     SP,2            ; local variable space
17:       push    ESI
18:       push    EDI
19:       ; check for zero divisor in DX:CX
20:       cmp     ECX,0
21:       jne     non_zero
22:       cmp     EDX,0
23:       jne     non_zero
24:       stc                     ; if zero, set carry flag
25:       jmp     SHORT skip      ; to indicate error and return
26: non_zero:
27:       mov     ESI,EDX         ; SI:DI := 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
        mov     SP,BP           ; clear local variable space
        pop     BP
        ret
div64  ENDP
```

6.5 Performance: Multiword Multiplication

We have discussed two methods for performing multiplication. Let us now look at the performance implications of these methods.

Experiment 1

We have presented a general multiplication algorithm based on longhand multiplication in Section 6.4.2. The alternative algorithm uses the mul instruction. Figure 6.1 shows the performance of these methods for performing

$$2^{64} - 1 \times 2^{64} - 1$$

As you can see, the mul instruction version is more than five times as fast. This is expected, as the longhand version performs multiplication on a bit-by-bit basis. This algorithm is suitable for implementation in hardware but not in software.

Experiment 2

In this experiment, we want to see if we can do better than 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 be done using only additions. Such multiplication, for example, is needed in the GetInt8 procedure. Suppose we want to multiply contents of AL (say X) by 10. This can be done as follows:

```
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
```

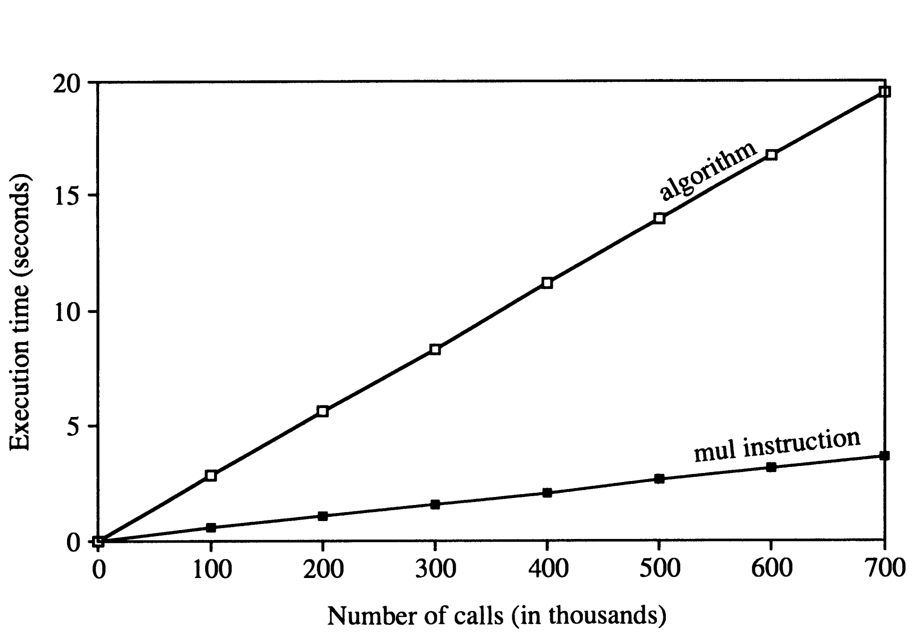


Figure 6.1 Performance of two procedures for multiplying two 64-bit numbers.

```
add     AX,AX       ; AX := 10X
```

Figure 6.2 shows the performance of the these two versions for multiplying $2^{32} - 1$ by 10. The add instruction version is about twice as fast. In Chapter 8 we will show that special multiplications by a power of 2 (often required for conversion of numbers from octal or hexadecimal systems) can be efficiently done by shift instructions.

6.6 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 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 on signed data.

The instruction set of Pentium includes instructions for addition, subtraction, multiplication, and division. While add and subtract instructions work on

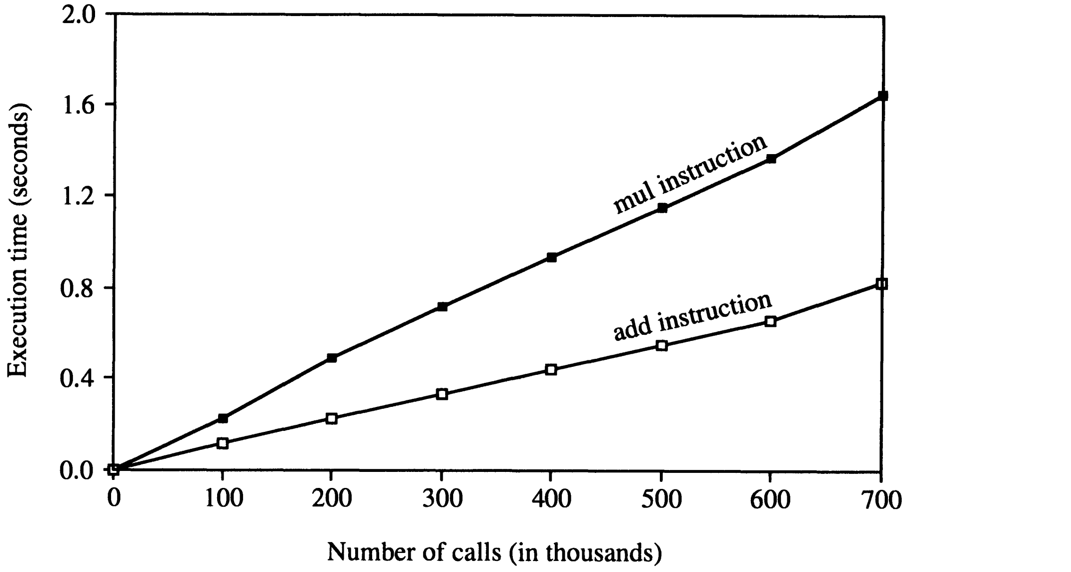


Figure 6.2 Performance of multiplication of a 32-bit number by 10.

both unsigned and signed data, separate instructions are required for signed and unsigned data for performing multiplication and division.

The arithmetic instructions of Pentium 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 can be implemented.

We demonstrated that multiplication by special values (for example, multiplication by 10) can be done more efficiently by using addition. Chapter 8 discusses how the shift operations can be used to implement multiplication by a power of 2. Such multiplication is often required to convert numbers from octal or hexadecimal number systems to the decimal system.

6.7 Exercises

6–1 What is the significance of the carry flag?

6–2 What is the significance of the overflow flag?

6–3 Suppose the sign flag is not available. Is there a way to detect the sign of a number? Is there more than one way?

6–4 When is the parity flag set? What is a typical application for it?

6–5 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 the lines with the mov instruction. The AL column represents the AL value after executing the corresponding instruction.

6–6 When subtracting two numbers, suppose the carry flag is set. What does it imply in terms of the relationship between the two numbers?

6–7 In the last example, suppose the overflow flag is set. What does it imply in terms of the relationship between the two numbers?

6–8 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.

6–9 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.

6–10 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.

6–11 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.

6–12 The inc and dec instructions do not affect the carry flag. Explain why it is really not required.

6–13 Suppose the add instruction is not available. Show how we can use the adc instruction to implement the add instruction. Of course, you can use other instructions as well.

6–14 Suppose the adc instruction is not available. Show how we can use the add instruction to implement the adc instruction. Of course, you can use other instructions as well.

6–15 Show how you can implement multiplication by 12 by using four additions. You can use registers for temporary storage.

6–16 What is the use of the neg instruction?

6–17 Show how you can implement the neg instruction with an add instruction.

6–18 Explain why multiplication requires two separate instructions to work on signed and unsigned data.

6–19 We have stated in Section 6.2.4 on page 231 that if we use double-length registers, multiplication does not result in an overflow. Justify this statement for 8-, 16-, and 32-bit operands.

6.8 Progamming Exercises

6–P1 The PutInt8 procedure has used repeated division by 10. Alternatively, you can display an 8-bit number by first dividing it by 100 and displaying the quotient, then dividing the remainder generated by 10 and displaying the quotient and the remainder (in that order). Modify the PutInt8 procedure to incorporate this method. Discuss the pros and cons of the two methods.

6–P2 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 the actual element values. The values of matrix elements should be within the range of the signed 8-bit 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, etc.

6–P3 We know that

$$1 + 2 + 3 + \ldots + N = \frac{N \times (N + 1)}{2}$$

Write a program that requests N as input and computes the lefthand side and the righthand side of the equation and verifies that they are equal and displays the result.

6–P4 Write a program to read a set of test scores as input and that 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 percent. For example, if the average is 18 and the maximum obtainable test score is 20, the percent average is 90 percent.

6–P5 Modify the above program to round the average test score. For example, if the average is 15.55, it should be rounded to 16.

6–P6 Modify the average test score program to display the fractional part as well. Display the average test score in the `dd.dd` format.

6–P7 Write a program to convert temperature from Celsius to Fahrenheit. The formula is

$$F = \frac{9}{5} \times C + 32$$

6–P8 Write a program to read the length L, width W, and height H of a box from input and that displays the volume and surface area of the box.

volume = $L \times W \times H$
surface volume = $2 \times (L \times H + L \times W + W \times H)$

Chapter 7

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
- To explain full and partial evaluation of logical expressions and their performance implications

Modern high-level languages like C and Pascal provide a variety of decision structures. These structures include selection structures such as `if` *constructs and iterative structures such as* `while` *and* `for` *loop constructs.*

Assembly language, being a low-level language, does not provide these structures directly. However, assembly language provides several basic instructions that can be used to construct these high-level language selection and iteration structures. These instructions include the unconditional jump, compare, conditional jump, and loop. A detailed description of these assembly language instructions is given in the first four sections.

Section 7.5 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 7.6, we will describe the indirect jump instruction and its use in implementing multiway switch or case statements in Section 7.7.

Section 7.8 describes the two common methods of evaluating Boolean expressions, and their performance impact is discussed in Section 7.9. The chapter concludes with a summary.

7.1 Unconditional Jump

The unconditional jump (jmp) instruction, as its name implies, 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: *direct* and *indirect*. The vast majority of jumps are of the direct type. Therefore, we focus our attention on direct jump instruction types and briefly discuss indirect jumps in Section 7.7.

Direct Jumps

In direct jump instruction, the target address is specified directly as a part of the instruction. In the following code fragment

```
                .
                .
        mov     CX,10
        jmp     CX_init_done
init_CX_20:
        mov     CX,20
CX_init_done:
        mov     AX,CX
repeat1:
        dec     CX
                .
                .
        jmp     repeat1
                .
                .
```

both the jmp instructions directly specify the target. Recall that as an assembly language programmer, you only specify the target address by using a label and let the assembler figure out the exact value by using its symbol table.

The instruction

```
jmp   CX_init_done
```

transfers control to an instruction that follows it. This is called a *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 that is specified in a jump instruction is not the absolute address of the target instruction. Rather, it specifies the relative displacement in bytes between the target instruction and the instruction following the jump instruction (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 IP register always points to the next instruction to be executed (see Chapter 2). Thus, after fetching the `jmp` instruction, the IP is automatically advanced to point to the instruction following the `jmp` instruction. Execution of `jmp` involves changing the IP from where it is currently pointing to the target instruction location. This is achieved by adding the difference (i.e., relative displacement) to the IP contents. This works fine because the relative displacement is a signed number—a positive displacement implies a forward jump and a negative displacement indicates a backward jump.

The specification of relative address as opposed to the absolute address of the target instruction is appropriate for dynamically relocatable code (i.e., for position-independent code).

The following code

```
forever:  jmp forever
```

is a valid statement that results in an infinite loop. Incidentally, this is a backward jump.

Where Is the Target?

If the target of a jump instruction is located in the same segment as the jump 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:

IP := IP + 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
IP := target-offset

Both target-segment and target-offset are specified directly in the instruction. Thus, for 16-bit segments, the instruction encoding for the intersegment jump takes 5 bytes: one byte for the specification of the opcode, 2 bytes for the target-segment, and another 2 bytes for the target-offset specification.

The majority of jumps are of the intrasegment type. Therefore, Pentium provides 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, can fit in a byte, a jump instruction can be encoded by using just two bytes: one byte for the opcode, and the other for the specification of 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 *short jump*.

If the target is outside this range, 2 or 4 bytes are used to specify the relative displacement. 2-byte displacement is used for 16-bit segments, and 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 *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 CX_init_done
```

The question that naturally arises at this point is: What if the target is not within −128 or +127 bytes? The assembler will inform 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     SHORT repeat1
```

in the example code on page 258 is redundant, as the assembler can automatically select the SHORT jump, if appropriate, for all backward jumps. However, with 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 7.1 *Example encodings of short and near jumps*

```
                              .
                              .
 8 0005  EB 0C             jmp    SHORT CX_init_done
 9 0007  B9 000A           mov    CX,10
10 000A  EB 07 90          jmp    CX_init_done
11                      init_CX_20:
12 000D  B9 0014           mov    CX,20
13 0010  E9 00D0           jmp    near_jump
14                      CX_init_done:
15 0013  8B C1             mov    AX,CX
16                      repeat1:
17 0015  49                dec    CX
18 0016  EB FD             jmp    repeat1
                              .
                              .
84 00DB  EB 03             jmp    SHORT short_jump
85 00DD  B9 FF00           mov    CX, 0FF00H
86                      short_jump:
87 00E0  BA 0020           mov    DX, 20H
88                      near_jump:
89 00E3  E9 FF27           jmp    init_CX_20
                              .
                              .
```

Figure 7.1 Example encoding of jump instructions.

Figure 7.1 shows some example encodings for short and near jump instructions. The forward short jump on line 8 is encoded in the machine language as EB 0C, where EB represents the opcode for the short jump. The relative offset to target `CX_init_done` is 0CH. From the code, it can be seen that this is the difference between the address of the target (address 0013H) and the instruction following the jump instruction on line 9 (address 0007H). Another example of a forward short jump is given on line 84.

The backward instruction on line 18 also uses the short jump form. In this case, the assembler can decide whether the short or near jump is appropriate. The relative offset is given by FDH (= −3D), which is the offset from the instruction following the jump instruction at address 18H to `repeat1` at 15H.

For near jumps, the opcode is E9H, and the relative offset is a 16-bit signed integer. The relative offset of the forward near jump on line 10 is 00D0H, which is equal to 00E3H − 0013H. The relative offset of the backward near jump on line 89 is given by 000DH − 00E6H = FF27H, which is equal to −217D.

The jump instruction encoding on line 10 requires some explanation. Since this is a forward jump and we have not specified that it could be a short jump, assembler reserves 3 bytes for a near jump (the worst case scenario). At the time of actual encoding, the assembler knows the target location and therefore uses the short jump version. Thus, EB 07 represents the encoding, and the third byte is not used and contains some rogue data. □□□□□□

7.2 Compare Instruction

Implementation of decision structures of high-level languages like `if-then` or the `if-then-else` structure in assembly language is a two step process:

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.

The compare (`cmp`) instruction has the format

```
cmp    destination, source
```

which effectively subtracts the source operand from the destination operand but does not affect any of the operands, as shown below:

```
destination - source
```

The flags are updated as if the `sub` operation has been performed. The main purpose of the `cmp` instruction is to update the flags so that a subsequent conditional jump instruction can test these flags.

Example 7.2 *Some examples of the compare instruction*

The four flags that are useful in establishing a relationship (<, ≤, >, etc.) between two integers are CF, ZF, SF, and OF. Table 7.1 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.

Table 7.1 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

Similarly, the OF is set if the result is out of range for signed numbers (which, for this example, is −128D to +127D).

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. □□□□□□

7.3 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.

7.3.1 Jumps Based on Single Flags

The Pentium instruction set provides two conditional jump instructions—one for jumps if the flag tested is set, and the other for jumps when the tested flag is cleared—for each arithmetic flag except the auxiliary flag. These instructions are summarized in Table 7.2.

Table 7.2 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	
	jcxz	jump if CX = 0	CX = 0 (no flags tested)
Testing for carry:			
	jc	jump if carry	CF = 1
	jnc	jump if no carry	CF = 0
Testing for overflow:			
	jo	jump if overflow	OF = 1
	jno	jump if no overflow	OF = 0
Testing for sign:			
	js	jump if (negative) sign	SF = 1
	jns	jump if no (negative) sign	SF = 0
Testing for parity:			
	jp	jump if parity	PF = 1
	jpe	jump if parity is even	
	jnp	jump if not parity	PF = 0
	jpo	jump if parity is odd	

As shown in Table 7.2, 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
```

can be written in the assembly language as

```
        cmp     count,100
```

```
            jz     S1
                   .
                   .
    S1:
            <statement1 code here>
                   .
                   .
```

But our use of jz is not conveying 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 7.2 is the jcxz instruction. This instruction does not test any flag but tests the contents of the CX register for zero. It is often used in conjunction with the loop instruction. Therefore, we postpone our discussion of this instruction to Section 7.4, which discusses the loop instruction.

7.3.2 Jumps Based on Unsigned Comparisons

When comparing two numbers

```
    cmp     num1,num2
```

it is necessary to know whether these numbers num1 and num2 are representing 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 are treating their contents as representing unsigned numbers. This is because, in unsigned representation, AL = 183D and DL = 110D. However, if the contents of AL and DL registers are treated as representing signed numbers, AL < DL as the

Table 7.3 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

AL register is storing a negative number (−73D), and the DL register is storing 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, etc.). There are six possible relationships between two numbers:

$$
\begin{aligned}
\text{num1} &= \text{num2}\\
\text{num1} &\neq \text{num2}\\
\text{num1} &> \text{num2}\\
\text{num1} &\geq \text{num2}\\
\text{num1} &< \text{num2}\\
\text{num1} &\leq \text{num2}
\end{aligned}
$$

For 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 7.3. 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.

Table 7.4 Examples with Snum1 > Snum2

Snum1	Snum2	ZF	OF	SF
56	55	0	0	0
56	−55	0	0	0
−55	−56	0	0	0
55	−75	0	1	1

7.3.3 Jumps Based on Signed Comparisons

The = and ≠ 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 7.5 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 ≠ simply involves testing whether the ZF is set or cleared, respectively. With the singed numbers, establishing < and > relationships is somewhat tricky.

Let us assume that we are executing the cmp instruction

```
cmp     Snum1,Snum2
```

Conditions for Snum1 > Snum2

Table 7.4 shows several examples in which Snum1 > Snum2 holds.

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 7.5, these conditions are 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. Table 7.6 shows several examples in which the Snum1 < Snum2 holds.

Table 7.5 Jumps based on signed comparison

Mnemonic	Meaning	condition tested
je jz	jump if equal jump if zero	ZF = 1
jne jnz	jump if not equal jump if not zero	ZF = 0
jg jnle	jump if greater jump if not less or equal	ZF = 0 and SF = OF
jge jnl	jump if greater or equal jump if not less	SF = OF
jl jnge	jump if less jump if not greater or equal	SF $\neq$ OF
jle jng	jump if less or equal jump if not greater	ZF = 1 or SF $\neq$ OF

Table 7.6 Examples with `Snum1` < `Snum2`

`Snum1`	`Snum2`	ZF	OF	SF
−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 7.5, this is the condition tested by the `jl` conditional jump instruction.

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 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` has *short* as well as *near* versions.

7.4 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 16 bits. The three loop instructions decrement the CX register before testing it for zero. Decrementing CX does not affect any of the flags. The format of these instructions along with the action taken is shown below.

Mnemonic	Meaning	Action

`loop target`	loop	CX := CX − 1
		if CX $\neq$ 0
		jump to target

`loope target`
`loopz target`
	loop while equal
	loop while zero
		CX := CX − 1
		if (CX $\neq$ 0 and ZF = 1)
		jump to target

`loopne target`
`loopnz target`
	loop while not equal
	loop while zero
		CX := CX − 1
		if (CX $\neq$ 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 2 byte encoding with a single byte indicating the relative displacement, which should be within −128 to +127.

The use of `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 7.3 *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    ODH
SIZE   EQU    81
.DATA
buffer  DB  SIZE DUP (?)   ; buffer for string input
.CODE
               .
               .
    mov     BX,OFFSET buffer  ; BX points to buffer
    mov     CX,SIZE           ; buffer size in CX
read_more:
    GetCh   AL
    mov     [BX],AL
    inc     BX
    cmp     AL,CR            ; see if char input 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 CX 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 jcxz provides a remedy for this situation by testing the CX register. The syntax of this instruction is

```
jcxz    target
```

which tests the CX register and if it is zero, control is transferred to the target instruction. Thus, it is equivalent to

```
cmp    CX,0
jz     target
```

except that jcxz does not affect any of the flags, while the cmp/jz combination affects the status flags. If the operand size is 32 bits, we can use the jecxz instruction instead of jcxz. Both instructions, however, use the same opcode E3H. The operand size determines the register—CX or ECX—used.

By using this instruction, the previous example can be written as

```
      mov      BX,OFFSET buffer  ; BX points to buffer
      mov      CX,SIZE           ; buffer size in CX
      jcxz     read_done
read_more:
      GetCh    AL
      mov      [BX],AL
      inc      BX
      cmp      AL,CR           ; see if char input 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    CX
jnz    target
```

Surprisingly, the loop instruction is slower than the corresponding dec/jnz instruction pair. The loop instruction takes five or six clocks depending on whether the jump is taken or not. The `dec/jnz` instruction pair takes only two clocks. Of course, the `loop` instruction is better for program readability.

2. Similarly, the `jcxz` instruction takes five or six clocks, whereas the equivalent

```
cmp    CX,0
jz     target
```

takes only 2 clocks. Thus, for code optimization, these complex instructions should be avoided.

7.5 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.

7.5.1 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 7.4 *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 included to improve readability of the code. The first statement in the then_part is redundant, but Turbo C generates it anyway. □□□□□□

Example 7.5 *An* if *example with an* and *logical operator*

The following code tests whether ch is a lower case 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 Turbo C is (the variable ch is mapped to the DL register):

```
      cmp    DL,'a'
      jb     not_lower_case
      cmp    DL,'z'
      ja     not_lower_case
lower_case:
      mov    AL,DL
      add    AL,224
      mov    DL,AL
not_lower_case:
              .
              .
```

The compound condition is implemented by two pairs of compare and conditional jump instructions. Notice that ch - 32 is implemented as addition of −32. Also, you will see redundancy in the code generated by the compiler. An advantage of writing in assembly language is that we can avoid such redundancies. □□□□□□

Example 7.6 *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:
              .
              .
```

Turbo C maps the variable `index` to the CX register. Also, the code uses the exclusive-or (`xor`) logical operator to zero CX. This logical operator is discussed in Chapter 8. □□□□□□

7.5.2 Iterative Structures

High-level languages like C and Pascal provide several looping constructs. These include `while`, `repeat-until`, and `for` loops. Here we will 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 7.7 *Example* `while` *loop*

Consider the following example code in C:

```
while(total < 700)
  {
    <loop body>
  }
```

Turbo C generates the following assembly language code:

```
      jmp    while_cond
while_body:
               .
      < instructions for
        while loop body >
               .
while_cond:
      cmp    BX,700
      jl     while_body
end_while:
               .
               .
```

The variable `total` is mapped to the BX register. An initial unconditional jump transfers control to `while_cond` to test the loop condition. □□□□□□

Repeat-Until Loop

This is a *post-test 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 7.8 *Repeat-until example*

Consider the following C code:

```
do
  {
    <loop body>
  }
while (number > 0);
```

The Turbo C compiler generates the following assembly language code:

```
loop_body:
                .
        < instructions for
          do-while loop body >
                .
cond_test:
        or     DI,DI
        jg     loop_body
end_do_while:
                .
                .
```

The variable number is mapped to the DI register. To test the loop condition, it uses or rather than the cmp instruction. □□□□□□

For Loop

The for loop is also called the *counting loop* because it iterates a fixed number of times. The Pascal version of the for loop is a simple counting loop, whereas the for loop in C is much more flexible and powerful. Here we consider only the counting for loops.

Example 7.9 *Upward counting* for *loop*

```
for (i = 0; i < SIZE; i++)    /* for (i = 0 to SIZE−1) */
  {
    <loop body>
  };
```

Turbo C generates the following assembly language code:

```
        xor    SI,SI
        jmp    SHORT for_cond
loop_body:
               .
        < instructions for
          the loop body >
               .
        inc    SI
for_cond:
        cmp    SI,SIZE
        jl     loop_body
               .
               .
```

As with the while loop, an unconditional jump transfers control to for_cond to first test the iteration condition before executing the loop body. The counting variable i is mapped to the SI register. □□□□□□

Example 7.10 *Downward counting* for *loop*

```
for (i = SIZE-1; i >= 0; i--)    /* for (i = SIZE−1 downto 0) */
  {
    <loop body>
  };
```

Turbo C generates the following assembly language code:

```
        mov    SI,SIZE-1
        jmp    SHORT for_cond
loop_body:
               .
        < instructions for
          the loop body >
               .
        dec    SI
for_cond:
        or     SI,SI
        jge     loop_body
                .
                .
```

The counting variable i is mapped to the SI register and or is used to test if i has reached zero. □□□□□□

7.6 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 7.11 *Linear search of an array of integers*

In this example, the user is asked to input an array of non-negative integers and then query whether a given number is in the array or not. The program uses a procedure that implements 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 non-negative integers into the array. The user, however, can terminate input by entering a negative number. The loop instruction, with CX initialized to MAX_SIZE (line 28), 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 to test these two conditions for termination of the search loop. The CX is initialized (line 83) to the size of the array. In addition, a compare (line 88) tests if there is a match between the two numbers. If so, the zero flag is set and loopne terminates the search loop. If the number is found, the index of the number is computed (lines 92 and 93) and returned in AX.

Program 7.30 Linear search of an integer array

```
TITLE      Linear search of integer array      LIN_SRCH.ASM
COMMENT |
        Objective: To implement linear search of an integer
                   array; demonstrates the use of loopne.
            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.
.MODEL SMALL
.STACK 100H
.DATA
MAX_SIZE       EQU 100
array          DW  MAX_SIZE DUP (?)
input_prompt   DB  'Please enter input array: '
               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

.CODE
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  input_prompt ; request input array
        mov     BX,OFFSET array
        mov     CX,MAX_SIZE
array_loop:
        GetInt  AX           ; read an array number
        nwln
        cmp     AX,0         ; negative number?
        jl      exit_loop    ; if so, stop reading numbers
        mov     [BX],AX      ; otherwise, copy into array
        inc     BX           ; increment array address
        inc     BX
        loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
        mov     DX,BX        ; DX keeps the actual array size
        sub     DX,OFFSET array  ; DX := array size in bytes
        sar     DX,1         ; divide by 2 to get array size
read_input:
        PutStr  query_number ; request number to be searched for
        GetInt  AX           ; read the number
        nwln
        push    AX           ; push number, size & array pointer
        push    DX
        push    OFFSET 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
        nwln
        cmp     AL,'Y'       ; if response is not 'Y'
        jne     read_input   ; repeat the loop
done:                        ; otherwise, terminate program
        .EXIT
main    ENDP

;------------------------------------------------------------
; 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 AX, are preserved.
;------------------------------------------------------------
linear_search  PROC
       push    BP
       mov     BP,SP
       push    BX            ; save registers
       push    CX
       mov     BX,[BP+4]     ; copy array pointer
       mov     CX,[BP+6]     ; copy array size
       mov     AX,[BP+8]     ; copy number to be searched
       sub     BX,2          ; adjust index to enter loop
search_loop:
       add     BX,2          ; update array index
       cmp     AX,[BX]       ; compare the numbers
```

```
        loopne  search_loop
        mov     AX,0            ; set return value to zero
        jne     number_not_found  ; modify it if number found
        mov     AX,[BP+6]       ; copy array size
        sub     AX,CX           ; compute array index of number
number_not_found:
        pop     CX              ; restore registers
        pop     BX
        pop     BP
        ret     6
linear_search  ENDP
        END   main
```

Example 7.12 *Sorting of an array of integers using the selection sort algorithm*

The main program is very similar to that in the last example, except for the portion that displays the sorted array. The sort procedure receives a pointer to the array to be sorted and its size via the stack. It uses the selection sort algorithm to sort the array in ascending order. The basic idea is as follows:

1. Search the array for the smallest element
2. Move the smallest element to the first position by exchanging values of the first and smallest element positions
3. Search the array for the smallest element from the second position of the array
4. Move this element to position 2 by exchanging values as in step 2
5. Continue this process until the array is sorted.

The selection sort procedure implements the following 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 7.31 implements this pseudocode with the following mapping of variables: `position` is maintained in SI, and DI is used for the index variable `j`. `min_value` is maintained in DX and `min_position` in AX. The number of elements to be searched for finding the minimum value is kept in CX.

Program 7.31 Sorting of an array of integers using the selection sort algorithm

```
TITLE    Sorting an array by selection sort     SEL_SORT.ASM
COMMENT |
        Objective: To sort an integer array using selection sort.
            Input: Requests numbers to fill array.
|          Output: Displays sorted array.
.MODEL SMALL
.STACK 100H
.DATA
MAX_SIZE       EQU 100
array          DW  MAX_SIZE DUP (?)
input_prompt   DB  'Please enter input array: '
               DB  '(negative number terminates input)',0
out_msg        DB  'The sorted array is:',0

.CODE
.486
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  input_prompt ; request input array
        mov     BX,OFFSET array
        mov     CX,MAX_SIZE
array_loop:
        GetInt  AX           ; read an array number
        nwln
        cmp     AX,0         ; negative number?
        jl      exit_loop    ; if so, stop reading numbers
        mov     [BX],AX      ; otherwise, copy into array
        add     BX,2         ; increment array address
        loop    array_loop   ; iterates a maximum of MAX_SIZE
exit_loop:
        mov     DX,BX        ; DX keeps the actual array size
        sub     DX,OFFSET array  ; DX := array size in bytes
        sar     DX,1         ; divide by 2 to get array size
        push    DX           ; push array size & array pointer
        push    OFFSET array
        call    selection_sort
        PutStr  out_msg      ; display sorted array
        nwln
        mov     CX,DX
        mov     BX,OFFSET array
display_loop:
        PutInt  [BX]
        nwln
        add     BX,2
        loop    display_loop
done:
        .EXIT
main    ENDP

;-----------------------------------------------------------
; 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.
;-----------------------------------------------------------
SORT_ARRAY  EQU  [BX]
selection_sort PROC
       pusha                 ; save registers
       mov     BP,SP
       mov     BX,[BP+18]    ; copy array pointer
       mov     CX,[BP+20]    ; copy array size
       sub     SI,SI         ; array left of SI is sorted
sort_outer_loop:
       mov     DI,SI
       ; DX is used to maintain the minimum value and AX
       ; stores the pointer to the minimum value
       mov     DX,SORT_ARRAY[SI]  ; min. value is in DX
       mov     AX,SI         ; AX := pointer to min. value
       push    CX
       dec     CX            ; size of array left of SI
```

```
sort_inner_loop:
        add     DI,2            ; move to next element
        cmp     DX,SORT_ARRAY[DI] ; less than min. value?
        jle     skip1           ; if not, no change to min. value
        mov     DX,SORT_ARRAY[DI] ; else, update min. value (DX)
        mov     AX,DI              ;       & its pointer (AX)
skip1:
        loop    sort_inner_loop
        pop     CX
        cmp     AX,SI           ; AX = SI?
        je      skip2           ; if so, element at SI is its place
        mov     DI,AX           ; otherwise, exchange
        mov     AX,SORT_ARRAY[SI]  ; exchange min. value
        xchg    AX,SORT_ARRAY[DI]  ; & element at SI
        mov     SORT_ARRAY[SI],AX
skip2:
        add     SI,2            ; move SI to next element
        dec     CX
        cmp     CX,1            ; if CX = 1, we are done
        jne     sort_outer_loop
        popa                    ; restore registers
        ret     4
selection_sort ENDP
        END   main
```

7.7 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 7.1 on page 261). 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 either through memory or a general-purpose register. Thus, we can write

```
    jmp    CX
```

if the CX register contains the offset of the target. In indirect jumps, the target offset is the absolute value (unlike 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 7.13 *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 7.32. An input between 0 and 9 is valid. Any other input to the program may cause the system to hang up or crash. The input 0 through 2 produces a simple message to indicate the class selection. Other digit inputs terminate the program.

In order to use the indirect jump, we have to build a jump table of pointers (see lines 11–20). The input digit is converted to act as an index into this table and is used in the indirect jump instruction (line 40). Since the range of the index value is not checked, an input like a produces an index value that is outside the range of the jump table. This can lead to unexpected system behavior. In one of the exercises, you are asked to remedy this problem.

Program 7.32 An example demonstrating the use of the indirect jump

```
 1: TITLE    Sample indirect jump example    IJUMP.ASM
 2: COMMENT |
 3:         Objective: To demonstrate the use of indirect jump.
 4:             Input: Requests a digit character from the user.
 5:                    WARNING: Typing any other character may
 6:                             crash the system!
 7: |          Output: Appropriate class selection message.
 8: .MODEL SMALL
 9: .STACK  100H
10: .DATA
11: jump_table  DW  code_for_0   ; indirect jump pointer table
12:             DW  code_for_1
13:             DW  code_for_2
14:             DW  default_code ; default code for digits 3-9
15:             DW  default_code
16:             DW  default_code
17:             DW  default_code
18:             DW  default_code
19:             DW  default_code
20:             DW  default_code
21:
22: prompt_msg  DB  'Type a character (digits ONLY): ',0
23: msg_0       DB  'Economy class selected.',0
24: msg_1       DB  'Business class selected.',0
25: msg_2       DB  'First class selected.',0
26: msg_default DB  'Not a valid code!',0
```

```

.CODE
INCLUDE  io.mac
main PROC
        .STARTUP
read_again:
        PutStr prompt_msg     ; request a digit
        sub    AX,AX          ; AX := 0
        GetCh  AL             ; read input digit and
        nwln
        sub    AL,'0'         ; convert to numeric equivalent
        mov    SI,AX          ; SI is index into jump table
        add    SI,SI          ; SI := SI * 2
        jmp    jump_table[SI] ; indirect jump based on SI
test_termination:
        cmp    AL,2
        ja     done
        jmp    read_again
code_for_0:
        PutStr msg_0
        nwln
        jmp    test_termination
code_for_1:
        PutStr msg_1
        nwln
        jmp    test_termination
code_for_2:
        PutStr msg_2
        nwln
        jmp    test_termination
default_code:
        PutStr msg_default
        nwln
        jmp    test_termination
done:
        .EXIT
main    ENDP
        END main
```

7.7.1 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 and Pascal provide a special construct for multiway conditional execution. In this section we look at the C switch construct for multiway conditional execution. Pascal provides the case statement for the same purpose.

Example 7.14 *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 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 other case statements.

Turbo C produces the assembly language code shown in Figure 7.2. 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. □□□□□□

```
_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 7.2 Assembly language code for full evaluation.

7.8 Evaluation of logical expressions

Typically, logical expressions involve more than one logical operator. Logical expressions can be evaluated in one of two ways: (1) by full evaluation, or (2) by partial evaluation. Each of these two methods is discussed next.

7.8.1 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 following 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 7.3 for the above logical expression.

7.8.2 Partial Evaluation

The final result of a logical expression can be obtained without completely evaluating it. Two rules aid in this.

1. In an expression of the form

   ```
   cond1 AND cond2
   ```

   the outcome is known to be false if one input is false. For example, if we follow the convention of evaluating the logical expression from left to right, as soon as we know that cond1 is false, we can assign false to the entire logical expression. Only when cond1 is true do we need to evaluate cond2 to know the final value of the logical expression.
2. Similarly, in an expression of the form

   ```
   cond1 OR cond2
   ```

   the outcome is known if cond1 is true. The evaluation can stop at that point. We need to evaluate cond2 only if cond1 is false.

This method of evaluation is used in C. The partial evaluation assembly language code for the previous logical expression (produced either by the Turbo Pascal compiler with an option or by the Turbo C compiler) is shown in Figure 7.4. The code does not use any logical instructions. Instead, the conditional

```
        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 7.3 Assembly language code for full evaluation.

jump instructions are used to implement the logical expression. Partial evaluation clearly results in an efficient code. Section 7.9 discusses performance implications of full and partial evaluation of logical expressions.

Partial evaluation also has an important advantage beyond the obvious reduction in evaluation time. This is illustrated in the following.

Suppose X and Y are inputs to the program. A statement like

```
        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 7.4 Assembly language code for partial evaluation.

```
if ((X > 0) AND (Y/X > 100))
             .
             .
```

can cause a divide-by-zero error if X = 0 when full evaluation is applied. However, with partial evaluation, when X is zero, (X > 0) is evaluated to be 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 not NULL before manipulating the data that it points to.

Of course with full evaluation, we can rewrite the above condition to avoid the divide-by-zero error as

```
if (X > 0)
    if (Y/X > 100)
           .
           .
```

7.9 Performance: Logical Expression Evaluation

Section 7.8 discussed the two common ways of evaluating logical expressions. As we have seen, full logical expression evaluation tends to generate long code, which also takes more time to execute. This section quantifies the performance impact of full and partial logical expression evaluation.

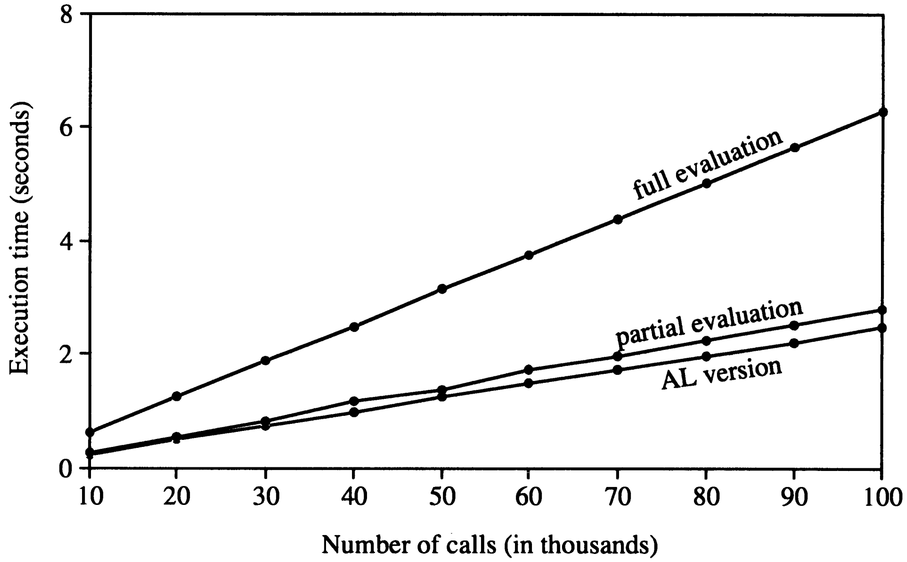


Figure 7.5 Performance impact of logical expression evaluation strategy.

Figure 7.5 shows the performance impact of these two methods. The logical expression evaluated is the same as that given in Section 7.8, which determines if a character is an alpha character. Every time the procedure is called, it evaluates the logical expression for all (128) standard ASCII characters. The x-axis in Figure 7.5 gives the number of times this procedure is called. Clearly, partial evaluation is substantially faster (more than twice as fast) than full evaluation.

For this particular example, we can write assembly language code that is even more efficient than the partial evaluation code shown in Figure 7.4 by observing that if the character code is below that of letter A, we know that it is not an alpha character. The assembly language version is shown below:

```
        cmp     ch,'A'
        jb      skip_if
        cmp     ch,'Z'
        jbe     skip2

        cmp     ch,'a'
        jb      skip_if
        cmp     ch,'z'
        ja      skip_if
skip2:
        << if body here >>
skip_if:
        << code following the if >>
```

The performance of this code is about 12 percent better than that of the partial evaluation code generated by the compiler. This is an example in which the assembly language code is more efficient than the code generated by the compiler. This, of course, is the result of our knowledge about the ASCII encoding scheme.

7.10 Summary

We discussed the jump, compare, and loop instructions in detail. These assembly language instructions are useful in implementing high-level language selection and iteration structures such as `if` and `while` constructs. Through detailed examples, we discussed how these high-level decision structures are implemented in assembly language. We also discussed the indirect jump instruction and its use in implementing multiway conditional statements (such as the `switch` statement in C and the `case` statement in Pascal).

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. Languages like C use partial evaluation, while Pascal uses full evaluation (although with compiler options, we can force the compiler to use partial evaluation). There is significant performance overhead in fully evaluating logical expressions depending on their complexity.

7.11 Exercises

7–1 What is the difference between SHORT and NEAR jumps?

7–2 What is the range of SHORT and NEAR jumps? Explain the reason for this range limit.

7–3 What are forward and backward jumps?

7–4 Why does the assembler need your help for forward near jumps?

7–5 As you know, all conditional jumps are SHORT jumps. How do you handle conditional near jumps?

7–6 Describe the semantics of the `jcxz` instruction and explain how it is useful.

7–7 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	

7–8 In Table 7.3, explain intuitively why the flags tested are necessary and sufficient to implement conditional jumps.

7–9 We have stated on page 268 that to detect Snum1<Snum2, the condition ZF = 0 is not necessary. Justify this statement.

7–10 In Table 7.5, explain intuitively why the flags tested are necessary and sufficient to implement conditional jumps.

7–11 What is the difference between `loop` and `loopne` instructions?

7–12 Compare and contrast direct and indirect jumps.

7–13 What high-level language construct requires the use of the indirect jump for efficient implementation?

7–14 Describe the two methods of evaluating logical expressions.

7–15 Discuss the advantages of partial evaluation over full evaluation of logical expressions.

7–16 Using a debugger (or the -S option with Turbo C), investigate when the indirect instruction is used for the `switch` statement (i.e., for how many cases).

7–17 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.

7.12 Progamming Exercises

7–P1 Modify Program 7.30 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/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.

7–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 7.30 to work on a sorted array. For this exercise, assume that the user supplies the input data in sorted order.

7–P3 In the last exercise, you have 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 7.31 to sort the array once after reading the input data.

7–P4 Modify the indirect jump program given in Program 7.32 so that it works for any input data without the system hanging up or crashing. That is, make the program safe to run.

7–P5 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 to read a positive integer from the user and that displays the single digit as obtained by the above procedure. For the example, the output should be 3.

Your program should detect negative number input as an error and terminate after displaying an appropriate error message.

7–P6 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

186 = 1*6*8 = 48
48 = 4*8 = 32
32 = 3*2 = 6

7–P7 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 an 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 of 1's.

7–P8 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

7–P9 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.

7–P10 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.

7–P11 Merge sort is a technique to combine two sorted arrays. Merge sort takes two sorted input arrays X and Y—say of size m and n—and produces a sorted array Z of size $m + n$ that contains all elements of the two input arrays. The pseudocode of merge sort is as follows:

```
mergesort (X, Y, Z, m, n)
    i := 0 {index variables for arrays X, Y, and Z}
    j := 0
    k := 0
    while ((i < m) AND (j < n))
```

```
            if (X[i] ≤ Y[j]) {find largest of two}
            then
                Z[k] := X[i] {copy and update indices}
                k := k+1
                i := i+1
            else
                Z[k] := Y[j] {copy and update indices}
                k := k+1
                j := j+1
            end if
        end while
        if (i < m) {copy remainder of input array}
            while (i < m)
                Z[k] := X[i]
                k := k+1
                i := i+1
            end while
        else
            while (j < m)
                Z[k] := Y[j]
                k := k+1
                j := j+1
            end while
        end if
    end mergesort
```

The merge sort algorithm scans the two input arrays while copying the smallest of the two elements from X and Y into Z. It updates indices appropriately. The first while loop terminates when one of the arrays is exhausted. Then the other array is copied into Z. Write a merge sort procedure and test it with two sorted arrays. Assume that the user will enter the two input arrays in sorted (ascending) order. The merge sort procedure receives the five parameters via the stack.

Chapter 8

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
- To demonstrate the benefits of using shift instructions for simple multiplication and division operations

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. Assembly language provides several logical instructions to implement logical expressions. These instructions are also useful in implementing bitwise logical operations. Section 8.1 describes the logical instructions available in the Pentium assembly language.

Bit manipulation is an important aspect of any high-level language. The logical instructions discussed in Section 8.1 are useful in bit manipulation. In addition, several shift and rotate instructions are provided to facilitate bit manipulation. Shift instructions are discussed in Section 8.2, while rotate instructions are described in Section 8.3.

Issues related to logical expressions in high-level languages are discussed in Section 8.4. Pentium provides several instructions to test and modify bits and to scan for a bit. These instructions are discussed in Section 8.5. Section 8.6 gives some examples to illustrate the application of logical and shift/rotate instructions. Section 8.7 discusses the performance implications of shift operations for multiplication. The chapter concludes with a summary.

S. P. Dandamudi, *Introduction to Assembly Language Programming*
© Springer Science+Business Media New York 1998

8.1 Logical Instructions

Logical instructions manipulate logical data just like arithmetic instructions manipulate arithmetic data (e.g., integers) with operations such as addition, subtraction, etc. 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 a bit is 0 it represents `false`, and a value of 1 represents `true`.

Most high-level programming languages provide the three basic logical operators: `and`, `or`, and `not`. Some also provide an *exclusive-or* (`xor`) operation. In this context, the normal `or` operator is called *inclusive-or*, and we will shortly see the differences between these two `or` operations.

Assembly language provides all of these logical operators in the logical family of instructions. There are 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 flag (CF) and the overflow flag (OF) 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. We next look at each of these instructions in turn.

8.1.1 The and Instruction

The logical `and` instruction is a binary operator with the following general format:

```
and    destination,source
```

The usual rules that govern the specification of destination and source operands apply to these instructions as well. It goes without saying explicitly, then, that the destination and source can both be 8-, 16-, or 32-bit operands, both can be in general-purpose registers, or one can be in memory.

The logical `and` operation sets the destination bit according to the truth table shown below:

Table 8.1 Some examples of and   AL,BL

Before execution of and		After execution of and			
AL	BL	AL	ZF	SF	PF
0101 1110	1110 0001	0100 0000	0	0	0
0110 1011	1001 0100	0000 0000	1	0	1
1111 1111	1111 1111	1111 1111	0	1	1
1001 0110	0110 1001	0000 0000	1	0	1

Truth table for the 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

The output bit is 1 if and only if the corresponding input bits are 1.

Table 8.1 gives some examples of how the instruction

```
and     AL,BL
```

works. To find the value of an output bit, look at the corresponding two input bits and use the truth table for the and operation. The key point is that the bits are treated individually even though they are together in a byte or word. It is interesting to note that ZF = 1 implies that SF = 0 and PF = 1.

Usage

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 8.4. Here we will 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 301), you will notice that the output bit is 0 whenever source b_i is 0, irrespective of the value of the other input bit. When the source b_i is 1, the output bit is identical to the destination b_i input. Thus, the source b_i is acting 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 2 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 2-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 8.1 *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 bits in the whole byte even.

If the number of 1 bits in the least significant 7 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 7 bits. Notice that the mask 7FH (01111111B) has a 0 only in the parity bit position. □□□□□□

Example 8.2 *ASCII-to-numeric conversion of digit characters*

In this example, we want to convert an ASCII decimal digit character to its equivalent 8-bit binary number. To see how this can be done by using the

Table 8.2 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

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 8.2.

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 4 bits without worry, which is what the following code does. If AL has the ASCII code of a decimal digit,

```
and     AL,0FH
```

would produce the desired result in AL. □□□□□□

Isolating Bits

Another typical use of the and instruction is to isolate selected bit(s) for testing. This is done by masking out all the other bits, as shown in the next example.

Example 8.3 *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. □□□□□□

8.1.2 The or Instruction

The or instruction has the general format

```
or      destination,source
```

and the operation of or is shown in the form of the truth table below:

Truth table for the 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

The or operation is a dual of and in the sense that the output is 0 if and only if the both the inputs are 0. Table 8.3 gives some examples of how the instruction

```
or      AL,BL
```

works.

Table 8.3 Some examples of `or AL,BL`

Before execution of or		After execution of or			
AL	BL	AL	ZF	SF	PF
0100 0010	0000 1010	0100 1010	0	0	0
0000 0000	0000 0000	0000 0000	1	0	1
1010 0000	0000 1111	1010 1111	0	1	1
1000 0000	1000 0000	1000 0000	0	1	0

Usage

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 8.4. 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, 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 2 bits to change to 1. Here are some examples that illustrate the use of the `or` instruction.

Example 8.4 *Even-parity encoding (partial code)*

Consider the even-parity encoding discussed in Example 8.1 (on page 302). If the number of 1 bits in the least significant 7 bits is odd, we have to make the parity bit 1 so that the total number of 1 bits 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 8.5 *Conversion of digits to ASCII characters*

This is the counterpart of Example 8.2 on page 302. 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 conversion is often required to print or display numbers. Refer to Table 8.2 on page 303.

The conversion process involves making the third and fourth bits (from left) of the binary number 1's. 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 2 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 the use of the mask 55H (01010101B). The second and instruction selects the even bits by using the mask AAH (10101010B). The or instruction simply pastes these 2 bytes together to produce the desired byte in the AL register.

8.1.3 The xor Instruction

The general format of the exclusive-or (xor) instruction is

```
xor    destination,source
```

The truth table for the xor operation is given below:

Table 8.4 Some examples of `xor AL,BL`

Before execution of or		After execution of or			
AL	BL	AL	ZF	SF	PF
0100 0010	0001 1010	0101 1000	0	0	0
0110 1101	0110 1101	0000 0000	1	0	1
1010 0110	0101 1001	1111 1111	0	1	1
1011 0110	0000 1111	1011 1001	0	1	0

Truth table for the 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

Table 8.4 gives some examples of how the instruction

```
xor     AL,BL
```

works.

If you compare this truth table with that of the or operation on page 304, you will notice that they differ only in the last row when both the inputs are 1. This apparently minor change between or and xor gives xor an interesting property—to selectively complement bits.

The truth table shows that 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 the complement of the other input. Thus, by using the appropriate mask, selected bits of an input can be inverted. The following example illustrates how the xor operation can be used to invert the odd bits of the AL register.

```
              AL  = 11010110B  ← operand to be manipulated
              BL  = 01010101B  ← mask byte
xor       AL,BL  = 10000011B
```

Since the mask is 55H, all odd bits of AL are flipped while copying the even bits to the output. Thus, in some sense, the xor operation can be thought of as a bit-specific not operation. In particular, if the mask has all 1's, the xor performs like the not operation (except for differences in how the flags are affected).

Usage

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 compound logical expression is discussed in Section 8.4. 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 a 1 bit in the bit positions that are to be flipped. The following example illustrates this application of the `xor` instruction.

Example 8.6 *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:

$$\begin{array}{rl} & \texttt{01000001B} \leftarrow \text{even-parity encoded ASCII character A} \\ \text{xor} & \underline{\texttt{10000000B}} \leftarrow \text{mask byte} \\ & \texttt{11000001B} \leftarrow \text{odd-parity encoded ASCII character A} \end{array}$$

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: `xor`ing 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 8.7 *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
<u>00100110B</u> ← 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 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 307. 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.

8.1.4 The not Instruction

The logical not operation is a unary operator (i.e., it requires only one operand). The general format is

```
not     destination
```

The not operation complements or flips the bits, as shown in the following truth table.

Truth table for the not operation

Input bit	Output bit
destination b_i	destination b_i
0	1
1	0

As an example, if AL contains 00110101B before executing

```
not     AL
```

the AL register will have 11001010B after executing the not instruction. One important difference between not and the other logical instructions is that not does not affect any of the flags.

Usage

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 8.4.

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 2's complement of an 8-bit signed number using not involves

```
not     AL
inc     AL
```

However, the Pentium instruction set also provides the neg instruction to reverse the sign of a number. Thus, the not instruction is not useful for this purpose.

8.1.5 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. The general format is

```
test    destination,source
```

Usage

This instruction is used only to update the flags, and a conditional jump instruction normally follows it. For instance, in Example 8.3 on page 303, 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:
               .
        <code for processing odd number>
               .
even_number:
               .
        <code for processing even number>
               .
```

8.2 Shift Instructions

Pentium provides two types of shift instructions: one type for performing logical shifts, and the other for performing 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 work 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. The most often tested flags after a shift instruction are CF and ZF.

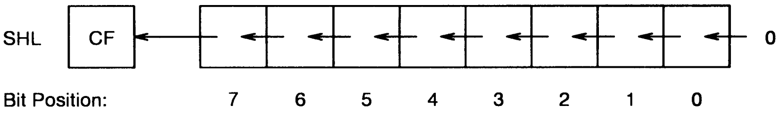
8.2.1 Logical Shift Instructions

The logical shift instructions have the following general format:

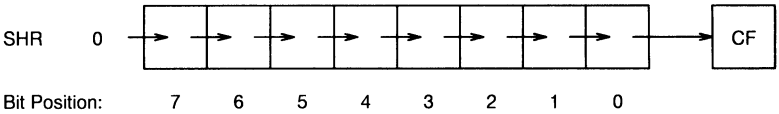
```
shl   destination,count
shr   destination,count
```

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 destination is shifted left or right by count bit positions.

The shl instruction can be used to left shift a destination operand. Each bit shift to the left causes the leftmost bit to move to the carry flag (CF), and the vacated rightmost bit is filled with a 0 bit. The bit that was in CF is lost as a result of the left shift operation.



The shr instruction works similarly but shifts bits to the right.



There are two versions of each instruction depending on how the count value is specified. The count can be given as an immediate value or in the CL register. Thus, the formats of these instructions are,

```
shl  destination,count      shr  destination,count
shl  destination,CL         shr  destination,CL
```

The first format can be used to specify directly the number of bit positions to be shifted. The value of 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 the shl and shr instructions. The CL versions are useful when we don't know the count value (e.g., given as a parameter in a procedure call). 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 0 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 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.

Here are some examples of shl and shr instructions.

Instruction	Before shift	After shift	CF
`shl AL,1`	AL = 1010 1110	AL = 0101 1100	1
`shr AL,1`	AL = 1010 1110	AL = 0101 0111	0

`mov CL,3`
`shl AL,CL`
	AL = 0110 1101	AL = 0110 1000	1

`mov CL,5`
`shr AX,CL`
	AX =
	1011 1101 0101 1001
		AX =
		0000 0101 1110 1011
			1

Usage

The shift 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 bit manipulation as illustrated by the following example.

Example 8.8 *Another encryption example*

Consider the encryption example discussed on page 308. 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 308.

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
```

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 shl and shr instructions. The shl instruction replaces the shifted bits by 0's and after the shl

```
AL = 00010000B
```

Similarly, shr also introduces 0's in the vacated bits on the left and after the shr instruction

```
AH = 00000100B
```

The or instruction pastes these 2 bytes together, as shown below:

```
            AL  = 00010000B
            AH  = 00000100B
                  ---------
or     AL,AH  = 00010100B
```

We will show in Section 8.3.1 that this can be done better by using a rotate instruction (see Example 8.9 on page 320). □□□□□□

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,

Table 8.5 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

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 8.5. 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 3 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, discarding any fractional part of the result.

8.2.2 Arithmetic Shift Instructions

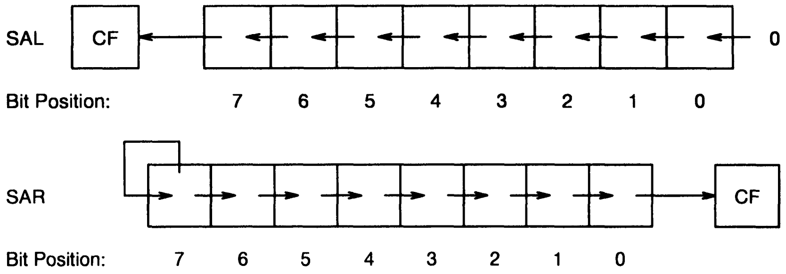
This set of shift instructions

`sal` (Shift Arithmetic Left)
`sar` (Shift Arithmetic Right)

can be used to shift signed numbers left or right, as shown below.

Table 8.6 Doubling of signed numbers

Signed binary number	Decimal value
00001011	+11
00010110	+22
00101100	+44
01011000	+88
11110101	-11
11101010	-22
11010100	-44
10101000	-88



Shifting left by one bit position corresponds to doubling the number, and shifting right by one bit position corresponds to halving it. As with the logical shift instructions, the CL register can be used to specify the count value. The general format is

```
sal    destination,count        sar    destination,count
sal    destination,CL           sar    destination,CL
```

Doubling Signed Numbers

Doubling a signed number by shifting it left by one bit position may appear to cause problems because the leftmost bit is used to represent the sign of the number. It turns out that this is not a problem at all. See the examples presented in Table 8.6 to develop your intuition. The first group presents the doubling effect on positive numbers and the second group shows the doubling effect on negative numbers. In both cases, the vacated bit is replaced by a 0. Why isn't shifting out the sign bit causing problems? The reason is that signed numbers are sign-extended to fit a larger-than-required number of bits. For example, if we want to represent numbers in the range of $+3$ and -4, 3 bits are sufficient to

Table 8.7 Division of signed numbers by 2

Signed binary number	Decimal value
01011000	+88
00101100	+44
00010110	+22
00001011	+11
10101000	−88
11010100	−44
11101010	−22
11110101	−11

represent this range. If we use a byte to represent the same range, the number is sign-extended by copying the sign bit into the higher order 5 bits, as shown below.

$$+3 = \overbrace{00000}^{\substack{\textit{sign bit}\\ \textit{copied}}}011\text{B}$$

$$-3 = \overbrace{11111}^{\substack{\textit{sign bit}\\ \textit{copied}}}101\text{B}$$

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 of the number. This rules out the use of `shr` for signed numbers. See the examples presented in Table 8.7. The `sar` instruction precisely does this—the sign bit is copied into the vacated bit on the left.

Remember that the shift right operation performs integer division. For example, right shifting 00001011B (+11D) by a bit results in 00000101B (+5D).

8.2.3 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

```
; multiplicand is assumed to be in AX
mov    CX,32   ; multiplier in CX
mul    CX
```

These two instruction sequences take twelve clock cycles. Of this, mul takes about eleven 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. Section 8.7 discusses the performance impact of these two versions.

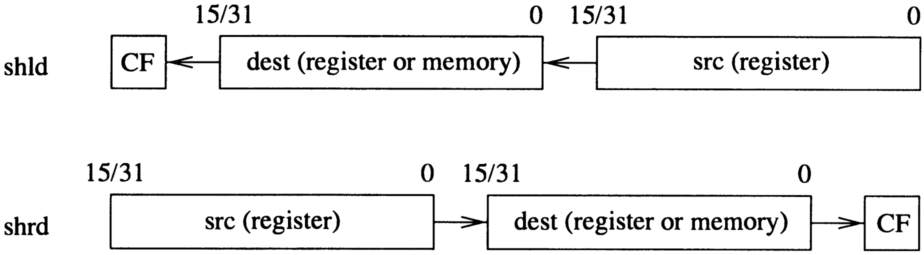
8.2.4 Double Shift Instructions

Pentium provides two double shift instructions for 32-bit and 64-bit shifts. These two instructions operate on either word or doubleword operands and produce a single word or doubleword result, respectively. The double shift instructions require three operands, as shown below:

```
shld    dest,src,count  ; left shift
shrd    dest,src,count  ; right shift
```

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 double shift instructions is that the src operand supplies the bits in double shift instructions, as shown below:



Note that the bits shifted out of the src operand go into the dest operand. However, the src operand itself is not modified by the double shift instructions. Only the dest operand is updated appropriately. As in the shift instructions, the last bit shifted out is stored in the carry flag. Later we present an example that demonstrates the use of the double shift instructions (see Example 8.10 on page 321).

8.3 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 double shift instructions provide this capability on word or doubleword operands. The rotate family of instructions are useful in remedying 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. The next two subsections consider these two types of instructions.

8.3.1 Rotate Without Carry

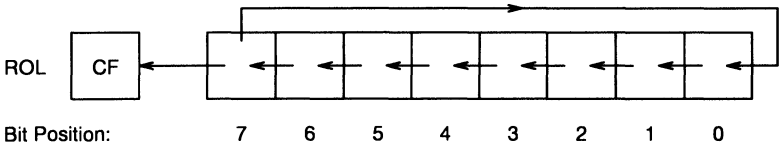
There are two instructions in this group:

rol (ROtate Left)
ror (ROtate Right)

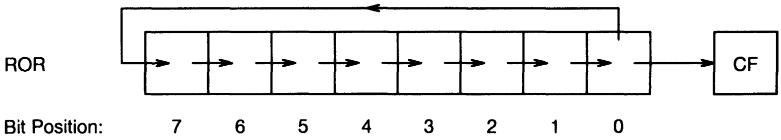
The format of these instructions is similar to the shift instructions and is given below:

```
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 being placed on the right side, as shown below:



The `ror` instruction performs right rotation, as shown below:



For both of these instructions, the CF will catch the last bit rotated out of destination. The following table illustrates the rotate operation by means of 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

Usage

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 313.

Example 8.9 *Encryption example revisited*

In Example 8.8, we have 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
```

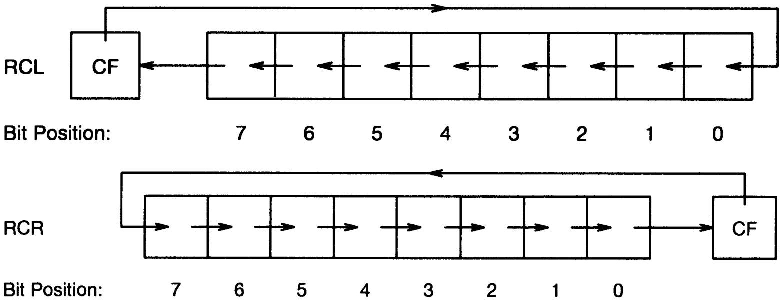
This is a much simpler solution than the one using shifts. □□□□□□

8.3.2 Rotate Through Carry

The instructions

`rcl` (Rotate through Carry Left)
`rcr` (Rotate through Carry Right)

include the carry flag in the rotation process. That is, the bit that is rotated out at one end goes into the carry flag, and the bit that was in the carry flag is moved into the vacated bit, as shown below.



In this context, it is useful to recall that there are three instructions that explicitly manipulate the carry flag: `stc`, `clc`, and `cmc`.

The following table contains several examples that illustrate the operation of the `rcl` and `rcr` instructions.

	Before execution		After execution	
Instruction	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

Usage

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 8.10 *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 double shift instructions for such multiword shifts. As an example, assume that we want to multiply a 64-bit unsigned number by 16. The 64-bit number is assumed to be in the EDX:EAX register pair with EAX holding the least significant 32 bits.

Rotate version:

```
        mov     CX,4        ; 4 bit shift
    shift_left:
        shl     EAX,1       ; moves leftmost bit of AX to CF
        rcl     EDX,1       ; CF goes to rightmost bit of DX
        loop    shift_left
```

Double shift version:

```
        shld    EDX,EAX,4 ; EAX is unaffected by shld
        shl     EAX,4
```

Similarly, if we want to divide the same number by 16, we can use the following code:

Rotate version:

```
        mov     CX,4        ; 4 bit shift
    shift_right:
        shr     EDX,1       ; moves rightmost bit of DX to CF
        rcr     EAX,1       ; CF goes to leftmost bit of AX
        loop    shift_right
```

Double shift version:

```
        shrd    EAX,EDX,4 ; EDX is unaffected by shld
        shr     EDX,4
```

□□□□□□

8.4 Logical Expressions in High-Level Languages

Some high-level languages like Pascal provide Boolean data types. Boolean variables can assume either a true or a false value. Other languages like C do not explicitly provide Boolean data types. This section discusses Boolean data representation and the evaluation of compound logical expressions.

8.4.1 Representation of Boolean Data

In principle, only a single bit is needed to represent the Boolean data. However, such a representation, while compact, is not convenient, as testing a variable involves isolating the corresponding bit.

Most languages use a byte to represent the Boolean data. If all bits of the byte are zero, it represents false; otherwise, true. It is strictly unnecessary to have all bits 1 to represent true.

In C language, which does not provide an explicit Boolean data type, any data variable can be used in a logical expression to represent Boolean data. The same rules mentioned above apply: if the value is 0, it is treated as false and any non-zero value is treated as true. Thus, we can use integer variables as Boolean variables in logical expressions, for example.

8.4.2 Logical Expressions

The logical instructions are useful in implementing logical expressions of high-level languages. For example, C provides the four logical operators discussed in Section 8.1.

C operator	meaning
&&	AND
||	OR
^	exclusive-OR
~	NOT

To illustrate the use of logical instructions in implementing logical expressions of high-level languages, 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 8.1. As we have seen in Chapter 7, C uses partial evaluation of logical expressions.

```
 1:         cmp    WORD PTR [BP-12],0  ; X = false?
 2:         je     false1              ; if so, (X && Y) := false
 3:         or     CX,CX               ; Y = false?
 4:         je     false1
 5:         mov    AX,1                ; (X && Y) := true
 6:         jmp    SHORT skip1
 7:  false1:
 8:         xor    AX,AX               ; (X && Y) := false
 9:  skip1:
10:         not    AX                  ; AX := ~(X && Y)
11:         push   AX                  ; save ~(X && Y)
12:         ; now evaluate the second term
13:         or     CX,CX               ; Y = true?
14:         jne    true2               ; if so, (Y || Z) := true
15:         cmp    WORD PTR [BP-14],0  ; Z = false?
16:         je     skip2
17:  true2:
18:         mov    AX,1                ; (X || Y) := true
19:         jmp    SHORT skip3
20:  skip2:
21:         xor    AX,AX               ; (X || Y) := false
22:  skip3:
23:         pop    DX                  ; DX := ~(X && Y)
24:         xor    DX,AX               ; ~(X && Y) ^ (Y || Z)
25:         je     end_if              ; if zero, whole exp. false
26:  if_body:
27:         mov    AX,CX               ; AX := Y
28:         add    AX,WORD PTR [BP-14] ; AX := Y + Z
29:         mov    WORD PTR [BP-12],AX ; X := Y + Z
30:  end_if:
31:                .
32:                .
```

Figure 8.1 Assembly language code for the example logical expression.

The variable X is mapped to [BP−12], Y to the CX register, and Z to [BP−14]. The code on lines 1–8 implements partial evaluation of (X && Y). 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 13–21 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).

8.4.3 Bit Manipulation

Some high-level languages like C provide bitwise logical operators. For example, C provides bitwise `and` (`&`), `or` (`|`), `xor` (`^`), and `not` (`~`) operators. These can be implemented by using the logical instructions provided in assembly language.

The C language also provides shift operators: left shift (<<) and right shift (>>). These operators can be implemented with the shift instruction of assembly language.

Table 8.8 shows how the logical and shift family of instructions are used in implementing the bitwise logical and shift operators of the C language. The variable `mask` is mapped to the SI register.

8.5 Bit Instructions

Pentium provides 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 8.12).

8.5.1 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
```

Table 8.8 Examples of bitwise operators

C statement	Assembly language code
`mask = mask>>2` (right shift mask by two bit positions)	`shr SI,2`
`mask = mask<<4` (left shift mask by four bit positions)	`shl SI,4`
`mask = ~mask` (complement mask)	`not SI`
`mask = mask & 85` (bitwise and)	`and SI,85`
`mask = mask | 85` (bitwise or)	`or SI,85`
`mask = mask ^ 85` (bitwise xor)	`xor SI,85`

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.

8.5.2 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
brf    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 brf). These two instructions affect only the zero flag. The other five status flags are undefined following a bit scan instruction.

8.6 Illustrative Examples

This section presents three examples that use the shift and rotate family of instructions.

Example 8.11 *Multiplication using only shifts and adds*

The objective of this example is to show how multiplication can be done entirely by shift and add operations. We consider multiplication of two unsigned 8-bit numbers. In order to use the shift operation, we have to express the multiplier as a power of 2. For example, if the multiplier is 64, the result can be obtained by shifting the multiplicand left by six bit positions (because $2^6 = 64$).

What if the multiplier is not a power of 2? In this case, we have to express this number as a sum of powers of 2. For example, if the multiplier is 10, it can be expressed as 8+2, where each term is a power of 2. Then the required multiplication can be done by two shifts and one addition.

The question now is: How do we express the multiplier in this form? If we look at the binary representation of the multiplicand (10D = 00001010B), there is a 1 in bit positions with weights 8 and 2. Thus, for each 1 bit in the multiplier, the multiplicand should be shifted left by a number of positions equal to the bit position number. In the above example, the multiplicand should be shifted left by 3 and 1 bit positions and then added. This procedure is formalized in the following algorithm.

```
mult8 (number1, number2)
    result := 0
    for ($i$ = 7 downto 0)
        if (bit(number2, $i$) = 1)
            result := result + number1 * $2^i$
        end if
    end for
end mult8
```

The function bit returns the *i*th bit of number2. The program listing is given in Program 8.33.

Program 8.33 Multiplication of two 8-bit numbers using only shifts and adds

```
TITLE   8-bit multiplication using shifts   SHL_MLT.ASM
COMMENT |
        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.
.MODEL SMALL
.STACK 100H
.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
INCLUDE io.mac
main    PROC
        .STARTUP
read_input:
        PutStr  input_prompt ; request two numbers
        GetInt  AX           ; read the first number
        nwln
        GetInt  BX           ; read the second number
        nwln
        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
        nwln
        cmp     AL,'Y'       ; if response is not 'Y'
        jne     read_input   ; repeat the loop
done:                        ; otherwise, terminate program
        .EXIT
main    ENDP

;-----------------------------------------------------------
; mult8 multiplies two 8-bit unsigned numbers passed on to
```

```
39:  ; it in registers AL and BL. The 16-bit result is returned
40:  ; in AX. This procedure uses only SHL instruction to do the
41:  ; multiplication. All registers, except AX, are preserved.
42:  ;-------------------------------------------------------------
43:  mult8  PROC
44:         push   CX             ; save registers
45:         push   DX
46:         push   SI
47:         xor    DX,DX          ; DX := 0 (keeps mult. result)
48:         mov    CX,7           ; CX := # of shifts required
49:         mov    SI,AX          ; save original number in SI
50:  repeat1:      ; multiply loop - iterates 7 times
51:         rol    BL,1           ; test bits of number2 from left
52:         jnc    skip1          ; if 0, do nothing
53:         mov    AX,SI          ; else, AX := number1*bit weight
54:         shl    AX,CL
55:         add    DX,AX          ; update running total in DX
56:  skip1:
57:         loop   repeat1
58:         rol    BL,1           ; test the rightmost bit of AL
59:         jnc    skip2          ; if 0, do nothing
60:         add    DX,SI          ; else, add number1
61:  skip2:
62:         mov    AX,DX          ; move final result into AX
63:         pop    SI             ; restore registers
64:         pop    DX
65:         pop    CX
66:         ret
67:  mult8  ENDP
68:         END  main
```

The main program requests two numbers from the user and calls the procedure mult8 and displays the result. The main program then queries the user whether to quit and proceeds according to the response.

The mult8 procedure multiplies two 8-bit unsigned numbers and returns the result in AX. It follows the algorithm discussed on page 327. The multiply loop (lines 50–57) tests the most significant 7 bits of the multiplier. The least significant bit is tested on line 58. Notice that the procedure uses rol rather than shl to test each bit (lines 51 and 58). The use of rol automatically restores the BL register after 8 rotates. □□□□□□

Example 8.12 *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 50–57) to test each bit. Since we are interested only in 1 bits, we can use a bit scan instruction to do this job. The modified mult8 procedure is shown below.

```
 1:  ;-----------------------------------------------------------
 2:  ; mult8 multiplies two 8-bit unsigned numbers passed on to
 3:  ; it in registers AL and BL. The 16-bit result is returned
 4:  ; in AX. This procedure uses only SHL instruction to do the
 5:  ; multiplication. All registers, except AX, are preserved.
 6:  ; Demonstrates the use of bit instructions BSF and BTC.
 7:  ;-----------------------------------------------------------
 8:  mult8  PROC
 9:         push    CX              ; save registers
10:         push    DX
11:         push    SI
12:         xor     DX,DX           ; DX := 0 (keeps mult. result)
13:         mov     SI,AX           ; save original number in SI
14:  repeat1:
15:         bsf     CX,BX           ; returns first 1 bit position in CX
16:         jz      skip1           ; if ZF=1, no 1 bit in BX - done
17:         mov     AX,SI           ; else, AX := number1*bit weight
18:         shl     AX,CL
19:         add     DX,AX           ; update running total in DX
20:         btc     BX,CX           ; complement the bit found by BSF
21:         jmp     repeat1
22:  skip1:
23:         mov     AX,DX           ; move final result into AX
24:         pop     SI              ; restore registers
25:         pop     DX
26:         pop     CX
27:         ret
28:  mult8  ENDP
```

The modified loop (lines 14–21) replaces the loop in the previous version. This code is more efficient because the number of times the loop iterates is equal to the number of 1 bits in BX. The previous version, on the other hand, always iterates seven times. Also note that we can replace the `btc` instruction on line 20 by a `btr` instruction. Similarly, the `bsf` instruction on line 15 can be replaced by a `brf` instruction. □□□□□□

Example 8.13 *Conversion of octal to binary*

An algorithm for converting octal numbers to binary is given in Chapter 2. 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 8.34. Note that we use the `shl` instruction to multiply by 8 (line 57). The rest of the code follows the pseudocode. The next section discusses the improvement in performance due to the use of the `shl` instruction for multiplication over the `mul` instruction.

Program 8.34 Octal-to-binary conversion

```
1:  TITLE   Octal-to-binary conversion using shifts   OCT_BIN.ASM
2:  COMMENT |
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:  .MODEL SMALL
9:  .STACK 100H
```

```
.DATA
octal_number   DB  4 DUP (?) ; to store octal number
input_prompt   DB  'Please input an octal number: ',0
out_msg1       DB  'The decimal value is: ',0
query_msg      DB  'Do you want to quit (Y/N): ',0

.CODE
INCLUDE io.mac
main  PROC
      .STARTUP
read_input:
      PutStr  input_prompt    ; request an octal number
      GetStr  octal_number,4  ; read input number
      nwln
      mov     BX,OFFSET 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
      nwln
      cmp     AL,'Y'       ; if response is not 'Y'
      jne     read_input   ; read another number
done:                      ; otherwise, terminate program
      .EXIT
main  ENDP

;-------------------------------------------------------------
; to_binary receives a pointer to an octal number string in
; BX 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     PROC
      push    BX           ; save registers
      push    CX
      push    DX
      xor     AX,AX        ; result := 0
      mov     CX,3         ; max. number of octal digits
repeat1:
      ; loop itarates a maximum of 3 times;
      ; but a NULL can terminates it early
      mov     DL,[BX]      ; 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     BX            ; move to next octal digit
        loop    repeat1       ; and repeat
finished:
        pop     DX             ; restore registers
        pop     CX
        pop     BX
        ret
to_binary       ENDP
        END     main
```

8.7 Performance: Shift Versus Multiplication

We have seen that shift can be used to perform multiplication and division. Example 8.11 has given a detailed algorithm to perform general multiplication by using only shift and add operations. In this section, we discuss the performance tradeoffs of shift and multiply instructions.

Experiment 1

In this experiment, we use Example 8.11 to see if the shift operation can be used as a general mechanism to perform multiplication. We consider multiplication of two unsigned 8 bit numbers (255D×128D).

As you can see, multiplier 128D (10000000B) has only a single 1 in its binary representation. The general shift algorithm given in Example 8.11, however, tests each of the 8 bits while performing shift only once. The overhead is substantial, as shown in Figure 8.2 (see "general shift" line).

The use of the `mul` instruction reduces execution time substantially (see the "multiply" line). Since 128D is a power of 2, we can perform the required multiplication by a single shift operation (shifting left 255 by seven bit positions). This operation reduces the execution time even further (see the "single shift" line). In fact, the single shift version is about twice as fast as the `mul` version.

The data presented here demonstrates that the shift should be used for multiplication only if we know that the multiplier is a power of 2. The octal-to-binary conversion is an example where the multiplier is a power of 2. The next experiment looks at the performance of this conversion.

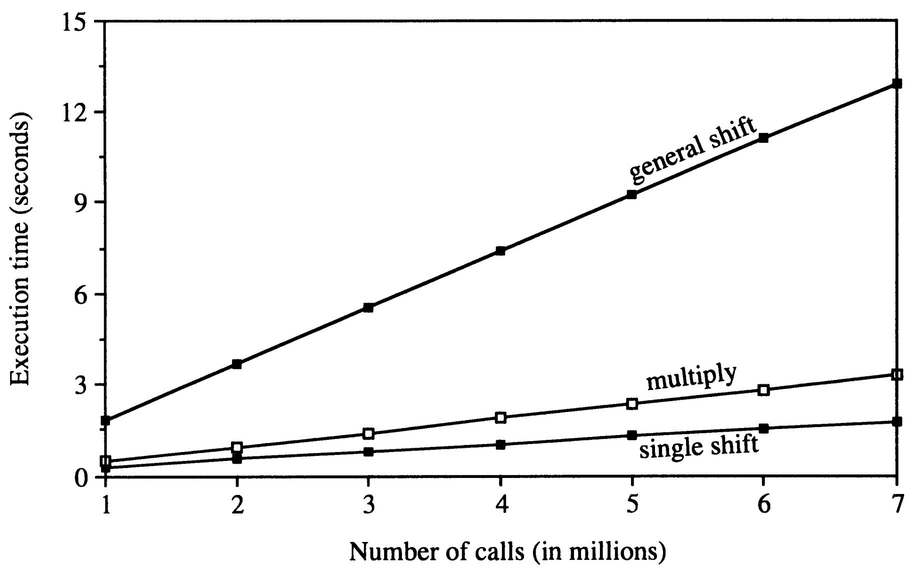


Figure 8.2 Relative performance of `shl` and `mul` for multiplying two 8-bit numbers.

Experiment 2

Conversion from octal to binary involves repeated multiplication by 8. This multiplication can be done efficiently by the `shl` instruction. Another common example is the conversion from hexadecimal to binary, which involves the multiplication by 16. Figure 8.3 shows the performance of the `shl` and `mul` versions of the `to_binary` procedure described in Example 8.13. The execution time of the `mul` version is given by the "multiply" line, and that of the `shl` version is given by the "shift" line. The `shl` version is about 14 percent faster. Clearly, this is an example where `shl` can be used to generate efficient assembly language code.

8.8 Summary

We discussed logical, shift, and rotate instructions available in the Pentium 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.

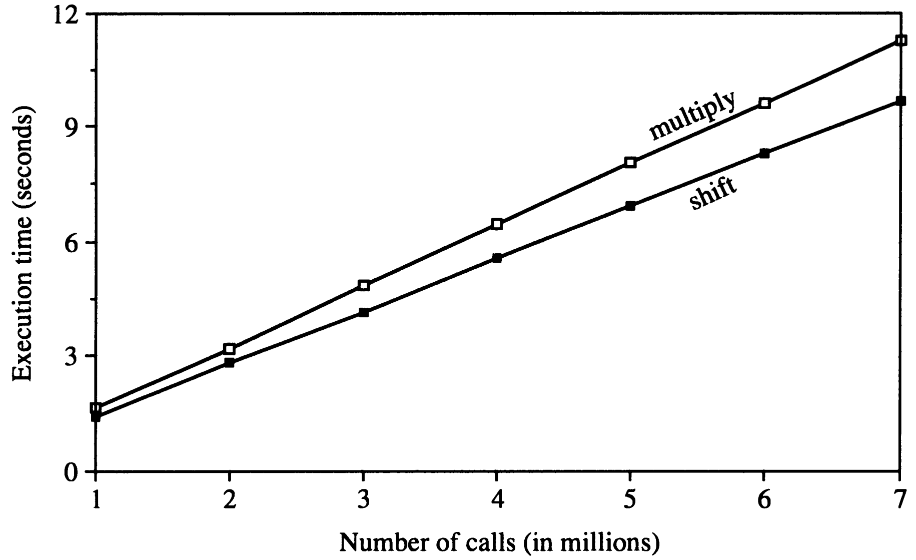


Figure 8.3 Performance of `mul` and `shl` versions for binary conversion.

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.

Pentium also provides two double shift instructions that work on either word or doubleword operands. In addition, four bit instructions are available for testing and modifying bits, and two instructions for scanning 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.

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. We also demonstrated that shift instructions can be profitably used for simple multiplication and division.

8.9 Exercises

8–1 Page 301 stated that ZF = 1 implies that SF = 0 and PF = 1. Explain why.

8–2 What is the difference between or and xor logical operators?

8–3 Logical and operation can be implemented by using only or and not operations. Show how this can be done. You can use as many or and not operations as you want. But try to implement by using only three not and one or operation.

8–4 Logical or operation can be implemented by using only and and not operations. Show how this can be done. You can use as many and and not operations as you want. But try to implement by using only three not and one and operation.

8–5 Explain how and and or logical operations can be used to "cut and paste" a specific set of bits.

8–6 Suppose the instruction set did not support the not instruction. How do you implement it using only and and or instructions?

8–7 Can we use the logical shift instructions shl and shr on signed data?

8–8 Can we use the arithmetic shift instructions sal and sar on unsigned data?

8–9 Give assembly language program fragment to copy low-order 4 bits from the AL register and higher-order 4 bits from the AH register into the DL register. You should accomplish this using only the logical operations of Pentium.

8–10 Repeat the above exercise using only the shift/rotate operations of the Pentium instruction set.

8–11 Show the assembly language program fragment to complement only the odd bits of the AL register using only the logical operations.

8–12 Repeat the above exercise using only the shift/rotate operations of the Pentium instruction set.

8–13 Explain the difference between bitwise and and logical and operations. Use an example to illustrate your point.

8–14 Repeat the above exercise for the or operation.

8–15 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				

8–16 Assuming that the contents of the AL register is treated as a signed number, fill in the blanks in the following table:

	Before execution		After execution	
Instruction	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		

8–17 Assuming that the CL register is initialized to 3, fill in the blanks in the following table:

	Before execution		After execution	
Instruction	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	?		

8–18 In Chapter 7, we discussed how the flags ZF, OF, and SF can be used to establish a relationship such as < and > between two signed numbers (see Table 7.5 on page 268). Show that the following conditions are equivalent.

	Condition given in Table 7.5	Equivalent condition
jg	ZF = 0 and SF = OF	((SF xor OF) or ZF) = 0
jge	SF = OF	(SF xor OF) = 0
jl	SF $\neq$ OF	(SF xor OF) = 1
jle	ZF = 1 or SF $\neq$ OF	((SF xor OF) or ZF) = 1

8.10 Progamming Exercises

8–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`.

8–P2 Modify the octal-to-binary conversion program shown in Program 8.34 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.

8–P3 Write a program to multiply two signed 8-bit numbers using only shift instructions. Your program can read the two input numbers with `GetInt` and display the result by `PutInt`.

8–P4 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

8–P5 Modify the program for the last exercise to work with the long floating-point representation.

8–P6 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.

8–P7 A file processing interrupt (Interrupt 57H) returns the file time stamp (i.e., time last modified) in the following format:

bits	meaning
0 – 4	seconds (in increments of 2)
5 – 10	minutes
11 – 15	hours

Since there are only 5 bits allocated to represent seconds, this field, which can represent numbers in the range 0–31, should be interpreted as representing the number of 2 second increments. For example, 11:10:06 p.m. is represented as

$$\underbrace{\texttt{01011}}_{\textit{hours}}\underbrace{\texttt{001010}}_{\textit{minutes}}\underbrace{\texttt{00011}}_{\textit{seconds}}$$

Write a program to read the three components (hours, minutes, and seconds) from the user and convert the time to the format shown above. The format should be displayed in the binary form. Error checking is required for all three components. For example, the range of hours is 0 to 24. Error should be reported for values outside this range.

8–P8 Write a program that reads four hexadecimal digits from the user representing the time stamp discussed in the last exercise and outputs the corresponding time. For example, if the input is 5943, the output of the program should be:

```
The time stamp is: 11:10:06 p.m.
```

8–P9 Interrupt 57H also gives the date stamp of a file (indicating the date that the file was last modified). The format of the date stamp is:

bits	meaning
0 – 4	day
5 – 8	month
9 – 15	year (relative to 1980)

The year is relative to 1980. Thus, 1994 is represented as 14 in the year field. For example, October 19, 1994 is represented as

$$\underbrace{\texttt{0001110}}_{\textit{year}}\underbrace{\texttt{1010}}_{\textit{month}}\underbrace{\texttt{10011}}_{\textit{day}}$$

Write a program to read the three components (year, month, and day) from the user and convert the date to the format shown above. Month is given as a decimal number between 1 and 12. The format should be printed in the binary form. Error checking is required for all three components. For example, the year cannot be less than 1980. Also find the maximum year that can be represented and use it for error checking.

8–P10 Write a program that reads four hexadecimal digits from the user representing the date stamp discussed in the last exercise and outputs the corresponding date. For example, if the input is 1D53, the output of the program should be:

> The date stamp is: October 19, 1994

8–P11 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 of Pentium.

8–P12 Repeat the last exercise by using *only the shift and rotate* instructions of Pentium.

8–P13 Display the status of the flags register. In particular, display the status of the carry, parity, zero, and sign flags. See Chapter 2 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.

8–P14 Repeat the last exercise using `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) is the complement of `lahf` and stores the contents of the AH register in the lower-order byte of the flags register.

Part III

Advanced Topics

Chapter 9

String Processing

Objectives

- To discuss string representation schemes
- To describe string manipulation instructions of Pentium
- 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 9.1 discusses some of the representation schemes used to store strings. Pentium supports string processing by a special set of instructions. These instructions are described in Section 9.2. Several examples are presented in Section 9.3. The purpose of these examples is to illustrate the use of string instructions in developing procedures for string processing. Section 9.4 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 a more efficient code. The efficacy of the string instructions is demonstrated in Section 9.5. The chapter concludes with a summary.

S. P. Dandamudi, *Introduction to Assembly Language Programming*

9.1 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 such a representation, if a string has fewer characters, it is extended by padding, for example, 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 have 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 the majority of strings are shorter than the fixed length used.

The variable-length representation avoids these problems. In this scheme, a string can have as many characters as required (usually, within some system-imposed limit). Associated with each string, there is a string length attribute giving the number of characters in the string. The length attribute of each string is given in one of two ways:

1. Explicitly storing string length
2. Using a sentinel character

These two methods are discussed next.

9.1.1 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 will let the assembler do the job for us at assembly time.

9.1.2 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 string length explicitly. The assumption here is that the sentinel character is a special character that cannot appear within a string. DOS, for example, has a display string function (function 09H of int 21H) which expects a string terminated by $. In this case, we can display a string like

```
string1    DB    'This is OK$'
```

but string2 cannot be properly displayed:

```
string2    DB    'Price = $9.99$'
```

While this is peculiar to DOS, we normally use a special, nonprintable character that does not appear in a string. We have been using the ASCII NULL-character (00H) to terminate strings. Such NULL-terminated strings are called *ASCIIZ strings*.

In this representation, string1 and string2 can be represented without causing any problems as

```
string1    DB    'This is OK',0
string2    DB    'Price = $9.99',0
```

The C language, for example, uses this representation to store strings. Also, several DOS file-related functions expect file names stored in this representation. In the remainder of this chapter, we will use this representation for storing strings.

9.2 String Instructions

Pentium provides five main string processing instructions. These can be used to copy a string, to compare two strings, and so on. The five basic instructions are:

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

Specifying Operands

As indicated, each string instruction requires 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.

Variations

Each string instruction can operate on 8-, 16-, or 32-bit operands. As a 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 9.2.2).

9.2.1 Repetition Prefixes

String instructions can be repeated by using a repetition prefix. There are three prefixes falling into two categories: *unconditional* or *conditional* repetition. These are:

unconditional repeat	
rep	REPeat
conditional repeat	
repe/repz	REPeat while Equal REPeat while Zero
repne/repnz	REPeat while Not Equal REPeat while Not Zero

None of the flags are affected by these instructions.

rep

This is an unconditional repeat prefix and causes the instruction to repeat according to the value in the CX register. The semantics of `rep` are:

```
while (CX ≠ 0)
    execute the string instruction;
    CX := CX–1;
end while
```

The CX register is first checked and if it is not 0, only then is the string instruction executed. Thus, if CX is 0 to start with, the string instruction is not executed at all. This is in contrast to the `loop` instruction, which first decrements and then tests if CX is 0. Thus, with `loop`, CX = 0 results in a maximum number of iterations, and usually a `jcxz` check is needed.

repe/repz

This is one of the two conditional repeat prefixes. Its operation is similar to that of `rep` except that repetition is also conditional on the zero flag (ZF), as shown below:

```
while (CX ≠ 0)
    execute the string instruction;
    CX := CX–1;
    if (ZF = 0)
    then
        exit loop
    end if
end while
```

The maximum number of times the string instruction is executed is determined by the contents of CX, as in the `rep` prefix. But the actual number of times the instruction is repeated is determined by the status of ZF. Conditional repeat prefixes are useful with `cmps` and `scas` string instructions.

repne/repnz

This prefix is similar to the `repe/repz` prefix except that the condition tested for termination of repetition is ZF = 1.

```
while (CX ≠ 0)
    execute the string instruction;
    CX := CX–1;
    if (ZF = 1)
    then
        exit loop
    end if
end while
```

9.2.2 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 the string processing direction is forward or backward. 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

9.2.3 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:SI and the destination string location is indicated by ES:DI in memory. After copying, the SI and DI registers are updated according to the value of the direction flag and the operand size. Thus, before executing the `movs` instruction, all four registers should be set up appropriately. (This is necessary even if you use the first format.) Note that our focus is on 16-bit segments. For 32-bit segments, we have to use ESI and EDI registers.

```
movsb — move a byte string
    ES:DI := (DS:SI)   ; copy a byte
    if (DF = 0)        ; forward direction
    then
        SI := SI+1
        DI := DI+1
    else               ; backward direction
        SI := SI–1
        DI := DI–1
    end if
Flags affected: none
```

For word and doubleword opernads, 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
string2    DB    80 DUP (?)
.CODE
    .STARTUP
    mov    AX,DS            ; set up ES
    mov    ES,AX            ;  to the data segment
    mov    CX,strLen        ; strLen includes NULL
    mov    SI,OFFSET string1
    mov    DI,OFFSET string2
    cld                     ; forward direction
    rep    movsb
```

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:SI) 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:SI)      ; copy a byte
        if (DF = 0)        ; forward direction
        then
            SI := SI+1
        else                    ; backward direction
            SI := SI−1
        end if
    Flags affected: none
```

Use of the rep prefix does not make sense, as it will leave only the last value in AL, AX, or EAX. This instruction, along with the stos instruction, is often used when processing is required while copying a string. This point is elaborated after describing the stos instruction.

Store a String (stos)

This instruction performs the complementary operation. It copies the value in AL (for stosb), AX (for stosw), or EAX (for stosd) to the destination string (pointed to by ES:DI) in memory.

```
    stosb — store a byte string
        ES:DI := AL             ; copy a byte
        if (DF = 0)             ; forward direction
        then
            DI := DI+1
        else                    ; backward direction
            DI := DI−1
        end if
    Flags affected: none
```

We can use the rep prefix with the stos instruction if our intention is to initialize a block of memory with a specific character, word, or doubleword value. For example, the code

```
.DATA
array1    DW    100 DUP (?)
.CODE
    .STARTUP
    mov    AX,DS            ; set up ES
    mov    ES,AX            ;  to the data segment
    mov    CX,100
    mov    DI,OFFSET array1
    mov    AX,-1
    cld                     ; forward direction
    rep    stosw
```

initializes array1 with −1. Of course, we could have done the same with

```
array1    DW    100 DUP (-1)
```

at assembly time if we want to initialize only once.

In general, the rep prefix is not useful with lods and stos instructions. These two instructions are often used in a loop to do value conversions while copying data. For example, if string1 only contains letters and blanks,

```
    mov    CX,strLen
    mov    SI,OFFSET string1
    mov    DI,OFFSET string2
    cld                      ; forward direction
loop1:
    lodsb
    or    AL,20H
    stosb
    loop  loop1
done:
           .
           .
```

can convert it to a lowercase string. Note that blank characters are not affected because 20H represents blank in ASCII, and the

```
or    AL,20H
```

instruction does not have any effect on it. The advantage of lods and stos is that they automatically increment SI and DI registers.

9.2.4 String Compare Instruction

The cmps instruction can be used to compare two strings.

```
cmpsb — compare two byte strings
        Compare the two bytes at DS:SI and ES:DI and set flags
        if (DF = 0)          ; forward direction
        then
            SI := SI+1
            DI := DI+1
        else                 ; backward direction
            SI := SI−1
            DI := DI−1
        end if
Flags affected: As per cmp instruction
```

The cmps instruction compares the two bytes, words, or doublewords at DS:SI and ES:DI and sets the flags just like the cmp instruction. Like the cmp instruction, cmps performs

```
(DS:SI) – (ES:DI)
```

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 SI and DI 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 the 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    CX,strLen
    mov    SI,OFFSET string1
    mov    DI,OFFSET string2
    cld                   ; forward direction
    repe   cmpsb
```

leaves SI pointing to g in string1 and DI to f in string2. Therefore, adding

```
    dec    SI
    dec    DI
```

leaves SI and DI pointing to the 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 361).

repne/repnz 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    CX,strLen
    mov    SI,OFFSET string1 + strLen - 1
    mov    DI,OFFSET string2 + strLen - 1
    std                   ; backward direction
    repne  cmpsb
    inc    SI
    inc    DI
```

leaves SI and DI pointing to the first character that matches in the backward direction.

9.2.5 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:DI should point to the string to be searched.

```
scasb — scan a byte string
        Compare AL to the byte at ES:DI and set flags
        if (DF = 0)          ; forward direction
        then
            DI := DI+1
        else                 ; backward direction
            DI := DI−1
        end if
Flags affected: As per cmp instruction
```

Like with the cmps instruction, the repe/repz or the repne/repnz prefix can be used.

```
.DATA
string1    DB    'abcdefgh',0
strLen     EQU   $ - string1
.CODE
    .STARTUP
    mov    AX,DS          ; set up ES
    mov    ES,AX          ;  to the data segment
    mov    CX,strLen
    mov    DI,OFFSET string1
    mov    AL,'e'       ; character to be searched
    cld                 ; forward direction
    repne  scasb
    dec    DI
```

This program leaves DI pointing to e in string1. The following example can be used to skip initial blanks.

```
.DATA
string1    DB   '      abc',0
strLen     EQU  $ - string1
.CODE
    .STARTUP
    mov    AX,DS          ; set up ES
    mov    ES,AX          ;  to the data segment
    mov    CX,strLen
    mov    DI,OFFSET string1
    mov    AL,' '       ; character to be searched
    cld                 ; forward direction
    repe   scasb
    dec    DI
```

This program leaves DI pointing to the first nonblank character (a in the example) in string1.

9.3 Illustrative Examples

We now give some examples that illustrate the use of the string instructions discussed in this chapter. All these procedures are available in the string.asm file. These procedures receive the parameters via the stack. 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 SI or ES and DI using 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 should be a 16-bit general-purpose register, and source is a pointer to a 32-bit memory operand. The instructions perform the following actions:

```
lds
     register := (source)
           DS := (source + 2)

les
     register := (source)
           ES := (source + 2)
```

The 16-bit value at `source` in memory is copied to `register` and the next 16-bit value (i.e., at `source+2`) is copied to the DS or ES register. Both instructions affect none of the flags. By specifying SI as the register operand, `lds` can be conveniently used to set up a source string. Similarly, a destination string can be set up by specifying DI with `les`. For completeness, you should note that Pentium also supports `lfs`, `lgs`, and `lss` instructions 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
STRING1    EQU    DWORD PTR [BP+4]
STRING2    EQU    DWORD PTR [BP+8]
```

Example 9.1

Write a procedure `str_len` *to return the string length.*

String length is the number of characters in a string, excluding the NULL character. We 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 DI registers from the stack. STR_MAX, the maximum length of a string, is moved into CX, and the NULL character (i.e., 0) is moved into the AL register. The direction flag is cleared to initiate a forward search. The string length is obtained by taking the difference between the end of the string (pointed to by DI) and the start of the string available at [BP+4]. The AX register is used to return the string length value. This is similar to the C function `strlen`, which can be called as `strlen (string1)`.

```
;-----------------------------------------------------------
;String length procedure. Receives a string pointer
;(seg:offset) via the stack. If not a string, CF is set;
;otherwise, string length is returned in AX with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
str_len PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    ES
        les     DI,STRING1 ; copy string pointer to ES:DI
        mov     CX,STR_MAX ; needed to terminate loop if BX
                           ;  is not pointing to a string
        cld                ; forward search
        mov     AL,0       ; NULL character
        repne   scasb
        jcxz    sl_no_string  ; if CX = 0, not a string
        dec     DI         ; back up to point to NULL
        mov     AX,DI
        sub     AX,[BP+4]  ; string length in AX
        clc                ; no error
        jmp     SHORT sl_done
sl_no_string:
        stc                ; carry set => no string
sl_done:
        pop     ES
        pop     DI
        pop     CX
        pop     BP
        ret     4          ; clear stack and return
str_len ENDP
```

Example 9.2

Write a procedure `str_cpy` *to copy a string* `string2` *to string* `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 CX. 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 AX with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
str_cpy PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    SI
        push    DS
        push    ES
        ; find string length first
        lds     SI,STRING2  ; source string pointer
        push    DS
        push    SI
        call    str_len
        jc      sc_no_string

        mov     CX,AX       ; source string length in CX
        inc     CX          ; add 1 to include NULL
        les     DI,STRING1 ; dest. string pointer
        cld                 ; forward copy
        rep     movsb
        mov     AX,[BP+4]   ; 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     SI
        pop     DI
        pop     CX
        pop     BP
        ret     8           ; clear stack and return
str_cpy ENDP
```

Example 9.3

Write a procedure `str_cat` *to concatenate a string* `string2` *to another string* `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 DI 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 AX with CF = 0.
;Preserves all registers.
;-----------------------------------------------------------
str_cat PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    SI
        push    DS
        push    ES
        ; find string length first
        les     DI,STRING1 ; dest. string pointer
        mov     CX,STR_MAX ; max string length
        cld                ; forward search
        mov     AL,0       ; NULL character
        repne   scasb
        jcxz    st_no_string
        dec     DI         ; back up to point to NULL
        lds     SI,STRING2 ; source string pointer
        push    DS
        push    SI
        call    str_len
        jc      st_no_string

        mov     CX,AX        ; source string length in CX
        inc     CX           ; add 1 to include NULL
        cld                  ; forward copy
        rep     movsb
        mov     AX,[BP+4]    ; 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     SI
        pop     DI
        pop     CX
        pop     BP
        ret     8            ; clear stack and return
str_cat ENDP
```

Example 9.4

Write a procedure `str_cmp` *to compare two strings* `string1` *and* `string2`.

This function uses the `cmpsb` instruction to compare two strings. It returns in AX 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 AX register. The corresponding function in C is `strcmp`, which can be invoked by `strcmp(stingl,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 AX with CF = 0 as shown below:
;    AX = negative value  if string1 < string2
;    AX = zero            if string1 = string2
;    AX = positive value  if string1 > string2
;Preserves all registers.
;-----------------------------------------------------------
str_cmp PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    SI
        push    DS
        push    ES
        ; find string length first
        les     DI,STRING2  ; string2 pointer
        push    ES
        push    DI
        call    str_len
        jc      sm_no_string

        mov     CX,AX       ; string1 length in CX
        inc     CX          ; add 1 to include NULL
        lds     SI,STRING1 ; string1 pointer
        cld                 ; forward comparison
        repe    cmpsb
        je      same
        ja      above
below:
        mov     AX,-1       ; AX = -1 => string1 < string2
        clc
        jmp     SHORT sm_done
same:
        xor     AX,AX       ; AX = 0 => string match
        clc
        jmp     SHORT sm_done
above:
        mov     AX,1        ; AX = 1 => string1 > string2
        clc
        jmp     SHORT sm_done
sm_no_string:
        stc                 ; carry set => no string
sm_done:
        pop     ES
        pop     DS
        pop     SI
        pop     DI
        pop     CX
        pop     BP
        ret     8           ; clear and return
str_cmp ENDP
```

Example 9.5

Write a procedure str_chr *to locate a character* chr *in a string* 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 AX. Note that chr is passed as a 16-bit value, even though only the lower half of the word 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 AX (if the
;search is successful; otherwise AX = NULL) with CF = 0.
```

```
;Preserves all registers.
;-----------------------------------------------------------
str_chr PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    ES
        ; find string length first
        les     DI,STRING1  ; source string pointer
        push    ES
        push    DI
        call    str_len
        jc      sh_no_string

        mov     CX,AX       ; source string length in CX
        inc     CX
        mov     AX,[BP+8]   ; read char. into AL
        cld                 ; forward search
        repne   scasb
        dec     DI          ; back up to match char.
        xor     AX,AX       ; assume no char match (AX=NULL)
        jcxz    sh_skip
        mov     AX,DI       ; 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     DI
        pop     CX
        pop     BP
        ret     6           ; clear stack and return
str_chr ENDP
```

Example 9.6

Write a procedure str_cnv *to convert a string* string2 *to another string* 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 CX, 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
```

```
;-------------------------------------------------------------
;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 AX with CF = 0.
;Preserves all registers.
;-------------------------------------------------------------
str_cnv PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    SI
        push    DS
        push    ES
        ; find string length first
        lds     SI,STRING2  ; source string pointer
        push    DS
        push    SI
        call    str_len
        jc      sn_no_string

        mov     CX,AX      ; source string length in CX
        inc     CX         ; add 1 to include NULL
        les     DI,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     AX,[BP+4]   ; 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     SI
        pop     DI
        pop     CX
        pop     BP
        ret     8           ; clear stack and return
str_cnv ENDP
```

Example 9.7

Write a procedure str_mov *to move a string* string1 *left or right by* num *number of positions.*

The objective of this example is to show how a particular direction of copying a string is important. This procedure receives a pointer to a string string1 and an integer num indicating the number of positions the string value is to be moved *within* the string. A positive num value is treated as a move to the right and a negative value as a move to the left. A 0 value has no effect. Note that the pointer received by this function need not be pointing to the beginning of string1. It is important to make sure that there is enough room in the original string in the intended direction of the move.

```
;-----------------------------------------------------------
;String move procedure. Receives a signed integer
;and a string pointer (seg:offset) via the stack.
;The integer indicates the number of positions to move
;the string:
;       -ve number => left move
;       +ve number => right move
;If string1 is not a string, CF is set;
;otherwise, string is moved left or right and returns
;a pointer to the modified string in AX with CF = 0.
;Preserves all registers.
;-------------------------------------------------------------
str_mov PROC
        push    BP
        mov     BP,SP
        push    CX
        push    DI
        push    SI
        push    DS
        push    ES
        ; find string length first
        lds     SI,STRING1  ; string pointer
        push    DS
        push    SI
        call    str_len
        jnc     sv_skip1
        jmp     sv_no_string
sv_skip1:
        mov     CX,AX       ; string length in CX
        inc     CX          ; add 1 to include NULL
        les     DI,STRING1
        mov     AX,[BP+8]   ; copy # of positions to move
        cmp     AX,0        ; -ve number => left move
        jl      move_left   ; +ve number => right move
        je      finish      ; zero => no move
move_right:
        ; prepare SI and DI for backward copy
        add     SI,CX       ; SI points to the
        dec     SI          ;  NULL character
        mov     DI,SI       ; DI = SI + # of positions to move
        add     DI,AX
        std                 ; backward copy
        rep     movsb
        ; now erase the remainder of the old string
        ;  by writing blanks
        mov     CX,[BP+8]   ; # of positions moved
        ; DI points to the first char of left-over string
        mov     AL,' '      ; blank char to fill
        ; direction flag is set previously
        rep     stosb
        jmp     SHORT finish
move_left:
        add     DI,AX
        cld                 ; forward copy
        rep     movsb
finish:
        mov     AX,[BP+8]   ; add # of positions to move
        add     AX,[BP+4]   ; to string pointer (ret value)
        clc                 ; no error
        jmp     SHORT sv_done
sv_no_string:
        stc                 ; carry set => no string
sv_done:
        pop     ES
        pop     DS
        pop     SI
        pop     DI
        pop     CX
        pop     BP
        ret     6           ; clear stack and return
str_mov ENDP
```

To move left, we let SI point to the same character of string1 as the pointer received by the procedure. We let DI := SI + num. Since num is negative for left move, DI points to where the character pointed by SI should move. A simple forward copy according to the string length (plus one) will move the string value. The extraneous characters left of the original string value will not cause any problems, as a NULL terminates the moved value, as shown below:

string1 before str_mov
⊔⊔⊔⊔abcd0
string1 after str_mov with the string pointing to a and num = −2
⊔⊔abcd0d0

where ⊔ indicates a blank.

To move right, we let SI point to the NULL character of string1 and DI to its right by num positions. A straightforward copy in the backward direction will move the string to its destination position. However, this leaves remnants of the old values on the left, as shown in the following example:

string1 before str_mov
⊔⊔abcd0⊔⊔
string1 after str_mov with the string pointing to a and num = 2
⊔⊔ababcd0

To eliminate this problem, str_mov erases the contents of the remaining characters of the original value by filling them with blanks. In this example, the first ab characters will be filled with blanks.

9.4 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 9.35. 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 16-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.

With indirect near procedure calls, this offset is given with one level of indirection. That is, the call instruction itself will contain either a memory word address (through a label), or a 16-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    BX
```

if BX contains the offset of the target procedure. As a part of executing this call instruction, the contents of the BX register are used to load IP to transfer control to the target procedure. Similarly, we can use

```
call    target_proc_ptr
```

if the word in memory at target_proc_ptr contains the offset of the target procedure. The jmp is another instruction that can be used for indirect jumps in exactly the same way as the indirect call to provide a one-way transfer of control.

Back to the Example

To facilitate calling the appropriate procedure, we maintain a procedure pointer table proc_ptr_table. The user query response is used as an index into this table to get the target procedure offset. The BX register is used as the index into this table. The instruction

```
call    proc_ptr_table[BX]
```

causes the indirect procedure call. The rest of the program is straightforward and is not discussed any further.

Program 9.35 String test program `str_test.asm`

```
                     .
                     .
.DATA
proc_ptr_table  DW  str_len_fun,str_cpy_fun,str_cat_fun
                DW  str_cmp_fun,str_chr_fun,str_cnv_fun
                DW  str_mov_fun
MAX_FUNCTIONS   EQU ($ - proc_ptr_table)/2

choice_prompt   DB  'You can test several functions.',CR,LF
                DB  '    To test       enter',CR,LF
                DB  'String length       1',CR,LF
                DB  'String copy         2',CR,LF
                DB  'String concatenate  3',CR,LF
                DB  'String compare      4',CR,LF
                DB  'Locate character    5',CR,LF
                DB  'Convert string      6',CR,LF
                DB  'Move string         7',CR,LF
                DB  'Invalid response terminates program.',CR,LF
                DB  'Please enter your choice: ',0

invalid_choice  DB  'Invalid chioce - program terminates.',0

string1         DB  STR_MAX DUP (?)
string2         DB  STR_MAX DUP (?)
                     . . .

.CODE
                     . . .

main    PROC
        .STARTUP
        mov     AX,DS
        mov     ES,AX
```

```
query_choice:
        xor     BX,BX
        PutStr  choice_prompt    ; display menu
        GetCh   BL               ; read response
        nwln
        sub     BL,'1'
        cmp     BL,0
        jb      invalid_response
        cmp     BL,MAX_FUNCTIONS
        jb      response_ok
invalid_response:
        PutStr  invalid_choice
        jmp     SHORT done
response_ok:
        shl     BL,1                ; multiply BL by 2
        call    proc_ptr_table[BX]  ; indirect call
        jmp     query_choice
done:
        .EXIT
main    ENDP
                    . . .

        END     main
```

9.5 Performance: Advantage of String Instructions

A question that naturally arises is: How beneficial are these string instructions? We will answer this question by looking at the cmpsb instruction in performing string comparisons.

There are two chief advantages that we get from 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.

For example, movsb can copy a byte from one memory location to another and increment both index registers. Such memory-to-memory transfer of a byte is not possible with the mov instruction, which requires an intermediate register to achieve the same, as indicated below:

```
mov     AL,[SI]
```

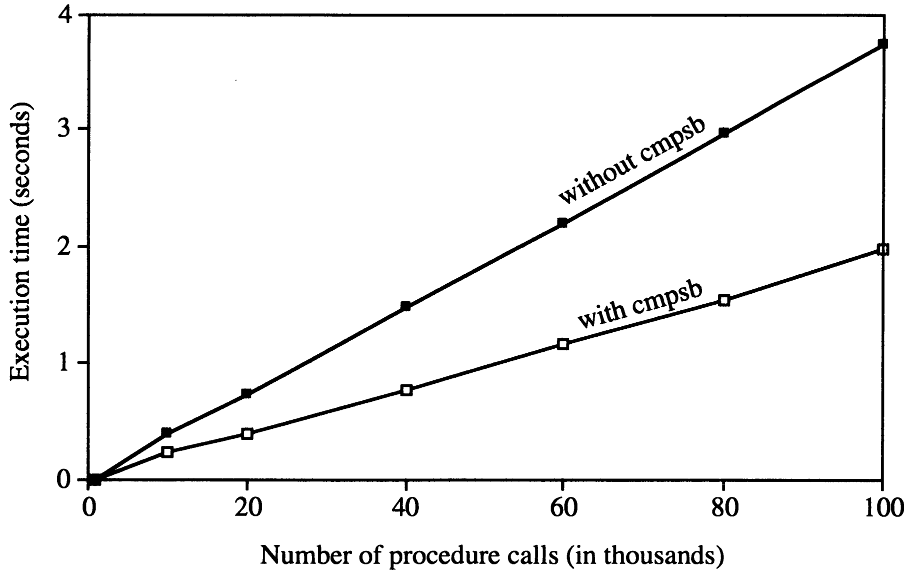


Figure 9.1 Performance impact of the cmpsb instruction in comparing two matching strings of length 100.

```
mov    ES:[DI],AL
inc    SI
inc    DI
```

We will compare two versions of the str_cmp procedure: one using cmpsb and the other without this instruction. Figure 9.1 shows the performance of these two versions. 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 have compared two matching strings of length 100. The version that uses the string instruction appears to perform almost twice as fast as the other version. The performance difference increases with increasing string length, as shown in Figure 9.2. In this figure, each procedure is called 60,000 times on two matching strings of varying length (as given on the x-axis).

Is the C Routine Worse?

Since this is a short, well-defined procedure, we do not expect the C routine to perform any worse than our best assembly language procedure. Indeed, tests on the Turbo C strcmp routine show that it has the same execution time as the

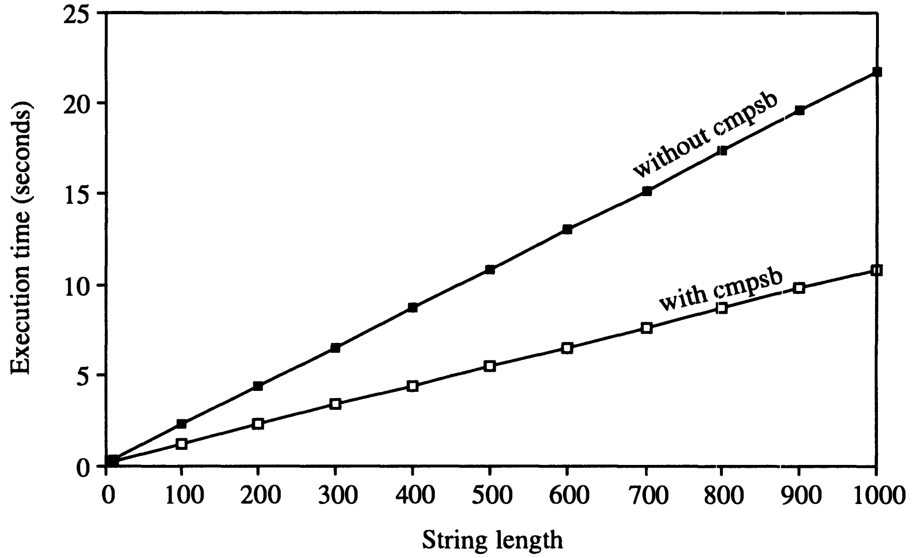


Figure 9.2 Performance impact of the `cmpsb` instruction in comparing two matching strings 60,000 times.

`cmpsb` version of the assembly language procedure. Thus, it is not beneficial to develop such routines; rather, the existing library routines should be used. These routines can be called from assembly language programs. Chapter 13 discusses how high-level languages and assembly languages can be interfaced.

9.6 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 a string. High-level programming languages like C use NULL-terminated storage representation for strings. We have also used the same representation to store strings.

Pentium provides 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:SI and/or ES:DI for

16-bit segments. For 32-bit segments, ESI and EDI registers are used instead of SI and DI 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 string processing procedures. Using string instructions can substantially reduce the executing time (up to about 50 percent) of string operations.

9.7 Exercises

9–1 What are the advantages and disadvantages of the fixed-length string representation?

9–2 What are the advantages and disadvantages of the variable-length string representation?

9–3 Discuss the pros and cons of storing the string length explicitly versus using a sentinel character for storing variable-length strings.

9–4 What is an ASCIIZ string?

9–5 We can write procedures to perform string operations without using the string instructions. What is the advantage of using the string instructions? Explain why?

9–6 Why doesn't it make sense to use the `rep` prefix with the `lods` instruction?

9–7 Explain why it does not make sense to use conditional repeat prefixes with `lods`, `stos`, or `movs` string instructions.

9–8 Both `loop` and repeat prefixes use the CX register to indicate the repetition count. Yet there is one significant difference between them in how they use the CX register value. What is this difference?

9–9 Identify a situation in which the direction of string processing is important.

9–10 Identify a situation in which a particular direction of string processing is mandatory.

9–11 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.

9–12 Compare the space and time requirements of lds and the code you have written in the last exercise. To do this exercise, you need to refer to the Pentium data book.

9–13 What is the difference between the direct procedure call and the indirect procedure call?

9–14 Explain how you can use the indirect procedure call to pass a procedure to be executed as a parameter.

9–15 Figure 9.1 shows that the cmpsb version performs better. Explain intuitively why this is so.

9–16 Suppose a given application can be written using either

```
rep     movsb
```

or

```
rep     movsw
```

Which one is more efficient and why?

9–17 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.

9.8 Progamming Exercises

9–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 but not the actual strings. It should copy at most the first num characters of string2 to string1.

9–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 a 0 value like the str_cmp procedure does.

9–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 AX 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.

9–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.

9–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.

9–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 AX the starting position of the first match. Matching should be case sensitive. A negative value is returned in AX if no match is found. For example, if

> `string` = Good things come in small packages.

and

> `substring` = in

the procedure should return 8 in AX indicating a match of `in` in `things`.

9–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.

9–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.

9–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.

9–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`

9–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 10

Macros and Conditional Assembly

Objectives

- To discuss macro definition and expansion
- To explain how blocks of statements can be repeated
- To describe conditional assembly directives
- To explore the performance tradeoffs associated with macros and procedures

We have seen that procedures are extremely useful to implement modular program design techniques. This chapter discusses another mechanism to modularize program code. Program modules can also be implemented by using macros. Simply put, a macro is a sophisticated text substitution mechanism. Although not used as frequently as procedures, macros are useful in certain situations. Macros are discussed in detail in Sections 10.1–10.6.

A related topic—how .LST file contents are controlled—is discussed in Section 10.7. Repetition directives, which are discussed in Section 10.8, can be used to repeat a block of statements. These directives can be used both inside and outside of a macro definition.

Section 10.9 discusses the conditional assembly directives, which allow conditional assembly of a block of statements. These directives are useful in generating customized code. Section 10.10 discusses nested macros.

Finally, Section 10.11 deals with the performance tradeoffs associated with macros and procedures. The chapter concludes with a summary.

S. P. Dandamudi, *Introduction to Assembly Language Programming*

10.1 What Are Macros?

Macros provide a means by which a block of text (code, data etc.) can be represented by a name (called a *macro name*). When the assembler encounters that name later in your program, the block of text associated with the macro name is substituted. The process is referred to as *macro expansion*. In simple terms, macros provide a sophisticated text substitution mechanism.

Macros are not unique to assembly language. High-level languages also support such mechanisms. For example, in C language, macros can be defined with the #define preprocessor directive. The statement

```
#define    CLASS_SIZE    90
```

causes 90 to be textually substituted for CLASS_SIZE. Similarly, if we define

```
#define    begin    {
#define    end      }
```

we can write C code using begin and end to group statements like in Pascal, as shown below:

```
if (value > 100)
begin
    value = 100;
    count++;
end;
```

This is a valid C code because the preprocessor changes the source code to

```
if (value > 100)
{
    value = 100;
    count++;
};
```

before passing it on to the C compiler.

Both MASM and TASM assemblers support macros. In fact, these assemblers provide three directives to support macro substitutions: =, EQU, and MACRO. We have already discussed how = and EQU directives can be used for constants whose values are available at assembly time (see Chapter 3). For example, we can write

```
CLASS_SIZE = 90
```

or

```
CLASS_SIZE EQU 90
```

The difference between = and EQU directives is that the = directive allows redefinition later on, while the constants defined by EQU cannot be redefined. However, both these directives are not useful for general text substitution (like we did with #define for begin and end). Macros provide such a flexibility. In fact, they have many advanced features to allow us to write powerful macros. We will take a look at some of these features in this chapter.

In assembly language, macros can be defined with MACRO and ENDM directives. The macro text begins with the MACRO directive and ends with the ENDM directive. The macro definition syntax is

```
macro_name    MACRO [parameter1, parameter2,...]
              macro body
              ENDM
```

In the MACRO directive, the parameters are optional (as indicated by the square brackets [ ]). macro_name is the name of the macro that, when used later in the program, causes a *macro expansion.* To invoke or call a macro, use the macro_name and supply the necessary parameter values. The format is

```
macro_name [argument1, argument2, ...]
```

Example 10.1

Here is our first macro example that does not require any parameters. We have seen in Chapter 8 that using shift left to multiply by a power of 2 is more efficient than using the imul instruction. So, let us write a macro to do this.

```
multAX_by_16  MACRO
              sal    AX,4
              ENDM
```

The macro code consists of a single sal statement, which will be substituted whenever the macro is called. Now we can invoke this macro by using the macro name multAX_by_16, as in the following example:

```
      .
      .
mov   AX,27
multAX_by_16
      .
      .
```

When the assembler encounters the macro name multAX_by_16, it is replaced (i.e., text substituted) by the macro body. Thus, after the macro expansion, the assembler finds the code

```
              .
              .
mov     AX,27
sal     AX,4
              .
              .
```

10.2 Macros with Parameters

Just like procedures, using parameters with macros aids in writing more flexible and useful macros. The number of parameters is limited by how many can fit in a single line. The previous macro always multiplies AX by 16. By using parameters, we can generalize this macro to operate on a byte, word, or doubleword located either in a general-purpose register or memory. The modified macro is:

```
mult_by_16      MACRO  operand
                sal    operand,4
                ENDM
```

The parameter operand can be any operand that is valid in the sal instruction. This macro can be invoked to multiply a byte, word, or doubleword located in a register or memory. 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 by

```
mult_by_16     count
```

Such a macro call will be expanded as

```
sal     count,4
```

Now, at least superficially, mult_by_16 looks like any other assembly language instruction, except that we have defined it. These are referred to as *macro-instructions*, which as we have seen, generate one or more assembly language instructions.

The 8086 processor does not allow specification of shift count greater than 1 as an immediate value. For this processor, we have to redefine the macro as

```
mult_by_16_8086 MACRO  operand
                sal    operand,1
                sal    operand,1
                sal    operand,1
                sal    operand,1
                ENDM
```

TASM, however, allows you to write immediate shift count values greater than 1 and replaces them by an equivalent set of shift instructions.

Example 10.2

You may have noticed that the Pentium instruction set does not allow memory-to-memory data transfer. We have to use an intermediate register to facilitate such a data transfer. We can write a macro to perform memory-to-memory data transfers using the basic instructions of the processor. Let us call this macro, which exchanges the values of two memory variables, Wmxchg to exchange words of data in memory.

```
Wmxchg    MACRO   operand1, operand2
          xchg    AX,operand1
          xchg    AX,operand2
          xchg    AX,operand1
          ENDM
```

You can easily verify that this sequence exchanges the memory words operand1 and operand2 while leaving AX as it is.

We can define similar macros for byte and doubleword operands. Here is an example for the byte operands.

```
Bmxchg    MACRO   operand1, operand2
          xchg    AL,operand1
          xchg    AL,operand2
          xchg    AL,operand1
          ENDM
```

Later in this chapter (see page 403), we will show that, using conditional assembly directives, these two macros can be combined into one.

10.3 Macros Versus Procedures

Macros are similar to procedures in some respects. Both improve programmer productivity by aiding in the development of modular source code. Both can be used when a block of code is repeated in the source code. There are, however, some significant differences between them.

1. **Parameter passing**
Parameter passing in a macro invocation is similar to that in a procedure call of a high-level language. The arguments are listed as part of a macro call. For example, to call the mult_by_16 macro to multiply the contents of AX by 16, we can write

```
mult_by_16    AX
```

Parameter passing in a procedure call often involves the stack. Assuming that the stack is used to pass the parameters, to call a procedure times16 to do the same job

```
push    AX
call    times16
```

The number of stack operations in preparation for a procedure call grows in direct proportion to the number of parameters passed. This, in addition to the call/ret overhead, is the additional overhead associated with a procedure call. Macros avoid this overhead by text substitution but increase the space requirement. The performance tradeoffs are further elaborated in Section 10.11.

2. **Types of parameters**
Since a macro is a text substitution mechanism, a variety of parameter types can be passed. For example, we can write a macro

```
shift    MACRO    op_code,operand,count
         op_code  operand,count
         ENDM
```

and invoke it as

```
shift    sal,AX,3
```

which results in the following expansion

```
sal    AX,3
```

Here op_code is the instruction mnemonic. Any mnemonic in the shift and rotate family of instructions can be given. Thus, the same macro can be used with all of the shift and rotate family of instructions on bytes, words, and doublewords that are either located in a register or memory. Clearly, such parameter types cannot be passed to a procedure.

3. **Invocation mechanism**
The main difference between a macro invocation and a procedure invocation is the following: macro invocation is done *at assembly time* by text

substitution, while procedure invocation is done *at run time* by transferring control to the procedure. This leads to the following tradeoff: Macros tend to increase the length of the executable code due to macro expansions. This leads to increased assembly time. Macro expansion also creates a nuisance at debugging time—repeatedly looking at a block of code (macro expansions) that you know works correctly. Of course, there are ways to suppress macro expansions (see the `.SALL` directive described in Section 10.7).

Procedures avoid these problems by transferring control to the only copy of the procedure code. In debugging, the procedure call, which appears as a single call instruction, can be skipped.

In summary, the tradeoffs are that using macros results in faster execution of the code for reasons discussed before (further elaborated in Section 10.11). But macros result in increased memory space due to macro expansions. Procedures save space, as only one copy of the procedure is kept. However, procedure invocation overhead (to pass parameters via the stack and for `call/ret`) increases the execution time. Note that macro invocation causes assembly-time overhead but not run-time overhead. The advantages and disadvantages associated with macros and procedures can be summarized as:

Type of overhead	Procedure	Macro
Memory space	lower	higher
Execution time	higher	lower
Assembly time	lower	higher

Given the state of modern technology, this time versus space tradeoff is not a major factor in preferring one over the other. The choice between macro and procedure depends on the application requirements. Our recommendation, for typical applications, is to use procedures except in some special situations as identified here.

When Are Macros Better?

1. Macros are useful in defining macro-instructions that extend the instruction set of a processor.

Example 10.3

The macro `mult_by_16` is an example for which it is impractical to write a procedure to do the same task. An equivalent procedure `times16` may look like this:

```
times16     PROC
            push    BP
            mov     BP,SP
            push    AX
            mov     AX,[BP+4]
            sal     AX,4
            mov     [BP+4],AX
            pop     AX
            pop     BP
            ret     2
times16     ENDP
```

This procedure can be invoked to multiply a word variable count by 16 as

```
push    count
call    times16
pop     count
```

The overhead involved is substantial. Clearly, this is an impractical proposition.

2. Macros are useful when text substitution is the only route available. Look at the following example.

Example 10.4

Suppose we want to preserve BX, CX, DX, SI, DI, and BP registers across procedure calls. We could use pusha and popa, but these instructions save and restore the AX register as well. But we want to return a result in AX. We can conveniently do this by the following two macros:

```
save_regs  MACRO              restore_regs  MACRO
           push   BP                        pop    BX
           push   DI                        pop    CX
           push   SI                        pop    DX
           push   DX                        pop    SI
           push   CX                        pop    DI
           push   BX                        pop    BP
           ENDM                             ENDM
```

It is not possible to write a procedure to do the same. Thus, even though we can rely on procedures to produce modular code most of the time, there are instances where procedures are inadequate. A proper mix of procedures and macros make programs more modular and readable, and therefore, aid in maintaining them.

10.4 Labels in Macros

In the macros we have seen so far, there were no jump instructions in the macro body. We will now look at a macro with flow control statements. Consider the following macro that converts a lowercase alpha character to the corresponding uppercase letter.

```
; There is a problem with this macro definition
to_upper0     MACRO   ch
              cmp     ch,'a'
              jb      done
              cmp     ch,'z'
              ja      done
              sub     ch,32
        done:
              ENDM
```

The macro `to_upper0` performs the following:

```
if ((ch ≥ 'a') AND (ch ≤ 'z'))
then ch := ch − 32
end if
```

If we invoke this macro more than once in a program, the label done appears more than once when the macro calls are expanded. This causes the assembler to complain about the duplicate label done. In addition, the program that is invoking this macro should not use the label done even if the macro is invoked only once in the program. This is clearly undesirable—particularly if we want to use macros from a library.

Such label problems are avoided in procedures, as the scope of any label declared within a procedure body is limited to that procedure body itself. We need a similar mechanism to limit the scope of labels declared in a macro.

The LOCAL directive is provided by the assembler for this purpose. It can be used to declare labels in a macro local to that macro. The syntax is

```
LOCAL local_label1 [, local_label2, ...]
```

Using the LOCAL directive, the `to_upper0` macro can be written as

```
to_upper    MACRO  ch
            LOCAL  done
            cmp    ch,'a'
            jb     done
            cmp    ch,'z'
            ja     done
            sub    ch,32
      done:
            ENDM
```

If the LOCAL directive is used in a macro, it must immediately follow the MACRO directive.

Typically, the labels that the assembler generates to replace the local labels are of the form

```
??XXXX
```

where XXXX is a hexadecimal number between 0 and FFFFH. The assembler maintains an internal counter to generate the number portion of the label. See the example .LST file shown in Figure 10.1. To avoid conflicts, you should not use labels in your program that begin with ??. A program can have up to $2^{16} = 65,536$ local labels, a large enough number for most programs.

10.5 Comments in Macros

Comments in macros deserve special attention. We would not like all the comments in a macro definition to appear every time the macro is expanded. On the other hand, we need to add comments to the macro body to explain the logic of the code—especially for complicated macros.

Note that the assembler does nothing to the code in a macro definition. The assembler generates actual code only when a macro is invoked. See the .LST output that follows this discussion (see Figure 10.1).

To provide the programmer flexibility regarding the comments in macros, assembler provides the ;; operator to suppress comments from appearing in macro expansions. Any comment that starts with ;; will appear only in the macro definition but not in macro expansions. The comments that start with ; (i.e., the standard comments) appear in macro expansions as well.

```
;;Converts a lowercase letter to uppercase.
to_upper  MACRO   ch
        LOCAL   done
        ; case conversion macro
        cmp     ch,'a'  ;; check if ch >= 'a'
```

```
      17 0000  B0 62                        mov     AL,'b'
      18                                    to_upper   AL
1     19                                    ; case conversion macro
1     20 0002  3C 61                        cmp     AL,'a'
1     21 0004  72 06                        jb      ??0000
1     22 0006  3C 7A                        cmp     AL,'z'
1     23 0008  77 02                        ja      ??0000
1     24 000A  2C 20                        sub     AL,32
1     25 000C                       ??0000:
      26 000C  8A D8                        mov     BL,AL
      27 000E  B4 31                        mov     AH,'1'
      28                                    to_upper   AH
1     29                                    ; case conversion macro
1     30 0010  80 FC 61                     cmp     AH,'a'
1     31 0013  72 08                        jb      ??0001
1     32 0015  80 FC 7A                     cmp     AH,'z'
1     33 0018  77 03                        ja      ??0001
1     34 001A  80 EC 20                     sub     AH,32
1     35 001D                       ??0001:
      36 001D  8A FC                        mov     BH,AH
```

Figure 10.1 An example listing file showing macro expansions.

```
        jb      done
        cmp     ch,'z'  ;;   and if ch >= 'z'
        ja      done
        sub     ch,32   ;; then ch := ch - 32
done:
        ENDM
```

Calling this macro by

```
mov     AL,'b'
to_upper   AL
mov     BL,AL
mov     AH,'1'
to_upper   AH
mov     BH,AH
```

generates the .LST file shown in Figure 10.1.

10.6 Macro Operators

There are five special operators to manipulate macros.

Operator	Meaning
;;	Suppress comment operator
&	Substitute operator
< >	Literal-text string operator
!	Literal-character operator
%	Expression evaluate operator

We have already seen how the ;; operator works. The rest of the section briefly discusses the remaining four operators.

Substitute Operator (&)

The substitute operator forces the assembler to substitute a parameter with the actual argument. The syntax is

```
&name
```

where name is the value of the argument in the macro call. The & operator is typically used to concatenate one or more parameters with other text.

Example 10.5

We can use the following macro to sort two numbers num1 and num2.

```
sort2    MACRO  cond, num1, num2
         LOCAL   done
         push    AX
         mov     AX,num1
         cmp     AX,num2
         j&cond  done
         xchg    AX,num2
         mov     num1,AX
    done:
         pop     AX
         ENDM
```

This macro works on 16-bit signed or unsigned numbers. The cond parameter specifies the relationship of num2 relative to num1. For example, to sort two unsigned numbers value1 and value2 such that value1 ≥ value2, we can invoke the macro as

```
sort2    ae,value1,value2
```

which causes the following macro expansion:

```
              push    AX
              mov     AX,value1
              cmp     AX,value2
              jae     ??0000
              xchg    AX,value2
              mov     value1,AX
       ??0000:
              pop     AX
```

If value1 and value2 are signed numbers, this macro should be invoked as

```
sort2    ge,value1,value2
```

which generates the jge conditional jump instruction in the macro expansion.

The substitute operator is also useful to force the assembler to substitute a parameter inside a quoted string. The following example illustrates this point.

Example 10.6

Suppose we want to generate a set of messages to inform the user when he/she inputs a number that is out of range. The following macro definition

```
;; incorrect macro definition
range_error     MACRO  number,variable
err_msg&number    DB   'variable: out of range',0
                ENDM
```

does not work because parameter substitution is not done inside a quoted string. For example, if we invoke the above macro as

```
range_error     1,Assignment_mark
```

it is expanded as

```
err_msg1    DB    'variable: out of range',0
```

which is not what we want. The correct macro definition is

```
;; correct macro definition
range_error     MACRO  number,variable
err_msg&number    DB   '&variable: out of range',0
                ENDM
```

When invoked as

```
     range_error    1,Assignment_mark
```

the macro will be expanded as

```
     err_msg1   DB   'Assignment_mark: out of range',0
```

Literal-text string operator (< >)

The literal-text string operator < > informs the assembler that the enclosed text should be treated as a single string rather than separate arguments. The syntax is

```
     <text>
```

The `text` is treated as a single argument even if it contains parameter separators. Some typical parameter separators are commas, spaces, tabs, etc. The assembler removes the angle brackets and uses text as the argument.

Example 10.7

Consider the `range_error` macro defined before. Suppose that we also want to inform the user of the correct range along with the error message. The modified macro is shown below.

```
range_error1     MACRO    number,variable,range
err_msg&number     DB   '&variable: out of range',0
range_msg&number  DB   'Correct range is &range',0
                 ENDM
```

When we invoke this macro as

```
range_error1     1,<Assignment mark>,<0 to 25>
```

the macro expansion will look like

```
err_msg1    DB   'Assignment mark: out of range',0
range_msg1  DB   'Correct range is 0 to 25',0
```

This operator can also be used to force the assembler to treat a character literally by removing its default special meaning. For example, <;> passes ; as an argument without treating it as the comment operator. This can also be done by using the literal-character operator, as we shall see next.

Literal-Character Operator (!)

The literal-character operator ! preceding a character forces the assembler to treat the character literally without its default special meaning. The syntax is

```
    !character
```

Thus, !; is equivalent to <;>. The following example shows an instance where this operator is useful.

Example 10.8

If we define a macro as

```
range_error2    MACRO     number,variable,range
err_msg&number  DB  '&variable: out of range - &range',0
                ENDM
```

we can invoke this in .DATA part as

```
range_error2    3,mark,<can!'!'t be !> 100>
```

to create the error message

```
err_msg3   DB   'mark: out of range - can''t be > 100',0
```

If we didn't use the ! operator in the third argument, the assembler would have interpreted the third argument as <can''t be>. Note that two successive single quotes will produce a single quote in the output.

Expression Evaluate Operator (%)

The syntax of the expression evaluate operator % is

```
    %expression
```

The expression is evaluated and its value is used to replace the expression itself. Typically, this operator is used to provide an argument in a macro call, as shown in the following example.

Example 10.9

Let us consider a macro to allocate and initialize an array of given size. The macro receives the name and size of the array, element size (B for byte, W for word, . . .), and the initial value.

```
init_arry  MACRO  element_size,name,size,init_value
name   D&element_size   size DUP (init_value)
           ENDM
```

We can call this macro to reserve space for an integer array of 47×7 (marks) and initialize it to −1.

```
init_array   W,marks,%NUM_STUDENTS*NUM_TESTS,-1
```

Assuming that

```
NUM_STUDENTS    EQU   47
NUM_TESTS       EQU    7
```

have been defined, the macro call will be expanded as

```
marks    DW    329 DUP (-1)
```

10.7 List Control Directives

There are several list control directives available to specify the contents of the output listing (.LST) file. These directives can appear anywhere in the assembly language source code. Each directive will be in effect until replaced by another directive. The two directives .LIST and .XLIST control the source lines in the .LST file.

Directive	Meaning
.LIST	Allows listing of subsequent source lines (This is the default mode.)
.XLIST	Suppresses listing of subsequent source lines

For example, if you want to suppress the contents of an include file, you can do so by

```
                .
                .
           .XLIST       ; suppress listing
INCLUDE    part0.inc
           .LIST        ; restore listing
                .
                .
```

By default, all source lines are placed in the .LST file.

The following three list control directives—.LALL, .SALL, and .XALL—affect only macro invocation calls. The assembler always lists the macro definitions in the .LST file.

Directive	Meaning
.LALL	Enables listing of macro expansions
.SALL	Suppresses listing of all statements in macro expansions
.XALL	Lists only the source statements in a macro expansion that generate code or data

The .LALL directive causes the assembler to list all statements in a macro expansion except the macro comments (see Section 10.5). The .SALL directive causes the assembler to suppress listing of *all* macro expansions. It lists only the macro invocation statements in the .LST file. The use of .SALL can substantially reduce the size of a .LST file. Use this directive when you don't want to see macro expansions.

The .XALL directive suppresses source statements such as comments, equates (EQU and =), etc. that do not generate code or data. It also suppresses repeat block directives (Section 10.8) and conditional assembly directives (Section 10.9).

The source file, which invokes the to_upper macro defined on page 386

```
          .
          .
.LALL
mov     AL,'a'
to_upper   AL
.SALL
mov     BL,AL
mov     AH,'1'
to_upper   AH
mov     BH,AH
mov     CL,'?'
.XALL
to_upper   CL
mov     DL,CL
          .
          .
```

generates the listing file shown in Figure 10.2.

When a macro is expanded, the number 1 appears in each of the expanded macro statements. This indicates the level of nesting. Since we have only a single level of nesting here, each macro expansion statement has a 1 at the left. Nested macros are discussed in Section 10.10.

```
     17 0000  B0 61                            mov    AL,'a'
     18                                        to_upper   AL
1    19                                        ; case conversion macro
1    20 0002  3C 61                            cmp     AL,'a'
1    21 0004  72 06                            jb      ??0000
1    22 0006  3C 7A                            cmp     AL,'z'
1    23 0008  77 02                            ja      ??0000
1    24 000A  2C 20                            sub     AL,32
1    25 000C                           ??0000:
     26 000C  8A D8                            mov     BL,AL
     27 000E  B4 31                            mov     AH,'1'
     28                                        to_upper   AH
     29 001D  8A FC                            mov     BH,AH
     30 001F  B1 3F                            mov     CL,'?'
     31                                        to_upper   CL
1    32 0021  80 F9 61                         cmp     CL,'a'
1    33 0024  72 08                            jb      ??0002
1    34 0026  80 F9 7A                         cmp     CL,'z'
1    35 0029  77 03                            ja      ??0002
1    36 002B  80 E9 20                         sub     CL,32
1    37 002E                           ??0002:
     38 002E  8A D1                            mov     DL,CL
```

Figure 10.2 An example listing file to illustrate the effect of list control directives.

10.8 Repeat Block Directives

There are three directives to repeat a block of statements—REPT, WHILE, IRP, and IRPC. These directives can be used both inside and outside a macro definition. These directives are mostly used to define and initialize variables in a data segment. Each directive identifies the beginning of a block of statements and ENDM indicates the end of a repeat block.

10.8.1 REPT Directive

The syntax of the REPT (REPeaT) directive is

```
REPT expression
    macro-body
ENDM
```

The macro-body is duplicated expression times. It is important to note that expression must evaluate to a constant at assembly time.

Example 10.10

The mult_by_16_8086 (on page 381) can be rewritten as

```
mult_by_16_8086  MACRO   operand
                 REPT 4
                     sal    operand,1
                 ENDM
                 ENDM
```

This type of code repetition use of the REPT directive is infrequent. A more common use is to generate look-up tables, as demonstrated in the next example.

Example 10.11

Suppose an application requires cubed data very frequently. Instead of using multiplication, we can precompute the data in a look-up table. Let us say we are interested in the first 10 integers. We can write

```
NUM_ENTRIES   EQU     10
.DATA
cube_table    LABEL   WORD
int_value = 0
REPT NUM_ENTRIES
    DW   int_value*int_value*int_value
    int_value = int_value+1
ENDM
```

The variable int_value created by using the = directive is used only at assembly time. Note that the = directive allows redefinition. This generates the data segment shown in Figure 10.3.

Here is a sample code to access this table, assuming that the integer to be cubed is in the SI register.

```
shl    SI,1          ; multiply SI by 2
mov    AX,cube_table[SI]
```

Since each entry in the table is a word (2 bytes), the integer to be cubed must be multiplied by 2 in order to use it as an index into the table.

```
      5 0000            .DATA
      6                 cube_table   LABEL    WORD
      7        = 0000   int_value = 0
      8                 REPT  NUM_ENTRIES
      9                         DW    int_value*int_value*int_value
     10                         int_value = int_value+1
     11                 ENDM
1    12 0000  0000              DW    int_value*int_value*int_value
1    13        = 0001           int_value = int_value+1
1    14 0002  0001              DW    int_value*int_value*int_value
1    15        = 0002           int_value = int_value+1
1    16 0004  0008              DW    int_value*int_value*int_value
1    17        = 0003           int_value = int_value+1
1    18 0006  001B              DW    int_value*int_value*int_value
1    19        = 0004           int_value = int_value+1
1    20 0008  0040              DW    int_value*int_value*int_value
1    21        = 0005           int_value = int_value+1
1    22 000A  007D              DW    int_value*int_value*int_value
1    23        = 0006           int_value = int_value+1
1    24 000C  00D8              DW    int_value*int_value*int_value
1    25        = 0007           int_value = int_value+1
1    26 000E  0157              DW    int_value*int_value*int_value
1    27        = 0008           int_value = int_value+1
1    28 0010  0200              DW    int_value*int_value*int_value
1    29        = 0009           int_value = int_value+1
1    30 0012  02D9              DW    int_value*int_value*int_value
1    31        = 000A           int_value = int_value+1
```

Figure 10.3 Data segment generated by the REPT directive example code.

10.8.2 WHILE Directive

The syntax of the WHILE directive is

```
WHILE  expression
       macro-body
ENDM
```

The `macro-body` is executed until the expression evaluates to false (zero). The expression is evaluated before each iteration of the macro body.

The previous example can be written using the WHILE directive as

```
WHILE int_value LT NUM_ENTRIES
```

```
        DW    int_value*int_value*int_value
        int_value = int_value+1
    ENDM
```

10.8.3 IRP and IRPC Directives

The directives

IRP — Iteration RePeat
IRPC — Iteration RePeat with Character substitution

both provide a means of supplying a variable parameter to each iteration of a repeat block. These two directives are similar except for how the variable parameter values are specified.

IRP Directive

The syntax of IRP is

```
IRP  parameter,<argument1 [,argument2, ...]>
     macro-body
ENDM
```

The angle brackets are required. The arguments are given as a list separated by commas. During the first iteration, argument1 is assigned to the parameter for use in the block of repeat statements; during the second iteration, argument2 is assigned, and so on. Therefore, the argument list specifies both the number of iterations and the actual values to be used in each iteration.

Example 10.12

The following code

```
.DATA
IRP value, <9,6,11,8,13>
    DB   value
ENDM
```

generates

```
.DATA
DB    9
DB    6
DB    11
DB    8
DB    13
```

Example 10.13

Suppose we want to generate a sequence of DB statements for the vowels in the order a, A, e, E, and so on. Suppose further that we want to write a macro in which we will list only the lowercase vowels and let the macro generate the uppercase vowel statements. Our first reaction would be to write the following macro:

```
;; This macro does not work
vowels    MACRO
          IRP   char, <a,e,i,o,u>
              DB    '&char'
              DB    %'&char'-32
          ENDM
          ENDM
```

Unfortunately, this does not work, as the expression evaluate operator can be used to provide an argument in a macro call. To write the desired macro, we have to follow a circuitous route as shown below:

```
defineDB  MACRO  value
DB    value
ENDM

IRP    char, <a,e,i,o,u>
defineDB '&char'
defineDB %'&char'-32
ENDM
```

Figure 10.4 shows the actual data segment statements generated by this code. Note that for the uppercase vowels, the ASCII equivalent decimal values are generated (for example, 65 for A, 69 for E, and so on).

IRPC Directive

The syntax of the IRPC directive is

```
IRPC parameter, string
     macro-body
ENDM
```

The macro-body is repeated once for each character in string. Like in the IRP directive, string specifies the number of iterations as well as the character to be used in each iteration.

```
1   13                          defineDB 'a'
2   14 0000  61                 DB    'a'
1   15                          defineDB %'a'-32
2   16 0001  41                 DB    65
1   17                          defineDB 'e'
2   18 0002  65                 DB    'e'
1   19                          defineDB %'e'-32
2   20 0003  45                 DB    69
1   21                          defineDB 'i'
2   22 0004  69                 DB    'i'
1   23                          defineDB %'i'-32
2   24 0005  49                 DB    73
1   25                          defineDB 'o'
2   26 0006  6F                 DB    'o'
1   27                          defineDB %'o'-32
2   28 0007  4F                 DB    79
1   29                          defineDB 'u'
2   30 0008  75                 DB    'u'
1   31                          defineDB %'u'-32
2   32 0009  55                 DB    85
```

Figure 10.4 Data segment to define vowels.

Example 10.14

The vowels macro in the previous example can be written using the IRPC directive by replacing

```
IRP    char, <a,e,i,o,u>
```

by

```
IRPC   char, aeiou
```

Example 10.15

This example generates code to test and set the zero flag if the character char in the AL register is an vowel—uppercase or lowercase.

```
                 .
                 .
        IRPC    char,aeiouAEIOU
            cmp    AL,'&char'
            je     finished
        ENDM
finished:
        jne    not_vowel
vowel:
                 .
                 .
not_vowel:
                 .
                 .
```

generates the following code:

```
                 .
                 .
        cmp    AL,'a'
        je     finished
        cmp    AL,'e'
        je     finished
                 .
                 .
        cmp    AL,'U'
        je     finished
finished:
        jne    not_vowel
vowel:
                 .
                 .
not_vowel:
                 .
                 .
```

This generates 10 cmp/je pairs of instructions. Obviously, this code is very long. Instead of using iteration at assembly time as in this example, iteration at run time can be used to reduce space.

10.9 Conditional Assembly

Both TASM and MASM provide conditional directives that allow assembly of a block of statements if the specified condition is true. There are situations in which conditional assembly helps generate efficient code, while in other situations, coding without using a conditional assembly directive is impossible. In this section, we will discuss selected conditional assembly directives and provide some examples that illustrate the points made here.

Both assemblers also provide conditional error directives that generate error messages if the specified condition is true. However, we will not discuss them here. Refer to the documentation that comes with the assembler.

We will now discuss the following types of conditional assembly directives.

Directive		Meaning
IF	IFE	Assembles if condition is true (IF) or false (IFE)
IFDEF	IFNDEF	Assembles if symbol is defined (IFDEF) or undefined (IFNDEF)
IFB	IFNB	Assembles if argument is blank (IFB) or not blank (IFNB)
IFIDN	IFDIF	Assembles if arguments are same (IFIDN) or different (IFDIF) — case sensitive
IFIDNI	IFDIFI	Assembles if arguments are same (IFIDNI) or different (IFDIFI) — case insensitive

The general syntax of conditional assembly directives is

```
IFxxx expression
   Tstatements
[ELSE
   Fstatements]
ENDIF
```

Each condition assembly directive ends with an ENDIF and there can be an optional ELSE clause present.

10.9.1 IF and IFE Directives

The syntax is

```
IF  expression
IFE expression
```

The IF directive assembles the then part if the expression evaluates to true (nonzero). The IFE directive is the IF counterpart and includes the then part if the expression is false (zero). Here are some operators that can be used in an expression:

Arithmetic operators:	$+$, $-$, $*$, $/$, mod, unary $+$ and $-$
Relational operators:	`EQ, GE, GT, LE, LT, NE`
Logical operators:	`NOT, AND, OR, XOR`

We now present two examples using these directives.

Example 10.16

The objective here is to write a macro that can perform left or right shift operation.

```
shift   MACRO   operand,count
        ;; positive count => left shift
        ;; negative count => right shift
        IFE count EQ 0
            IF count GT 0      ;; left shift
               shl   operand,count
            ELSE               ;; right shift
               ;; count is negative
               shr   operand,count
            ENDIF
        ENDIF
        ENDM
```

By defining this macro, we have effectively defined a new macro-instruction that can left or right shift efficiently using the processor shift family instructions.

Example 10.17

In Example 10.2 on page 381, we have written two macros—`Wmxchg` and `Bmxchg`—for exchanging the values of two memory variables. We can combine these two macros into a single macro by using the `IF` directive along with the TYPE operator. The TYPE operator returns the number of bytes reserved for the operand in memory. Table 10.1 shows the values returned by the TYPE operator.

Table 10.1 Values returned by TYPE operator

Type of memory operand	value returned
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	FFFFH
FAR	FFFEH
constant	0

```
mxchg  MACRO  operand1, operand2
       IF (TYPE operand1) NE (TYPE operand2)
          %OUT Operands of mxchg do not match.
       ELSE
            IF (TYPE operand1) EQ 1  ;BYTE operands
               xchg    AL,operand1
               xchg    AL,operand2
               xchg    AL,operand1
            ELSE                     ;WORD operands
               xchg    AX,operand1
               xchg    AX,operand2
               xchg    AX,operand1
            ENDIF
       ENDIF
       ENDM
```

This example uses a directive that we have not discussed: %OUT. The syntax of the %OUT directive is

```
%OUT    text
```

where text can contain any character. The directive displays text onscreen. It is useful in displaying error or warning messages during the assembly process as used in this example.

10.9.2 IFDEF and IFNDEF Directives

The syntax of these directives is

```
IFDEF  symbol
IFNDEF symbol
```

If symbol is defined (IFDEF) or not defined (IFNDEF), the conditional block of statements is assembled. These directives, however, will not test the value of symbol.

A common use of these directives is to customize code for a particular processor. For instance, assume that we want to save all the registers before starting a procedure so that they can be restored at the end of the procedure before return. If we are using an 80186 or later processors, the pusha instruction pushes all registers (AX, BX, CX, DX, SI, DI, BP, and SP) onto the stack. On the other hand, no such instruction is available in 8086; we have to push each of these registers individually.

We could maintain two versions of the source code—one for 8086 and the other for the later processors. But it creates problems in program maintenance, as any modification to the program has to be done on both versions to maintain consistency. Worse still, in other situations there may be more than two versions.

Using these conditional assembly directives, we can write a single source code which can generate the required version. The following two examples illustrate this point.

Example 10.18

We can define an assembly time variable PROC_TYPE to identify the processor type. Then we can use /D option (/D PROC_TYPE=486) to generate code targeted for the 80486 processor.

A problem arises if the /D option is not specified on the assembler command. This can be avoided by using the IFNDEF directive for default processor, as shown below.

```
IFNDEF  PROC_TYPE
        PROC_TYPE    EQU    8086
ENDIF
```

Then if no processor type is specified on the assembly command line, the default 8086 version is generated.

Example 10.19

Now we can write the required macro to generate the customized code that is dependent on the processor type.

```
pushall MACRO
```

```
IF PROC_TYPE EQ 8086
   push    AX
   push    BX
   push    CX
   push    DX
   push    SP
   push    BP
   push    SI
   push    DI
ELSE
   pusha
ENDIF
ENDM
```

10.9.3 IFB and IFNB Directives

The syntax of these directives is

```
IFB   <argument>
IFNB  <argument>
```

The angle brackets are required. The IFB directive assembles the conditional block of statements if argument is blank; IFNB assembles if argument is not blank.

These directives are useful to test the presence as well as the number of arguments specified in a macro call.

Example 10.20

Consider the mult_by_16 macro definition on page 380. If the macro is invoked by

```
mult_by_16
```

the macro is expanded as

```
sal     ,4
```

which causes the assembler to generate error for the sal instruction and you have no clue that this has been caused by an improper specification of the macro parameters. The assembler will not tell you this.

We can rewrite mult_by_16 to check for this and report appropriate errors at assembly time.

```
mult_by_16    MACRO    operand,extra
IFB   <operand>
   %OUT ERROR: No argument in mult_by_16
EXITM
ENDIF
IFNB  <extra>
   %OUT WARNING: Redundant argument in mult_by_16
ENDIF
sal   operand,4
ENDM
```

The IFB statement checks to make sure that the macro call specifies the argument. If not, we use the %OUT directive to write error messages on the screen and terminate the macro expansion process by using the EXITM directive. The EXITM directive stops any macro expansion or repeat block expansion that is in progress. All remaining statements after EXITM are ignored.

The IFNB statement checks to see if there are extra arguments specified in the macro call. If so, a warning message is displayed on the screen and the argument is ignored—the macro expansion is not stopped.

10.9.4 IFIDN and IFDIF Directives

The syntax of the IFIDN (IF IDeNtical) and IFDIF (IF DIFferent) directives is

```
IFIDN  <argument1>,<argument2>
IFDIF  <argument1>,<argument2>
```

The angle brackets are required. The condition block of statements are assembled if the arguments are identical (IFIDN) or different (IFDIF) character strings. These two directives are case sensitive (i.e., AX and Ax are not identical). The case insensitive versions of these directives are IFIDNI and IFDIFI (i.e, with these directives, AX and Ax are treated as identical).

Example 10.21

Let us write a macro that can receive a 16-bit value back. We want to design our macro such that the argument can be any general-purpose register or a memory operand. The macro tricky saves AX and BX registers at the beginning as the macro body uses these two registers in the macro body. These two registers are restored at the end.

```
;; incorrect version
tricky    MACRO    dest
```

```
;; dest can be a 16-bit register or memory
push    AX
push    BX
     .
  macro body
  uses AX and BX
     .
mov    dest,AX
pop    BX
pop    AX
ENDM
```

A problem with the `tricky` macro is that we cannot call this with either AX or BX as the argument. This problem is fixed in the `tricky1` macro that uses the `IFDIFI` directive to conditionally save and restore registers.

```
;; Correct version
tricky1    MACRO     dest
    IFDIFI <AX>,<dest>
        push    AX
    ENDIF
    IFDIFI <BX>,<dest>
        push    BX
    ENDIF
         .
       macro body
       uses AX and BX
         .
    IFDIFI <AX>,<dest>
        mov    dest,AX
        pop    AX
    ENDIF
    IFDIFI <BX>,<dest>
        pop    BX
    ENDIF
    ENDM
```

This is an example that cannot be coded to meet the requirements of the macro without using the `IFDIFI` directive. Note that all these directives introduce only assembly-time overhead but no run-time overhead. For more examples, see the `io.mac` file.

10.10 Nested Macros

Macros can also be nested. When macros are expanded, the nesting level is shown on the left of each expanded statement. This is illustrated in the following example.

Example 10.22

In this example, the shift macro (given on page 402) is rewritten using nested macros. This macro works for all of the 80X86 family of processors. In this macro, the common parts are replaced by the shifty macro.

```
shifty  MACRO   lr,operand,count
        IF PROC_TYPE EQ 8086
           IF count LE 4
              REPT count
              sh&lr   operand,1
              ENDM
           ELSE
              mov   CL,count
              sh&lr   operand,CL
           ENDIF
        ELSE
           sh&lr    operand,count
        ENDIF
        ENDM

shift   MACRO   operand,count
        ;; positive count => left shift
        ;; negative count => right shift
        IFE count EQ 0
            IF count GT 0      ;; left shift
               shifty  1,operand,count
            ELSE               ;; right shift
               ;; count negative
               shifty  r,operand,-count
            ENDIF
        ENDIF
        ENDM
```

Invoking this macro with

```
;*********************
```

```
shift   AX,2
;*********************
shift   count,-5
;*********************
```

generates the partial .LST file shown in Figure 10.5 for the 8086 processor.

The leftmost column gives the macro level. As shown on lines 45 and 46, the repeat directives are also counted as macros. The listing file would appear better if the .XLIST is used.

10.11 Performance: Macros Versus Procedures

Let us now look at the performance tradeoff between macros and procedures. We will use the bubble sort example discussed in Chapter 1. We will modify the assembly language procedure by replacing lines

```
xchg    AX,[SI+2]
mov     [SI],AX
```

by either a macro invocation or a procedure call to swap two elements.

Experiment 1

Here, our interest is in looking at the call/ret overhead associated with procedure calls. To do this, we will use the macro

```
mxchg   MACRO
        xchg    AX,[SI]
        xchg    AX,[SI+2]
        xchg    AX,[SI]
        ENDM
```

with the macro call mxchg replacing the two lines of code in the original assembly language procedure. Note that the three xchg instruction sequences exchange values at [SI] and [SI+2] while preserving the contents of the AX register.

The procedure version for this experiment assumes that the register SI holds the address of the first value to be exchanged. The address of the second value is assumed to be at SI+2. Thus, there is no overhead associated with passing parameters.

```
mxchg   PROC
        xchg    AX,[SI]
        xchg    AX,[SI+2]
```

```
    42                                    ;********************
    43                                    shift   AX,2
1   44                                    IFE 2 EQ 0
1   45                                        IF 2 GT 0
1   46                                           shifty  1,AX,2
2   47                                    IF PROC_TYPE EQ 8086
2   48                                       IF 2 LE 4
2   49                                          REPT 2
2   50                                          shl   AX,1
2   51                                          ENDM
3   52 0005  D1 E0                              shl   AX,1
3   53 0007  D1 E0                              shl   AX,1
2   54                                       ELSE
2   55                                          mov   CL,2
2   56                                          shl   AX,CL
2   57                                       ENDIF
2   58                                    ELSE
2   59                                       shl    AX,2
2   60                                    ENDIF
1   61                                        ELSE
1   62                                           shifty  r,AX,-2
1   63                                        ENDIF
1   64                                    ENDIF
    65                                    ;********************
    66                                    shift   count,-5
1   67                                    IFE -5 EQ 0
1   68                                        IF -5 GT 0
1   69                                           shifty  1,count,-5
1   70                                        ELSE
1   71                                           shifty  r,count,--5
2   72                                    IF PROC_TYPE EQ 8086
2   73                                       IF --5 LE 4
2   74                                          REPT --5
2   75                                          shr   count,1
2   76                                          ENDM
2   77                                       ELSE
2   78 0009  B1 05                              mov   CL,--5
2   79 000B  D3 2E 0000r                        shr   count,CL
2   80                                       ENDIF
2   81                                    ELSE
2   82                                       shr    count,--5
2   83                                    ENDIF
1   84                                        ENDIF
1   85                                    ENDIF
    86                                    ;********************
```

Figure 10.5 Partial listing file showing nested macro expansions.

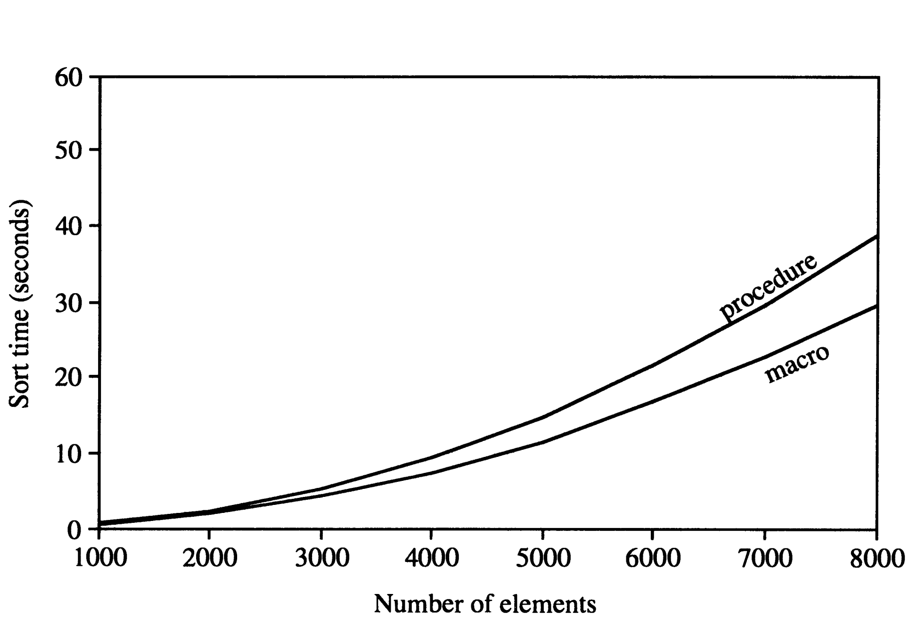


Figure 10.6 Performance impact of `call/ret` overhead.

```
        xchg    AX,[SI]
        ret
mxchg   ENDP
```

The procedure call

```
call    mxchg
```

replaces the two lines of code in the original assembly language procedure.

Figure 10.6 shows the performance of these two versions. The x-axis gives the number of elements to be sorted, while the y-axis gives the time required to sort the input array in ascending order. The input array is initially sorted in descending order. The data suggests that the `call/ret` overhead in this example is about 25–30 percent.

Experiment 2

The procedure `mxchg` used in the last experiment has no parameters. In general, we write procedures that receive parameters via the stack. In this experiment we will modify the procedure so that it receives the parameters via the stack.

The two parameters required by the procedure are the pointers to the values in memory that are to be swapped. The modified procedure is:

```
mxchg   PROC
        push    BP
        mov     BP,SP
        push    SI
        mov     SI,[BP+4]
        xchg    AX,[SI]
        xchg    AX,[SI+2]
        xchg    AX,[SI]
        pop     SI
        pop     BP
        ret     2
mxchg   ENDP
```

We will call this procedure with

```
push    SI
call    mxchg
```

The macro is revised to correspond to the modified procedure definition. The revised macro is:

```
mxchg   MACRO   ptr1
        push    SI
        mov     SI,ptr1
        xchg    AX,[SI]
        xchg    AX,[SI+2]
        xchg    AX,[SI]
        pop     SI
        ENDM
```

In our example, we can invoke this macro by

```
mxchg    SI
```

The difference between the macro and procedure versions is that the procedure version includes the overhead in retrieving the parameters from the stack and the related house keeping activities (such as saving BP, etc.).

Figure 10.7 plots the performance of these versions. The difference between the macro and procedure versions in this experiment increases to about 50 percent—about twice that of the corresponding difference in Experiment 1. The additional difference over that of Figure 10.6 is attributable to parameter passing via the stack.

While there is no reason to use either a macro or a procedure to swap two values in this example, it does serve our purpose of measuring the associated overheads.

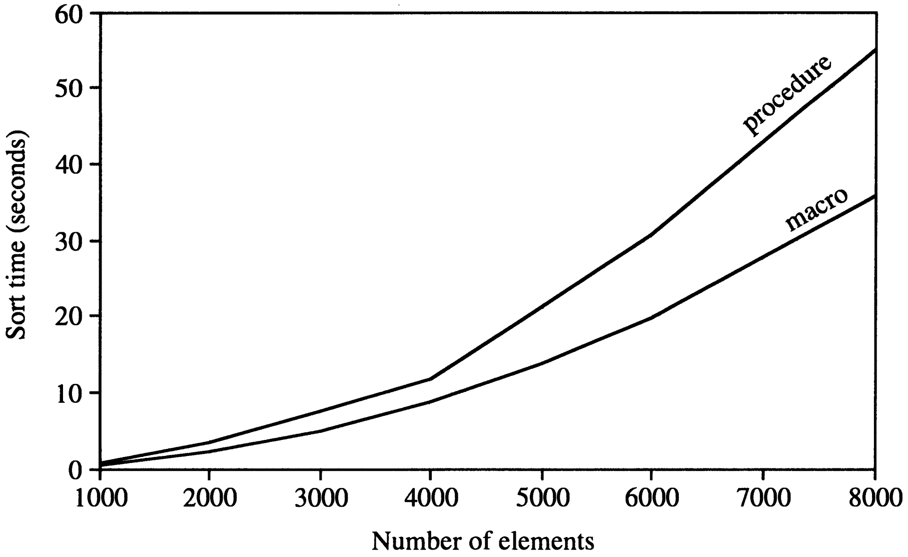


Figure 10.7 Performance impact of `call/ret` and parameter passing overheads.

10.12 Summary

A macro is a sophisticated text substitution mechanism. Assemblers provide three directives to support the macro definition. The directives = and EQU are primitive in the sense that they can be used only for defining constants whose values are available at assembly time. For sophisticated text substitution, we have to rely on the MACRO directive. Each macro definition should end with an ENDM directive. Macros can be defined, like procedures, with parameters as well.

Both macros and procedures can be used to write modular programs. There are, however, some inherent tradeoffs between the two. Macros tend to increase space requirements of a program, as they replace each macro call by the actual code. Procedures, on the other hand, maintain a single copy of the code and the control is transferred to this copy whenever a procedure is called. The transferring of control typically involves two types of overhead:

1. `call/ret` overhead to transfer control back and forth between the caller and callee;

2. parameter passing overhead, which usually involves the stack.

The last section looked at the performance tradeoffs in the context of the bubble sort example.

We have discussed five list control directives—`.LIST`, `.XLIST`, `.LALL`, `.SALL`, and `.XALL`. The first two directives are general-purpose, while the last three directives affect only macro invocations.

Both TASM and MASM assemblers provide three directives to repeat a block of statements—`REPT`, `IRP`, and `IRPC`. These directives can be used both inside and outside of macro definitions. We have demonstrated the use of these directives by means of several examples.

Several conditional assembly directives are available to conditionally include statements in the assembly process. Conditional assembly directives are useful in generating customized code from a single source program. Furthermore, we have shown that in certain cases, coding without using a conditional assembly directive is impossible.

10.13 Exercises

10–1 Discuss the advantages and disadvantages of macros.

10–2 Discuss the advantages and disadvantages of procedures.

10–3 What are the three assembler directives that allow macro definition? Discuss the differences and similarities among them.

10–4 What are the advantages of allowing parameters in a macro definition?

10–5 What is the purpose of `EXITM`?

10–6 Explain a scenario where a parameter type can be used as a macro parameter but not as a procedure parameter.

10–7 Discuss how the inherent tradeoffs between macros and procedures are affected by the current technology.

10–8 What are the two run-time overheads of a procedure call? How do macros avoid these overheads?

10–9 Why should comments receive special attention in macro definition?

10–10 In a procedure body, we can define labels without any problem. Explain why defining labels in a macro definition causes a problem and how we can avoid this problem.

10–11 What is the difference between the macro operators `< >` and `!`?

10–12 What is the difference between the `.SALL` and `.XALL` directives?

10–13 *The* `IRP` *directive can be used to replace the* `IRPC` *directive.* Do you agree with this statement? Justify your answer.

10–14 *The* `IRPC` *directive can be used to replace the* `IRP` *directive.* Do you agree with this statement? Justify your answer.

10–15 Describe a macro definition that cannot be written without using IFIDN/IFDIF or IFIDNI/IFDIFI directives.

10–16 Explain why the results of Experiment 1 of Section 10.11 did not include parameter passing overhead.

10.14 Progamming Exercises

10–P1 Define a macro to multiply two unsigned integers.

```
mult    dest,src
```

The operands of this macro are memory variables. If the result cannot be stored in dest, the carry flag should be set to indicate overflow; otherwise, the carry flag should be cleared. Assume that dest and src operands are both byte, word, or doubleword variables.

10–P2 Write a macro to implement the if-then construct. The semantics are

```
if (ZF = 1)
then
    execute then_proc
end if
```

The macro is invoked by

```
if_z_then    then_proc
```

where then_proc is the procedure name that will be executed if ZF = 1.

10–P3 Write a macro to implement the if-then-else construct. The semantics are

```
if (ZF = 1)
then
    execute then_proc
else
    execute else_proc
end if
```

The macro is invoked by

```
if_z_then_else    then_proc, else_proc
```

where then_proc is the procedure name that will be executed if ZF = 1, and else_proc is the procedure name that will be executed if ZF = 0.

10–P4 Write a macro to implement the while_z construct. The semantics are

```
while (ZF = 1)
    execute proc
end while
```

The macro is invoked by

```
while_z    proc
```

where `proc` is the procedure name that will be executed while ZF = 1.

10–P5 Pentium procedure calls are unconditional, unlike jumps. Write a conditional call macro-instruction that calls a procedure `proc` if ZF = 1. This macro-instruction is invoked as

```
call_z    proc
```

10–P6 Write a macro `Bmult10` that multiplies the source operand `src` by 10 and stores the result in the `dest` operand. It is invoked as

```
Bmult10    dest,src
```

which performs

```
dest := src × 10
```

You should use only shifts and adds. Try to do it using only three 1-bit left shifts and one add operation. For this exercise, registers need not be preserved.

10–P7 Modify the above macro to preserve registers. Also, make sure that your macro works properly when the `src` and `dest` operands are either registers, memory locations, or a combination of the two.

10–P8 Write a macro `mult10` that works for byte, word, and doubleword operands. The macro `mult10` performs a similar function as the `Bmult10` macro in the last example.

10–P9 All conditional jumps are short jumps. For example, `ja` can only jump to a label within approximately ±127 bytes. Write a macro `Lja` that can perform `ja` to any label in the current segment. The macro is invoked as

```
Lja    label
```

10–P10 Generalize the macro you wrote for the last question. Write a macro `Ljcond` that works for any conditional jump instruction. It receives two parameters: `cond` to identify the jump condition, and `label` of the target instruction. The macro, for example, can be invoked as

```
Ljcond    ge,target
```

to cause a long jump `jge` to `target` location.

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 Pentium instructions that support arithmetic in ASCII and BCD representations
- To discuss the tradeoffs among the binary, ASCII, and BCD representations

We discuss the binary number system in Appendix A. 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/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

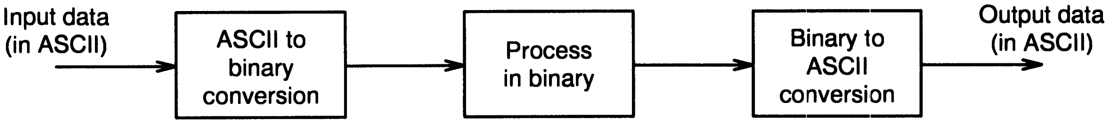
S. P. Dandamudi, *Introduction to Assembly Language Programming*

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 have 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/printed, they should be in ASCII form. Thus, it is necessary to convert numbers from ASCII to binary at the input end and, again, to convert back to the ASCII form to output results, as shown below:



We have 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 will 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 sign. But if a sign representation is needed, an additional byte can be used to associate a sign with a BCD number. 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, 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

Pentium provides four instructions to process numbers in 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 either by using 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 results in AL to an ASCII result, we have to insert 3 into the upper four bits of AL. □□□□□□□

To add multiple digit decimal numbers, we have to use a loop that adds one digit at a time starting from the rightmost digit. Program 11.36 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 Program 11.36).

Program 11.36 ASCII addition of two 10-digit numbers

```
TITLE   Addition of two integers in ASCII form   ASCIIADD.ASM
COMMENT |
        Objective: To demonstrate addition of two integers
                   in the ASCII representation.
            Input: None.
|          Output: Displays the sum.
.MODEL SMALL
.STACK 100H
.DATA
sum_msg   DB  'The sum is: ',0
number1   DB  '1234567890'
number2   DB  '1098765432'
sum       DB  10 DUP (' '),0 ; add NULL char. to use PutStr

.CODE
```

```
16:  INCLUDE io.mac
17:  main  PROC
18:        .STARTUP
19:        ; SI is used as index into number1, number2, and sum
20:        mov     SI,9            ; SI points to rightmost digit
21:        mov     CX,10           ; iteration count (# of digits)
22:        clc                     ; clear carry (we use ADC not ADD)
23:  add_loop:
24:        mov     AL,number1[SI]
25:        adc     AL,number2[SI]
26:        aaa                     ; ASCII adjust
27:        pushf                   ; save flags because OR
28:        or      AL,30H          ;  changes CF that we need
29:        popf                    ;  in the next iteration
30:        mov     sum[SI],AL      ; store the sum byte
31:        dec     SI              ; update SI
32:        loop    add_loop
33:        PutStr  sum_msg         ; display sum
34:        PutStr  sum
35:        .EXIT
36:  main  ENDP
37:        END     main
```

The program adds two numbers number1 and number2 and displays the sum. We use SI as an index into the input numbers, which are in ASCII representation. SI is initialized to point to the rightmost digit (line 20). The loop count (10) is set up in CX (line 21). The addition loop (lines 23–32) adds one digit by taking any carry generated by the previous iteration into account. This is done by using 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 22 using the clc (clear carry) instruction.

Note that the aaa instruction produces the result in unpacked BCD form. To convert to ASCII form, we have to or the result with 30H (line 28). 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 27) and restore (line 29) 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.

```
29H = 00101001B
69H = 01101001B
---   ---------
92H = 10010010B
```

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.

```
27H = 00100111B
34H = 00110100B
---   ---------
5BH = 01011011B
```

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.

```
52H = 01010010B
61H = 01100001B
B3H = 10110011B
```

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 an add or adc. 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 11.37.

Program 11.37 Packed BCD addition of two 10 digit numbers

```
TITLE   Addition of integers in packed BCD form   BCDADD.ASM
COMMENT |
        Objective: To demonstrate addition of two integers
                   in the packed BCD representation.
            Input: None.
|          Output: Displays the sum.
SUM_LENGTH    EQU    10
.MODEL SMALL
.STACK 100H
.DATA
sum_msg   DB  'The sum is: ',0
number1   LABEL   BYTE
          DT  1234567890    ; stores in packed BCD form
number2   LABEL   BYTE
```

```
          DT  1098765432    ; stores in packed BCD form
BCDsum    LABEL   BYTE
          DT  ?
ASCIIsum  DB  SUM_LENGTH DUP (' '),0 ; add NULL char.

.CODE
.486
INCLUDE io.mac
main  PROC
      .STARTUP
      sub     SI,SI
      mov     CX,5           ; loop iteration count
      clc                    ; clear carry (we use ADC)
add_loop:
      mov     AL,number1[SI]
      adc     AL,number2[SI]
      daa                    ; ASCII adjust
      mov     BCDsum[SI],AL  ; store the sum byte
      inc     SI             ; update index
      loop    add_loop
      call    ASCII_convert
      PutStr  sum_msg        ; display sum
      PutStr  ASCIIsum
      .EXIT
main  ENDP
;-----------------------------------------------------------
; Converts the packed decimal number (5 digits) in BCDsum
; to ASCII represenation and stores it in ASCIIsum.
; All registers are preserved.
;-----------------------------------------------------------
ASCII_convert PROC
      pusha                  ; save registers
      ; SI is used as index into ASCIIsum
      mov     SI,SUM_LENGTH-1
      ; DI is used as index into BCDsum
      sub     DI,DI
      mov     CX,5           ; loop count (# of BCD digits)
cnv_loop:
      mov     AL,BCDsum[DI]  ; AL := BCD digit
      mov     AH,AL          ; save the BCD digit
      ; convert right digit to ASCII & store in ASCIIsum
      and     AL,0FH
      or      AL,30H
      mov     ASCIIsum[SI],AL
```

```
59:         dec     SI
60:         mov     AL,AH           ; restore the BCD digit
61:         ; convert left digit to ASCII & store in ASCIIsum
62:         shr     AL,4            ; right shift by 4 positions
63:         or      AL,30H
64:         mov     ASCIIsum[SI],AL
65:         dec     SI
66:         inc     DI              ; update DI
67:         loop    cnv_loop
68:         popa                    ; restore registers
69:         ret
70:   ASCII_convert ENDP
71:         END     main
```

Storage space for the two input numbers and for the sum (BCDsum) is allocated by using the DT (Define Ten-byte) assembler directive. With this storage allocation, we can represent numbers in the range

±999,999,999,999,999,999

When initializing using DT, the values are stored in packed BCD representation in byte reverse order. The digits within a byte are, however, not reversed. For example,

```
DT    1234567890
```

is stored as

```
90 78 56 34 12H
```

Since we want to process these numbers one byte at a time, we use the LABEL directive on lines 12, 14, and 16. Note that we could have as well used the DB directive with "proper" initialization. However, we want to demonstrate the use of DT in initializing packed BCD numbers.

The code is similar to that given in Program 11.36. However, since we add two decimal digits during each loop iteration, only five iterations are needed to add two 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).

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

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.

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 also required to be in ASCII form, 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.

In order to assess these tradeoffs, we have conducted two experiments. In the first experiment, two numbers are received as input in the ASCII form. However, to eliminate variable delays associated with entering numbers through the keyboard, these two numbers are not given from the keyboard. Rather, the five-character string is read directly from memory. In all cases, the sum is stored in ASCII form in memory, as in Program 11.36. In ASCII addition, no conversion is needed either at the input end or on the output side (see Table 11.1). However, to add the two decimal numbers, we have to do it byte by byte in a loop (see Program 11.36). In this example, the loop iterates five times, which reduces processing speed.

Experiment 1

The objective of this experiment is to see the impact of conversion overhead. In binary addition, we have to first convert the ASCII input into binary form. Again, after addition, the sum has to be converted back to ASCII form. For this, we have modified procedures `proc_PutInt` and `proc_GetInt`. The advantage of performing binary addition is that it is the fastest of the three representations.

The packed BCD representation also requires conversion from ASCII representation at the input and to ASCII representation at the output. However, this conversion is simpler than the binary conversion, which requires arithmetic multiplication and division. Conversion to BCD can be done by using only logical and shift operations. For an example, see procedure `ASCII_convert` in Program 11.37 for conversion from packed BCD to ASCII form.

Figure 11.1 shows the execution time to perform a single addition of two five-digit decimal numbers given in ASCII form. The resulting sum is also stored in ASCII form. The x-axis gives the number of calls to the procedure in thousands and the y-axis gives the execution time in seconds. Due to high overhead in converting numbers between ASCII and binary, the binary version (see *binary add* line) takes more than three times that taken by 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. The data in this figure show the impact of conversion overhead. Thus, if an application reads data in ASCII form and performs simple arithmetic and produces output in ASCII form, doing it in ASCII is better than in the other two forms.

Experiment 2

In this experiment, let us see if the relative performance changes when we perform a number of arithmetic operations between input and output. To conduct such an experiment, we have modified the code of the previous experiment such that it performs ten additions on the same two input decimal numbers. For

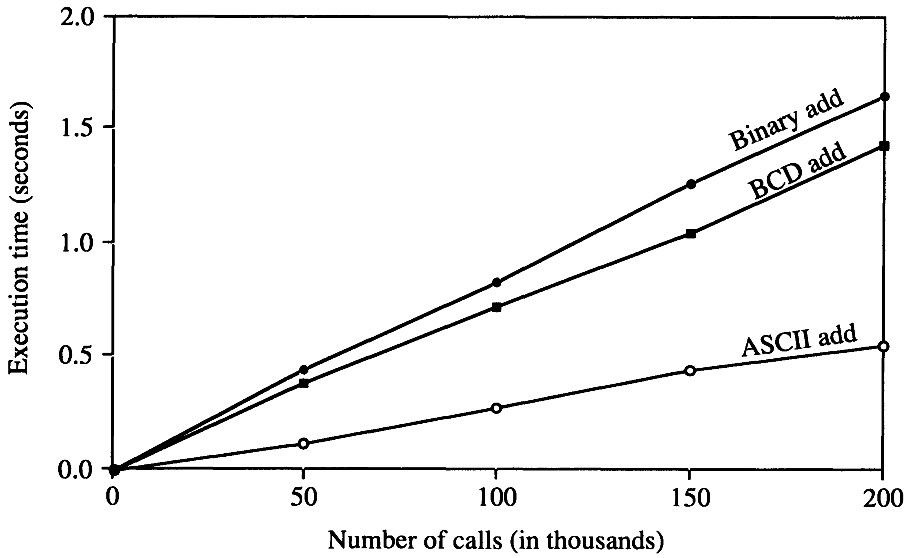


Figure 11.1 Performance tradeoffs when the conversion overhead dominates.

example, in the binary version, after converting the two decimal numbers from ASCII to binary, it performs ten additions using the add instruction and finally converts the sum back to ASCII.

The results of this experiment are shown in Figure 11.2. In this case, the relative performance is the converse of that in Figure 11.1. The binary version performs the best, while the ASCII worst. In fact, the ASCII version is about 2.5 times slower than the binary version. The BCD version, on the other hand, is only about 50 percent slower than the binary version. In this experiment, processing speed dominates the conversion overhead. Thus, binary form is preferred in applications where the data comes in binary form or if number processing is not simple.

A Final Word

The last observation is strictly not true. Suppose that we are using long integers (32 bits) rather than the 16-bit numbers, as in the last two experiments. If the processor were to support only 16-bit multiplication or division operations, the conversion from ASCII to binary and vice versa would cause very high overhead. As a result, the conversion overhead is substantial, as the required

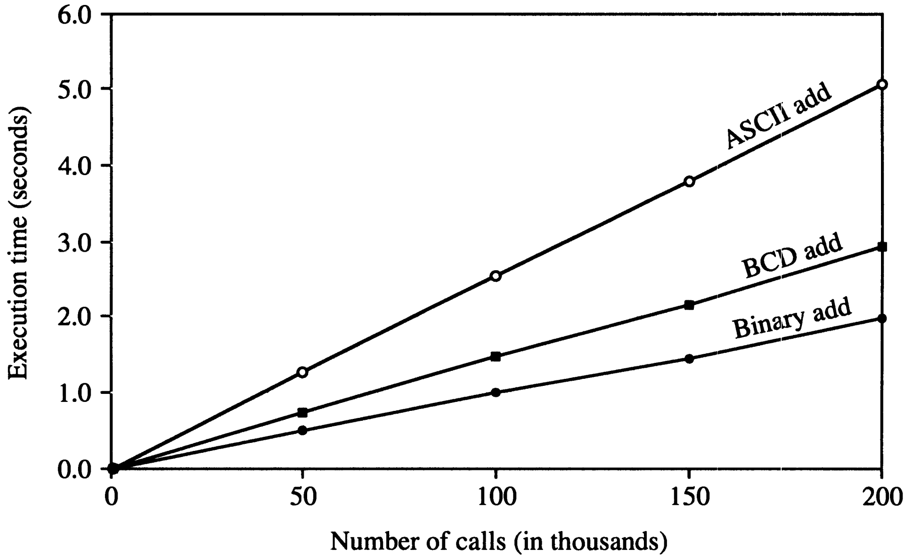


Figure 11.2 Performance tradeoffs when number processing dominates.

multiplication and division should be done in software. For example, for 32-bit data, the execution times for 10,000 calls to perform a single addition, as in Experiment 1, are:

Binary version:	2.64 seconds
Packed BCD version:	0.11 seconds
ASCII version:	0.055 seconds

In this case, ASCII representation is the best choice for performing the arithmetic. Of course, Pentium supports 32-bit arithmetic operations that can eliminate the additional overhead. However, a similar situation arises when we are considering 64-bit data.

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 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 representations. 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 demonstrated these tradeoffs using an example.

11.6 Exercises

11–1 Briefly give the reasons for using either ASCII or BCD representations.

11–2 How is 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) 32025
(c) 2491
(d) 4385

11–5 Explain why `pushf` and `popf` instructions are needed in Program 11.36.

11–6 Assuming that an ASCII digit is in the AL register, write an assembly language code fragment to convert it to unpacked BCD representation.

11–7 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.

11–8 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.

11–9 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
```

11–10 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
```

11–11 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
```

11–12 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
```

11–13 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
```

11–14 Discuss the conversion versus processing overhead tradeoffs associated with the binary, ASCII, and BCD (both packed and unpacked) representations.

11.7 Progamming Exercises

11–P1 Modify `asciiadd.asm` (Program 11.36) to read two decimal numbers from the user instead of taking them from memory. The two numbers from the user should be read as a string. You can use `GetStr` to read the input numbers.

11–P2 Modify `asciiadd.asm` (Program 11.36) 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 a string using `GetStr`.

11–P3 Modify `bcdadd.asm` (Program 11.37) to receive two decimal numbers from the user instead of taking them from memory. The two numbers from the user should be read as an ASCII string using `GetStr`. The input numbers should be converted to the packed BCD representation for performing the addition, as in Program 11.37. The result should be converted back to ASCII so that it can be displayed by `PutStr`.

11–P4 Modify the program for the last exercise to perform subtraction.

11–P5 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 (and read by your program using `GerStr`). The result should be displayed by using `PutStr`.

Hint: You need to use a loop that mimics the behavior of longhand multiplication (i.e., multiply one digit at a time).

11–P6 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 (and read by your program 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.

Chapter 12

Interrupts and Input/Output

Objectives

- To describe the interrupt mechanism of Pentium
- To explain software and hardware interrupts
- To discuss DOS and BIOS interrupt services to interact with I/O devices such as the keyboard, printer, and display screen
- To illustrate writing user defined interrupt service routines
- To provide an understanding of some peripheral support chips
- To discuss how programmed input/output is done with I/O ports using `in` and `out` instructions
- To describe the polling mechanism and the associated overheads

Interrupts, like procedures, can be used to alter a program's flow of control to a procedure called 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). Interrupts are introduced in Section 12.1, and Section 12.2 discusses a taxonomy of interrupts. The interrupt invocation mechanism is described in Section 12.3.*

Both DOS and BIOS provide several software interrupt service routines. Software interrupts are introduced in Section 12.4. Section 12.5 discusses the keyboard services of DOS and BIOS. The next section discusses the DOS

S. P. Dandamudi, *Introduction to Assembly Language Programming*

interrupt services for text output on the display screen. Printer services of DOS and BIOS are described in Section 12.7. Section 12.8 discusses exceptions.

Hardware interrupts are introduced in Section 12.9, and these are discussed for the remainder of the chapter. 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 of accessing involves direct I/O access. This method of I/O device access 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 I/O ports using `in` *and* `out` *instructions to access I/O ports. Sections 12.10 through 12.12 also discuss this topic.*

Polling is an alternative to interrupts and is useful in some situations. Polling, however, introduces overhead. The impact of this overhead is studied in Section 12.13. The last section summarizes the chapter.

12.1 Introduction

Interrupt is a mechanism by which a program's flow of 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 original program resumes execution as if it were not interrupted. This behavior is analogous to a procedure call. There are, however, some basic differences between procedures and interrupts that make interrupts almost indispensable.

One of the main differences is that interrupts can be initiated by both software and hardware. In contrast, procedures are purely software-initiated. The fact that interrupts can be initiated by hardware is the principal factor behind the power of the interrupt mechanism. This capability gives us an efficient way by which external devices (outside the CPU) can get the attention of the CPU.

Software-initiated interrupts—called simply *software interrupts*—are caused by executing 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 the immediate attention of the CPU? For example, you have written a program to display the first 90 Fibonacci numbers on the screen. While running the program, however, you have realized that your program never terminates because of a simple programming mistake (e.g., you forgot to increment the index variable controlling the loop). Obviously, you want to abort the program and return control to the operating system. As you know, in most cases this can be done by `ctrl-break`. For this example, `ctrl-break` certainly works. The important point is that this is not an anticipated event—so cannot be programmed into the code. Strictly speaking, we can include code to handle all possible events, or at least most likely events, but such an alternative makes the program very inefficient.

The interrupt mechanism provides an efficient way to handle unanticipated events. Referring to the previous example, the `ctrl-break` could cause an interrupt to draw the attention of the CPU away from the user program. The interrupt service routine associated with `ctrl-break` can terminate the program and return control to the operating system.

Another difference between procedures and interrupts is that ISRs are normally memory-resident. In contrast, procedures—including library routines—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.

12.2 A Taxonomy of Interrupts

We have already identified two basic categories of interrupts—software-initiated and hardware-initiated (see Figure 12.1). The third category is called *exceptions*. Exceptions handle instruction faults. An example of an exception is the divide error fault, which is generated whenever divide by 0 is attempted. This error condition occurs during the `div` or `idiv` instruction if the divisor is 0. Later we see an example of this fault in Section 12.8, which discusses exceptions in detail.

Software interrupts are written into a program by using the `int` instruction. The main use of software interrupts is in accessing I/O devices such as a keyboard, printer, display screen, disk drive, etc. Software interrupts can be further classified into *system-defined* and *user-defined*. There are two types of system-defined software interrupts: interrupt services supported by DOS, and those supported by BIOS (Basic Input/Output System). The BIOS is the lowest-level software that is stored in ROM (read-only memory). Note that DOS and other application software is loaded from disk.

The interrupt service routines provided by DOS and BIOS are not mutually exclusive. There are some services, such as reading a character from the key-

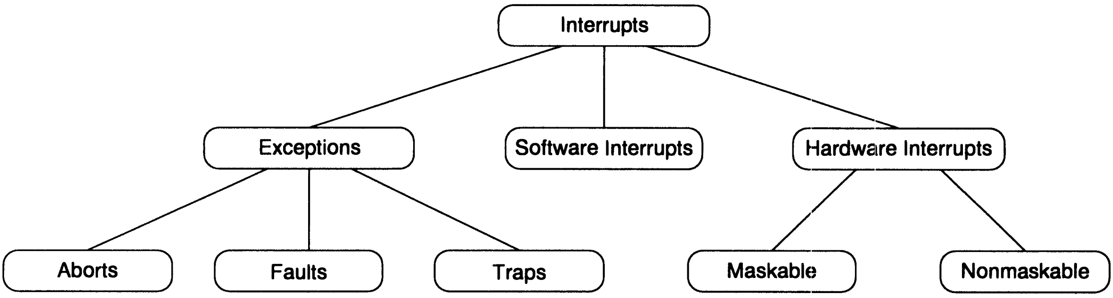


Figure 12.1 A taxonomy of Pentium interrupts.

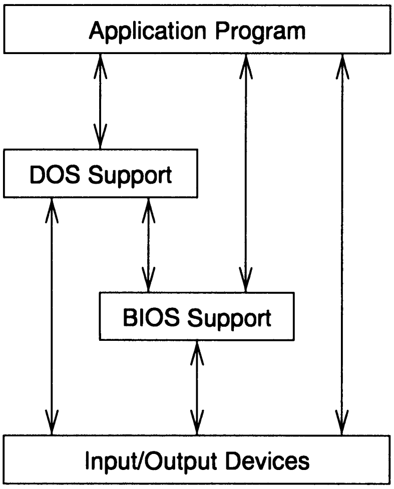


Figure 12.2 Various ways of interacting with I/O devices.

board, provided by both DOS and BIOS. In fact, DOS uses BIOS-supported routines to provide some services that control the system hardware (see Figure 12.2).

Hardware interrupts are generated by hardware devices to get the attention of the CPU. For example, when you strike a key, the keyboard hardware generates an external interrupt causing the CPU to suspend its present activity and execute the keyboard interrupt service routine to process the key. After completing the keyboard ISR, the CPU resumes what it was doing before the interruption.

Hardware interrupts can be either *maskable* or *nonmaskable*. The nonmaskable interrupt (NMI) is always attended to by the CPU immediately. Note that when we say immediate, the CPU does not suspend the execution of the current instruction in the middle. It completes the current instruction and then services the interrupt. 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 CPU is executing a `main` program. An interrupt occurs. As a result, the CPU suspends the `main` as soon as it finishes the current instruction of `main` and the control is transferred to the ISR1. If ISR1 has to be executed without any interruption, the CPU can mask further interrupts until ISR1 is completed. Suppose that, while executing ISR1, another maskable interrupt occurs. Service to this interrupt would have to wait until ISR1 is completed.

12.3 Interrupt Processing

12.3.1 Interrupt Processing in 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. Interrupt types range between 0 and 255. The interrupt type number is used as an index into a table that stores the addresses of ISRs. This table is referred to as the *interrupt descriptor table* (IDT). Like the global and local descriptor tables (GDT and LDT, as discussed in Chapter 2), each descriptor (or vector as they are often called) is essentially a pointer to an ISR and requires 8 bytes. The interrupt type number is scaled by 8 to form an index into the IDT.

The IDT may reside anywhere in physical memory. The location of the IDT is maintained in an IDT register IDTR. The IDTR is a 48-bit register that stores 32 bits of IDT base address and a 16-bit IDT limit value. However, the IDT does not require more than 2048 bytes, as there can be at most 256 descriptors. In a system, the number of descriptors could be much smaller than the maximum allowed. In this case, the IDT limit can be set to the required size. If a descriptor is referenced that is outside the limit, the processor enters shutdown mode.

There are two special instructions to load (`lidt`) and store (`sidt`) the contents of the IDTR register. Both instructions take the address of a 6-byte memory as the operand. In the next subsection, we describe interrupt processing in the real mode, which is the focus of this chapter.

12.3.2 Interrupt Processing in Real Mode

In real mode, the IDT is located at base address 0. Each vector takes only 4 bytes as opposed to 8 bytes in the protected mode. Each vector consists of a CS:IP pointer to the associated ISR: 2 bytes for specifying the code segment (CS), and 2 bytes for the offset (IP) within the code segment. Figure 12.3 shows the interrupt vector table layout in the memory.

Since each entry in the interrupt vector table is 4 bytes long, interrupt type is multiplied by 4 to get the corresponding ISR pointer in the table. For example, `int 2` can find the ISR pointer at memory address 2×4 = 00008H. The first 2 bytes at the specified address are taken as the offset value and the next 2 bytes as the CS value. Thus, executing `int 2` causes the CPU to suspend its current program and calculate the address in the interrupt vector table (which is 2×4 = 8 for this example) and read CS:IP values and transfer control to that memory location.

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. 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
```

When an interrupt occurs, the following actions are taken:

1. Push flags register onto the stack
2. Clear interrupt enable and trap flags
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).

Note that EIP is used instead of IP for 32-bit segments. On receiving an interrupt, the flags register is automatically saved on the stack. The interrupt

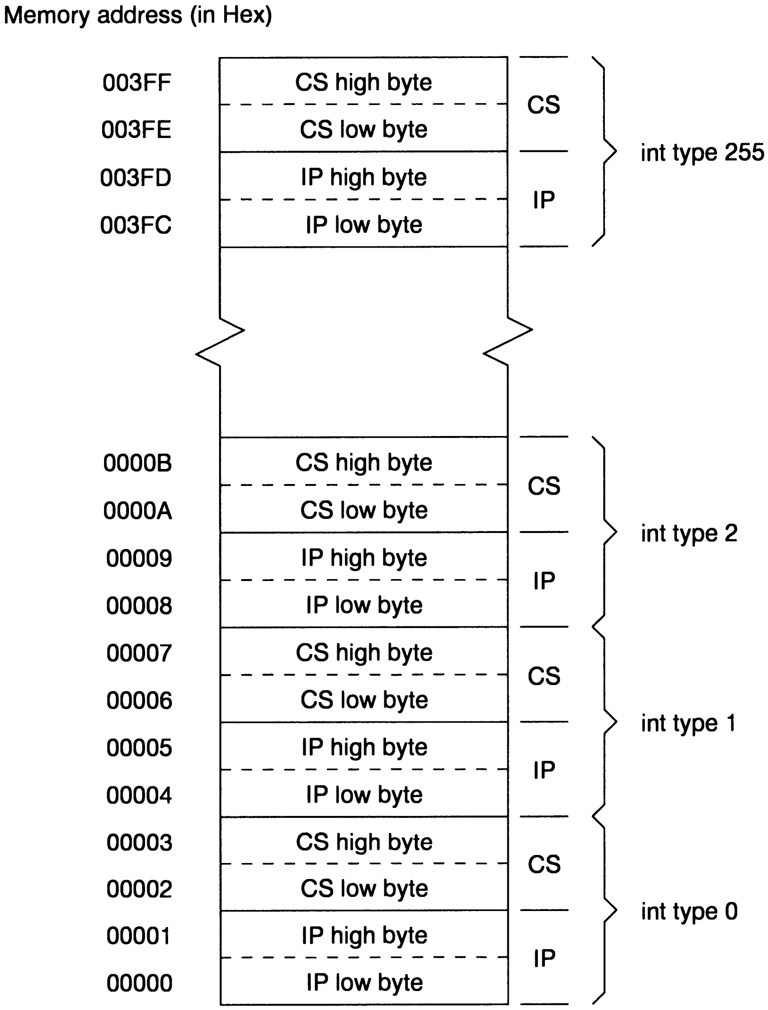


Figure 12.3 Interrupt vector table in memory (real mode).

enable flag is cleared. This disables attending further interrupts until this flag is set. 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.

The current CS and IP values are pushed onto the stack. In most cases, these values (i.e., CS:IP) point to the instruction following the current instruction. (See Section 12.8 for a discussion of a different behavior in case of a fault.) If it is a software interrupt, CS:IP points to the instructions following the `int` instruction. The CS and IP registers are loaded with the address of the ISR from the interrupt vector table.

The last instruction of an ISR is the `iret` instruction and serves the same purpose as `ret` for procedures. The actions taken on `iret` are:

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.

12.4 Software Interrupts

Software interrupts are initiated by executing an interrupt instruction. The format of this instructions 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 are possible. This is a sufficiently large number, as each interrupt type can be parameterized to provide several services. For example, all DOS services are provided by `int 21H`. There are more than 80 different services (called functions) provided by DOS. All these functions are invoked by `int 21H`. Registers are used to pass parameters and to return values. The required function number is placed in the AH register. Any required additional parameters should be placed in specific registers. Usually, the function also returns values in registers. We will discuss some of the `int 21H` services later in this chapter.

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 example how they can be used. We organize our discussion around the I/O devices. The following I/O devices are considered: the keyboard, display screen for text data, and printer.

12.5 Keyboard Services

12.5.1 Keyboard Description

Associated with each I/O device, there is a *device controller* or *I/O controller* that acts as a 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 CPU 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. On your PC, every time a key is depressed or released, the 8259 interrupt controller generates a hardware interrupt `int 09H`. 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 buffer has the capacity to store up to 15 keys. When the buffer is full, pressing a key causes the BIOS to beep, indicating that the key stroke is lost.

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 12.1 shows the scan codes for the IBM-PC keyboard.

The scan code of a key does not have any relation to the ASCII code of the corresponding character. The `int 09H` 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, as opposed to the last-in-first-out (LIFO) structure of the stack. 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.

12.5.2 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

Table 12.1 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			

number. The following seven functions are provided by DOS for interacting 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 entered

This function can be used to read a character from the keyboard buffer. If the keyboard buffer is empty, this function waits until a character is typed. The received character is echoed to the display screen. If the character is a `ctrl-break`, an interrupt 23H is invoked, which aborts the program.

Function 06H — Direct console I/O
There are two sub-functions associated with this function—keyboard input or character display. The DL register is used to specify the desired sub-function.

Sub-function — Keyboard Input

Inputs: AH = 06H
DL = FFH
Returns: ZF = 0 if a character is available
In this case, the ASCII code of the key
entered is placed in the AL register.
ZF = 1 if no character is available

If a character is available, the zero flag (ZF) is cleared (i.e., ZF = 0) and the character is returned in the AL register. If no character is available, this function does not wait for a character to be typed. Instead, control is returned immediately to the program and the zero flag is set (i.e., ZF = 1). The input character is not echoed. No `ctrl-break` check is done by this function.

Sub-function — 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 12.5.3).

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.

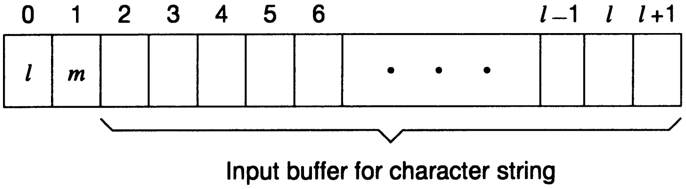
Function 0AH — Buffered keyboard input

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 keys 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.

12.5.3 Extended Keyboard Keys

The IBM PC keyboard has several keys that are not the ASCII characters. 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 12.1 lists the keyboard scan codes for the extended keys. In contrast, when an ASCII key is pressed, the first byte in the keyboard buffer (which is 30 bytes long to store 15 type-ahead keys with two bytes for each key) is ASCII code for the key, and the second byte is the scan code of the key.

To read a character from the keyboard using DOS functions, extended keys require two function calls, as shown in the following procedure.

```
Read the next character code into AL using function 08H
if (AL ≠ 0)
then
    AL = ASCII code (ASCII character)
else        {extended key}
    read the scan code of the extended key into AL using
        function 07H
    AL = scan code (extended key character)
end if
```

Example 12.1

In this example, we look at the GetStr procedure that we used to read a string from the keyboard. The GetStr is a macro (see the io.mac file listing) that can receive up to two parameters: a pointer to a buffer to store the input string, and an optional buffer length value. The macro, after checking the parameters, places the buffer pointer in AX and the buffer length in CX and calls the procedure proc_GetStr. This procedure actually reads a string from the keyboard using the buffered keyboard input function 0AH. The pseudocode for the procedure is as follows.

```
proc_GetStr ()
    save registers used in the procedure
    if (CX < 2)
    then
        CX := 2
    else
        if (CX > 81)
        then
            CX := 81
        end if
    end if
    use function 0AH to read input string into
        temporary buffer str_buffer
    copy input string from str_buffer to
        user buffer and append NULL
        restore registers
        return
    end proc_GetStr
```

The DOScall macro is defined as follows:

```
DOScall   MACRO   function_number
          mov     AH,function_number
          int     21H
          ENDM
```

Program 12.38 Procedure to read a string from the keyboard

```
;------------------------------------------------------------
; Get string (of maximum length 80) from keyboard.
;     AX <-- pointer to a buffer to store the input string
;     CX <-- buffer size = string length + 1 for NULL
; If CX < 2,  CX := 2 is used to read at least one character.
; If CX > 81, CX := 81 is used to read at most 80 characters.
;------------------------------------------------------------
proc_GetStr  PROC
      push       DX      ; save registers
      push       SI
      push       DI
      push       ES
      mov        DX,DS   ; set up ES to point to DS
      mov        ES,DX   ;  for string instruction use
      mov        DI,AX   ; DI := buffer pointer
      ; check CX bounds
      cmp        CX,2
      jl         set_CX_2
      cmp        CX,81
      jle        read_str
      mov        CX,81
      jmp        SHORT read_str
set_CX_2:
      mov        CX,2
read_str:
      ; use temporary buffer str_buffer to read the string
      ;  in using function 0AH of int 21H
      mov        DX,OFFSET str_buffer
      mov        SI,DX
      mov        [SI],CL  ; first byte = # of chars. to read
      DOScall    0AH
      inc        SI       ; second byte = # of chars. read
      mov        CL,[SI]  ; CX := # of bytes to copy
      inc        SI       ; SI = input string first char.
      cld                 ; forward direction for copy
      rep        movsb
      mov        BYTE PTR [DI],0    ; append NULL character
      pop        ES       ; restore registers
      pop        DI
      pop        SI
      pop        DX
      ret
proc_GetStr  ENDP
```

12.5.4 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 an interrupt of type 16H. One difference between DOS and BIOS functions is that if you use DOS services, the keyboard input can be redirected.

Function 00H — Read a character from the keyboard

Input:	AH	=	00H
Returns:	if AL	$\neq$	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

You can see from the ASCII table in Appendix E that 00H represents NULL in ASCII. Note that returning the NULL key ASCII code (00H) is interpreted as reading an extended key. Then how will you recognize the NULL key? This is a special case and the only ASCII key that is returned as an extended key character. Thus, if AL = 0 and AH = 3 (the scan code for the `@` key), then the contents of AL should be treated as the ASCII code for the NULL key.

Here is a simple routine to read a character from the keyboard, which is a modified version of the routine given on page 452.

```
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. In this case, the ASCII and scan codes are placed in the AL and AH registers as in function 00H. The codes, however, are not removed from the keyboard buffer.

This function can be used to take a peek at the next character without actually removing it from the buffer. It provides similar functionality as the DOS function 0BH (see page 451). 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 the next character without actually reading it from the buffer.

Function 02H — Check keyboard status

Table 12.2 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

Input:	AH =	02H
Returns:	AL =	status of the shift and toggle keys

The bit assignment is shown in Table 12.2. 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).

Example 12.2

In this example, we write a program that reads a character string from the keyboard and displays the input string along with its length. The string input is terminated either by pressing both 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 str_len and
        string pointer from the stack
    repeat
        read keyboard status (use int 16H function 2)
        if (both shift keys depressed)
```

```
                then
                    goto end_read
                else
                    read keyboard key (use int 16H function 0)
                    copy the character into the string buffer
                    increment buffer pointer
                    display the character on the screen
                end if
            (string length = str_len)
        end_read:
            append NULL character to string input
            find and return the string length
            return
        end read_string
```

Unfortunately, this procedure will not work properly. In most cases, the only way to terminate the string input is by actually entering 80 characters. Pressing both shift keys will not terminate the string input unless a key is entered while holding both shift keys down. Why? The problem with the above code is that the repeat loop briefly checks the keyboard status (takes only a few microseconds). It then waits for you to type a key. When you enter a key, it reads the ASCII code of the key and initiates another repeat loop iteration. Thus, every time you enter a key, the program checks the status of the two shift keys within a few microseconds after a key has been typed. Therefore, read_string will almost never detect the condition that both shift keys are depressed (with the exception noted before).

To correct this problem, we have to modify the procedure as follows:

```
read_string()
    get maximum string length str_len and
        string pointer from the stack
read_loop:
    repeat
        read keyboard status (use int 16H function 2)
        if (both shift keys depressed)
        then
            goto end_read
        else
            check keyboard buffer status (use int 16H function 1)
            if (a key is available)
            then
```

```
                read keyboard key (use int 16H function 0)
                copy the character into the string buffer
                increment buffer pointer
                display the character on screen
            end if
        end if
    (string length = str_len)
end_read:
    append NULL character to string input
    find and return the string length
    return
end read_string
```

With the modification, the procedure's `repeat` loop spends most of the time performing the following two actions:

1. read keyboard status (using `int 16H` function 2)
2. check if a key has been pressed (using `int 16H` function 1)

Since function 1 does not wait for a key to be entered, the procedure properly detects the string termination condition (i.e., depression of both shift keys simultaneously).

Program 12.39 `funnystr.asm` demonstrates the use of BIOS functions to read a string from the keyboard

```
COMMENT |       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
.MODEL SMALL
.STACK   100H
.DATA
string      DB  STR_LENGTH DUP (?)
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

```

```
.CODE
INCLUDE  io.mac
main    PROC
        .STARTUP
        PutStr  prompt_msg
        mov     AX,STR_LENGTH-1
        push    AX              ; push max. string length
        mov     AX,OFFSET 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
main    ENDP
;-----------------------------------------------------------
; String read procedure using BIOS int 16H. Receives string
; pointer and the length via the stack. Length of the string
; is returned in AX.
;-----------------------------------------------------------
read_string     PROC
        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     BYTE PTR[BX],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
read_string        ENDP
           END     main
```

12.6 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 three DOS functions to display text on the screen. Two of these functions display a single character, while the third function displays a string of characters 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 449, provides both keyboard input and display output services. A character code other than FFH in DL will cause 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 should 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. As a consequence, this function cannot be used to display a string containing $.

Example 12.3

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 12.40.

Program 12.40 Procedure to send a newline (CR and LF) to the screen

```
;------------------------------------------------------------
; Sends CR and LF to the screen. Uses display function 2
;------------------------------------------------------------
proc_nwln  PROC
           push     DX
           mov      DL,0DH        ; carraige return
           DOScall  2
           mov      DL,0AH        ; line feed
           DOScall  2
           pop    DX
           ret
proc_nwln  ENDP
```

12.7 Printer Support

DOS supports three parallel printers (LPT1, LPT2, and LPT3). It also supports four serial devices (COM1, COM2, COM3, and COM4) which can include serial printers. Programming a parallel printer is easier than programming a serial printer. Serial printers require specification of the serial communication characteristics such as the baud rate, parity, stop bits, etc.

Parallel printers require that the data be transmitted in parallel (i.e., 8 bits wide). In contrast, serial printers receive data in bit-serial fashion much like a modem. Parallel printers should be located close to the source of data (typically less than 6 feet). Serial printers, on the other hand, can be located farther away (up to about 50 feet).

In this section, our discussion is limited to parallel printers. Both DOS and BIOS support parallel printers. We start our discussion with the DOS functions.

12.7.1 DOS Printer Services

DOS provides two functions to print a character on the standard printer (LPT1).

Function 05H — Print a character

Inputs: AH = 05H
DL = ASCII code of the character to be printed
Returns: Nothing

The character to be printed should be placed in DL before invoking the DOS function 05H. This function waits until the printer accepts the character. You can use `ctrl-break` to terminate the waiting. Most printers maintain an internal buffer (usually with a capacity to store 80 characters) to store the characters received for printing. Actual printing is done when either the buffer is full or when a carriage return is received.

The following macro `print_char` can be used to print a character on the default printer.

```
print_char  MACRO   char
            push    AX          ;; save registers used
            push    DX          ;;  in the macro
            mov     AH,05H      ;; select print function
            mov     DL,char     ;; char. to be printed
            int     21H
            pop     DX          ;; restore registers
            pop     AX
            ENDM
```

12.7.2 BIOS Printer Support

BIOS printer services are provided by interrupt 17H. There are three functions. The value in the AH register is used to select a function.

Function 00H — Print a character

Inputs: AH = 0
AL = ASCII code of the character to be printed
DX = printer number
(0 = LPT1, 1 = LPT2, 2 = LPT3)
Returns: AH = printer status byte as shown below.

Bit	Meaning
0	Printer time-out if the bit is 1
1	Reserved
2	Reserved
3	Printer I/O error if the bit is 1
4	Printer select 0 — Printer offline 1 — Printer online
5	Printer out of paper if the bit is 1
6	Acknowledge if the bit is 1
7	Printer ready 0 — Printer busy 1 — Printer not busy

Bit 0 is used to indicate that BIOS received no response from the printer adaptor. For successful operation, bit 3 should be 0. Bit 7 indicates whether the printer is ready to receive a character (when not busy) or not (when busy).

Function 01H — Initialize printer

Inputs: AH = 1
DX = printer number
(0 = LPT1, 1 = LPT2, 2 = LPT3)
Returns: AH = printer status byte as shown in function 00H.

This function can be used to reset the selected printer to the power-up status.

Function 02H — Get printer status

Inputs: AH = 2
DX = printer number
(0 = LPT1, 1 = LPT2, 2 = LPT3)
Returns: AH = printer status byte as shown in function 00H.

This function can be used to read the status of the selected printer. For example, before sending a character, the printer status can be read to ensure that the printer is online, not out of paper, and ready to receive a character.

12.8 Exceptions

Exceptions are classified into *faults*, *traps*, and *aborts* depending on the way they are reported and whether or not 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. Further, the instruction pointer is adjusted to point to the divide instruction so that, after returning from the exception handler, the divide instruction is restarted. 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 a hard disk and resume program execution starting with the instruction that caused the exception. In this example, it clearly makes sense to restart the instruction that caused the exception.

Traps, on the other hand, are reported at the instruction boundary immediately following the instruction during which the exception was detected. For instance, overflow exception (interrupt 4) is a trap. Therefore, no instruction restart is supported. User-defined interrupts are also examples of traps.

Aborts are exceptions that report severe errors. Examples include hardware errors and inconsistent values in system tables.

There are several interrupts predefined by Pentium. These are called *dedicated* interrupts. These include the first five interrupts:

Interrupt type	Purpose
0	Divide error
1	Single-step
2	Nonmaskable interrupt (NMI)
3	Breakpoint
4	Overflow

The NMI is clearly a hardware interrupt that will be discussed in Section 12.9. A brief description of the remaining four interrupts is given here.

Divide Error Interrupt

The CPU generates an interrupt of type 0 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 ISR displays a *divide overflow* message and terminates the program.

Single-Step Interrupt

Single stepping is a useful debugging tool to observe the behavior of a program instruction by instruction. To start single stepping, the trap flag (TF) bit in the flags register should be set (i.e., TF = 1). When TF is set, the CPU automatically generates a type 1 interrupt after executing each instruction. Some exceptions do exist, but we will not bother about them here.

The ISR for a type 1 interrupt can be used to display relevant information about the state of the program. For example, the contents of all registers could be displayed. Shortly, we will present an example program that initiates and stops single stepping (see the example on page 466).

To end single stepping, TF should be cleared. The CPU, however, does not have any instructions to manipulate the TF directly. Instead, we have to resort to an indirect means. This is illustrated in the example on page 466.

Breakpoint Interrupt

If you have used a debugger (which you should have by now) such as the Turbo Debugger, you already know the usefulness of inserting breakpoints while debugging a program. A type 3 interrupt is dedicated to the breakpoint interrupt. 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

A 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 a part of the instruction. Unlike `int 4`, which unconditionally generates a type 4 interrupt, `into` generates a type 4 interrupt only if the overflow flag is set. We do not normally use `into`, as the overflow condition is usually detected and processed by using the conditional jump instructions `jo` and `jno`.

Example 12.4

As an example of an exception, we write an ISR to single step a piece of code (let us call it *single-step code*). During single stepping, we display the contents of the AX and BX registers after the execution of each instruction 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 CPU in the single-step mode, we have to set the TF. Since there are no instructions to manipulate TF directly, we have to use an indirect means: first use `pushf` to push flags onto the stack; then manipulate the TF bit; and finally, use `popf` to restore the modified flags word from the stack to the flags register. The code on lines 42–46 of Program 12.41 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
```

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 27–30: 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 33–39: The vector of our ISR is installed by using function 25H of `int 21h`.

Lines 62–68: The original `int 1` vector is restored using function 25h of `int 21h`.

A Note

It is not necessary to restore the old vector before we exit the program. DOS does this for us. DOS stores the old values in the PSP area and restores them as a part of a function 4CH call. Thus, it is not necessary to read and store the old interrupt vector value in our program. However, DOS does not restore interrupt vectors for all interrupts. As an example, it does not restore the original vector for `int 09H`. Thus, it is good practice to save and restore interrupt vectors within your program. We will follow this practice for all the examples.

Program 12.41 An example to illustrate the installation of a user-defined ISR

```
 1: TITLE   Single-step program      STEPINTR.ASM
 2: COMMENT |
 3:         Objective: To demonstrate how ISRs can be defined
 4:                    and installed.
 5:             Input: None
 6:            Output: Displays AX and BX values for
 7: |                  the single-step code
 8:
 9: .MODEL SMALL
10: .STACK 100H
11: .DATA
12: old_offset  DW  ?    ; for old ISR offset
13: old_seg     DW  ?    ;   and segment values
14: start_msg   DB  'Starts single stepping process.',0
15: AXequ       DB  'AX = ',0
16: BXequ       DB  ' BX = ',0
17:
18: .CODE
19: INCLUDE io.mac
20:
21: main    PROC
22:         .STARTUP
23:         PutStr  start_msg
24:         nwln
25:
26:         ; get current interrupt vector for int 1H
27:         mov     AX,3501H       ; AH := 35H and AL := 01H
28:         int     21H            ; returns the offset in BX
```

```
        mov     old_offset,BX  ;   and the segment in ES
        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,OFFSET 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
main    ENDP
;----------------------------------------------------------
```

```
;Single-step interrupt service routine replaces int 01H.
;-----------------------------------------------------------
sstep_ISR  PROC
        sti                  ; enable interrupt
        PutStr  AXequ        ; display AX contents
        PutInt  AX
        PutStr  BXequ        ; display BX contents
        PutInt  BX
        nwln
        iret
sstep_ISR  ENDP
        END   main
```

12.9 Hardware Interrupts

We have seen how interrupts can be caused by the software instruction `int`. Since these instructions are placed in a program, such software interrupts are called *synchronous* events. Hardware interrupts, on the other hand, are of hardware origin and *asynchronous* in nature. These interrupts are typically used by peripheral or I/O devices such as a keyboard to alert the CPU that they require its attention.

Hardware interrupts can be further divided into either *maskable* or *nonmaskable* interrupts (see Figure 12.1). 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 its type (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 CPU initiates an interrupt acknowledge sequence. As a part of this sequence, the CPU sends out an interrupt acknowledge (INTA) signal, and the interrupting device is expected to place the interrupt type number on the data bus. This number is used to identify the interrupt type.

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, floppy disk, and printer all generate interrupts when they require the attention of the CPU.

When more than one device interrupts, we have to have a mechanism to prioritize these interrupts (if they come simultaneously) and forward only one interrupt request at a time to the CPU 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 will defer our discussion of this chip until Section 12.11.1.

12.10 Direct Control of I/O Devices

Figure 12.2 on page 442 shows three ways of interacting with I/O devices by an application program. 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 either from DOS or from BIOS, or when we want a nonstandard access, we have to access these devices directly—the third method shown in Figure 12.2.

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 Chapter 2 for a related discussion.

12.10.1 Accessing I/O Ports

To facilitate access to the I/O ports, Pentium 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

Pentium provides two register I/O instructions: `in` and `out`. The `in` instruction is used to read data from an I/O port, and the `out` instruction to write data to an I/O port. A port address can be any value in the range 0 to FFFFH. The first 256 ports (i.e., ports with address in the range 0 to FFH) are directly addressable—i.e., address is given as a part of the instruction—by `in` and `out` instructions.

Both `in`/`out` instructions can be used to read/write 8-, 16-, or 32-bit data. Each instruction can take one of two forms, depending on whether a port is directly addressable or not. The general formats of the `in` instruction are:

```
in     accumulator,port8 — direct addressing format
in     accumulator,DX    — indirect addressing format
```

The first form uses the direct addressing mode and can only be used to access the first 256 ports. In this case, the I/O port address, which is in the range 0 to FFH, is given by the `port8` operand. In the other form, the I/O port address is given indirectly via the DX register. The contents of the DX register are treated as the port address.

In either form, the first operand `accumulator` must be AL, AX, or EAX. This choice determines whether a byte, word, or doubleword is read from the specified port.

The corresponding forms for the `out` instruction are

```
out    port8,accumulator — direct addressing format
out    DX,accumulator    — indirect addressing format
```

Notice the placement of the port address. In the `in` instruction, it is the source operand and in the `out` instruction, it is the destination operand signifying the direction of data movement.

Block I/O Instructions

Pentium supports two block I/O instructions: `ins` and `outs`. These instructions can be used to move blocks of data between I/O ports and memory. These I/O instructions are, in some sense, similar to the string instructions discussed in Chapter 9. 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 direct addressing format for the I/O port specification.

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.

12.11 Peripheral Support Chips

Recall from Chapter 2 that I/O devices are not interfaced directly to the CPU. Rather, each device has a device or peripheral controller that acts as an intermediary between the device and the CPU, as shown in Figure 12.4

In this section, we start our discussion by explaining how multiple devices can interrupt the CPU using the Intel 8259 programmable interrupt controller chip. Then, we proceed to describe the Intel 8255 programmable peripheral interface chip.

12.11.1 8259 Programmable Interrupt Controller

To accommodate more than one interrupting device in the system, your PC uses the Intel 8259 programmable interrupt controller (PIC) chip. The 8259

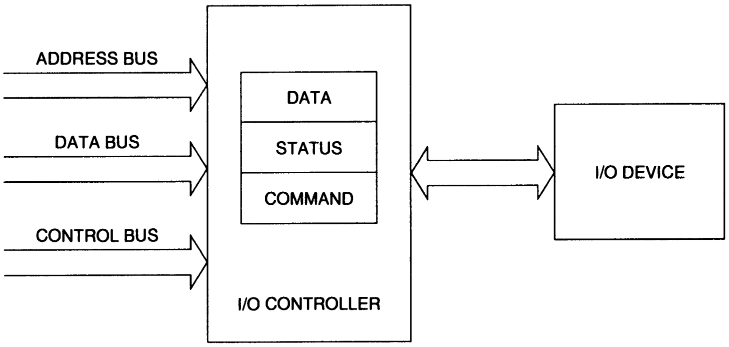


Figure 12.4 Input/output device interface to the system.

PIC can service interrupts from up to eight hardware devices. These interrupts are received on lines IRQ0 through IRQ7, as shown in Figure 12.5.

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 IRQ0–IRQ7. 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.

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 seven numbers to the remaining seven IRQ lines in increasing order, with IRQ7 generating an interrupt of type 0FH.

All communication between the CPU and the 8259 occurs via the data bus. The 8259 PIC is an 8-bit device requiring two ports for ICR and IMR. These are mapped to the I/O address space, as shown in Table 12.3. Table 12.4 shows the mapping of IRQ input of the 8259 to various devices in the system.

Note that the CPU 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 the IMR. Each bit in the IMR enables (if the bit is 0) or disables (if the bit is 1) its associated interrupt. Bit 0 is associated with IRQ0, bit 1 with IRQ1, and so on. For example, we can use the code

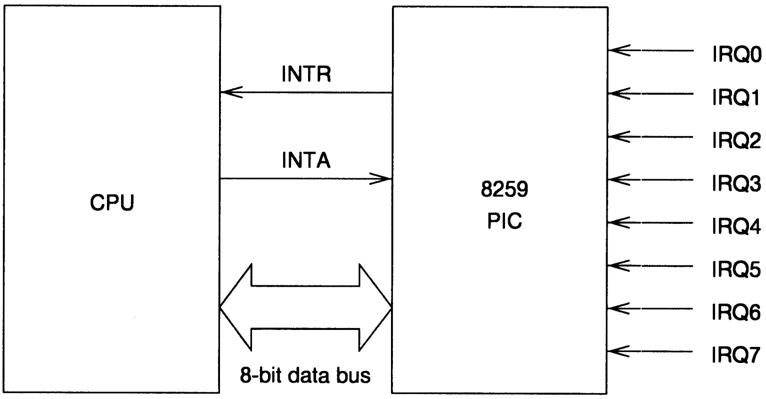


Figure 12.5 Intel 8259 programmable interrupt controller.

Table 12.3 8259 port address mapping

8259 register	Port address
ICR	20H
IMR	21H

Table 12.4 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

Table 12.5 8255 port address mapping

8255 register	port address
PA (input port)	60H
PB (output port)	61H
PC (input port)	62H
Command register	63H

```
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 CPU, as it has a higher priority than the keyboard interrupt. Once the timer ISR is completed, the 8259 forwards the keyboard interrupt to the CPU for processing. To facilitate this, the 8259 should know when an ISR is completed. The end of an ISR execution is signaled to the 8259 by writing 20H into the ICR. Thus the code fragment

```
mov     AL,20H
out     20H,AL
```

can be used to indicate end-of-interrupt (EOI) to the 8259 PIC. This code fragment appears before the `iret` instruction of an ISR.

12.11.2 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 I/O space as shown in Table 12.5. The BIOS configures the three ports of the 8255 a shown in Table 12.5. 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 12.6.

The keyboard interface is provided by port PA, and PB7. The hardware within the keyboard scans the keys to check the state of the keys (i.e., depressed or released). The keyboard sends an interrupt to 8259 (on the IRQ1 line)

Table 12.6 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		
	PB0	8253 timer 2 gate to speaker (1 enables channel 2 timer)
	PB1	speaker data (1 enables timer clock to go to the speaker driver)
	PB2	selects source for port PC bits 0–3
	PB3	cassette motor
	PB4	Enable RAM parity check
	PB5	Disable I/O channel check
	PB7	selects source for PA input 0 — keyboard scan code 1 — configuration switch 1 Also, 1 is used as keyboard acknowledge

whenever there is a change in a key state. Recall that IRQ1 generates a type 9 interrupt. 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 read by the processor. This acknowledgment can be signaled 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 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 replace the BIOS `int 9` ISR to illustrate the keyboard interface.

12.12 A Hardware Interrupt Example

In this section, we illustrate how a hardware interrupt routine can be written. As an example, we write a type 9 interrupt routine to replace the BIOS supplied `int 09` routine.

Example 12.5

Our objective is to write a replacement ISR for `int 09H`. Recall that a type 9 interrupt is generated via the IRQ1 line of the 8259 PIC by the keyboard every time a key is depressed or released.

The logic of the main procedure can be described as follows:

```
main()
    save the current int 9 vector
    install our keyboard ISR
    display "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
     (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. After reading the key scan code in Program 12.42 (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 at lines 154 and 155 will send an end-of-interrupt (EOI) 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 12.42 A keyboard ISR to replace BIOS int 09H keyboard ISR

```
1:  TITLE   Keyboard interrupt service program   KEYBOARD.ASM
2:  COMMENT |
3:          Objective: To demonstrate how the keyboard works.
4:              Input: Key strokes from the keyboard. Only left
5:                     and right shift keys are recognized.
6:                     ESC key restores the original keyboard ISR
7:                     and terminates the program.
8:  |          Output: Displays the key on the screen.
```

```

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

.MODEL SMALL
.STACK 100H
.DATA
install_msg    DB  ’New keyboard ISR installed.’,0
keyboard_data  DB  -1   ; keyboard buffer
keyboard_flag  DB  0    ; keyboard shift status
old_offset     DW  ?    ; storage for old int 09H vector
old_segment    DW  ?
; lowercase scan code to ASCII conversion table.
; ASCII code 0 is used for scan codes we are not interested.
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
.CODE
INCLUDE io.mac

main    PROC
        .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,OFFSET kbrd_ISR  ; ISR offset in DX
        mov     AX,2509H     ; AH := 25H and AL := 09H
        int     21H
        pop     DS           ; restore DS

repeat:
        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     repeat
newline:
        nwln
        jmp     repeat
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
main    ENDP
;-----------------------------------------------------------
;This procedure waits until a valid key is entered at the
; keyboard. The ASCII value of the key is returned in AL.
;-----------------------------------------------------------
read_kb_key  PROC
        cmp     keyboard_data,-1  ; -1 is an invalid entry
        je      read_kb_key
        mov     AL,keyboard_data
        mov     keyboard_data,-1
        ret
read_kb_key  ENDP
;-----------------------------------------------------------
;This keyboard ISR replaces the original int 09H ISR.
;-----------------------------------------------------------
kbrd_ISR  PROC
        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,OFFSET lcase_table ; shift OFF, use lowercase
        jmp     SHORT get_ASCII       ;  conversion table
shift_key_on:
        mov     BX,OFFSET 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      keyboard_flag,1     ; shift bit (i.e., LSB) := 1
        jmp     SHORT EOI_to_8259
shift_off:
        and     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
kbrd_ISR  ENDP
        END   main
```

12.13 Performance: Polling Versus Interrupts

Interrupts provide a convenient mechanism to draw the attention of the CPU to process an event. We have seen how hardware interrupts can be used to interact with I/O devices. We have shown how a keyboard can use interrupts for processing its requests. This type of I/O is called *interrupt-driven I/O*.

An alternative to interrupt-driven I/O is the *programmed I/O*. In programmed I/O, the program repeatedly checks the status of an I/O device (actually, the status register of the associated I/O controller) until the desired condition is indicated. This process is called *polling*. Thus, programmed I/O is typically characterized by loops. These loops are called *polling loops*. Polling is used for other purposes as well. For instance, in Program 12.39 on page 458, we used polling to check the keyboard status using `int 16H`.

Interrupt-driven processing is efficient because it eliminates the overhead associated with polling loops. Furthermore, polling can only be done at places in the program (by explicitly writing code to do the polling) where you can anticipate the occurrence of a particular condition. Thus, with polling, we cannot handle unexpected events.

To illustrate the amount of polling overhead, we use our bubble sort example from Chapter 1. As written, the program cannot be terminated even by `ctrl-`

break. This is because a ctrl-break check is done whenever the keyboard or display is accessed (either to read a keyboard character or to display a character on the screen). However, the bubble sort procedure does not access these two devices while performing the sort. The frequency of ctrl-break can be increased by adding

break = on

to the configuration file (it can also be typed at the terminal). This increases the ctrl-break check frequency (for example, by also checking whenever a disk is accessed). This additional check is of no use to us. Therefore, once the sort operation begins, the sort procedure runs to completion (unless the system is reset).

To cause programmed termination of the bubble sort procedure, we will modify the assembly language routine. We will include code

```
next_pass:
        mov     AH,0BH          ; to process ctrl-break
        int     21H
```

after each iteration to read the keyboard buffer status using the DOS function 0BH. Note that this DOS function performs the ctrl-break check.

Figure 12.6 shows the performance of the original version and the modified version. Both versions are run to completion without premature termination. The data shows that the polling overhead decreases with the array size to be sorted. For example, the polling overhead, as a percentage of the original sort time, decreases from about 38 percent to 17 percent when the array size is increased from 2000 elements to 8000 elements, as shown in the following table.

Array size	Overhead
2000	38%
5000	22%
8000	17%

The decrease in polling overhead is to be expected, as the time required for polling remains constant for each pass, while the execution time of each pass increases with the array size.

The polling overhead is overwhelming if we increase the polling frequency. For example, instead of polling once during each pass, if we modify the code to do polling each time a swap is performed, the execution time for a 1000-element array increases from less than 0.5 seconds to more than 50 seconds!

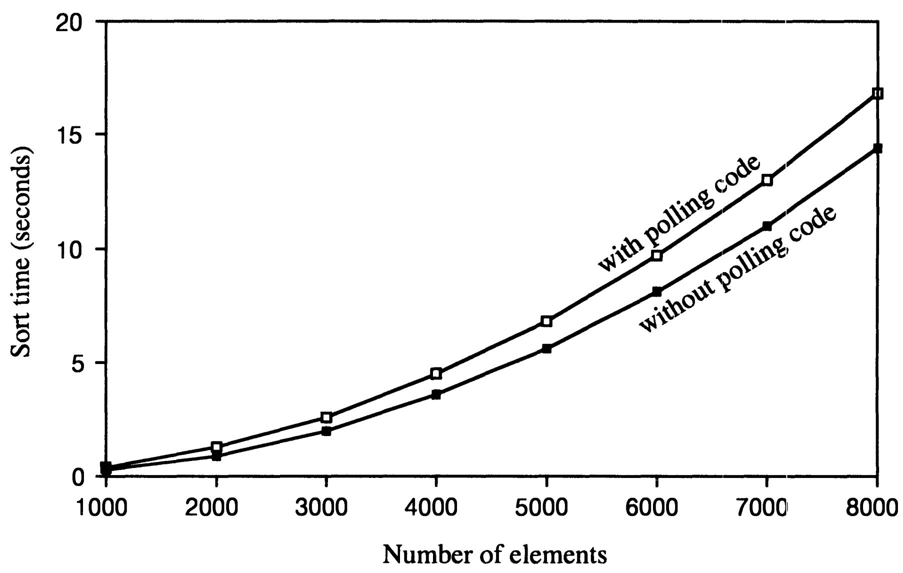


Figure 12.6 Polling overhead for the bubble sort example of Chapter 1.

12.14 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 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 CPU to service their requests.

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. Hardware interrupts can be masked or disabled by manipulating the interrupt flag using `sti` and `cli` instructions. Masking of individual external interrupts can be done by manipulating the IMR of the 8259 PIC.

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.

We briefly introduced polling as an alternative to interrupt-driven I/O. Polling introduces overhead due to pooling loops. The last section examined the impact of this overhead on the bubble sort example of Chapter 1.

12.15 Exercises

12–1 What is the difference between a procedure and an interrupt service routine?

12–2 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.

12–3 How would you categorize the interrupts generated by the keyboard?

12–4 Explain how one can disable all maskable hardware interrupts efficiently. Efficiency here refers to both time- and space-efficiency of the code.

12–5 Describe another way to disable all maskable hardware interrupts.

12–6 Write a piece of code to disable all maskable hardware interrupts except the timer and keyboard interrupts. Refer to the interrupt table on page 474.

12–7 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 the two instructions?

12–8 Suppose that the CPU is currently executing the keyboard interrupt service routine, which is shown below:

```
keyboard_ISR    PROC
        sti
         .
         .
        ISR body
         .
         .
        iret
keyboard_ISR    ENDP
```

Suppose that, while in the middle of executing the keyboard ISR, a timer interrupt has occurred. Describe the activities of the CPU until it completes processing the keyboard interrupt service routine.

12–9 What happens in the scenario described in the last question if the `sti` instruction is deleted from the keyboard ISR?

12–10 Discuss the advantages and disadvantages of the three ways an application program can interact with I/O devices (see Figure 12.2).

12–11 Describe the actions taken (until the beginning of the execution of ISR) by the CPU in response to an interrupt `int 10H`. You can assume real mode of operation.

12–12 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?

12–13 What is the difference between the DOS keyboard function 0BH and the BIOS keyboard function 01H?

12–14 Describe how extended keyboard keys are handled.

12–15 Discuss the tradeoffs associated with polling and interrupts.

12.16 Progamming Exercises

12–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 maximum possible value. 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.

12–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.

12–P3 Convert `toupper.asm` given in Chapter 3 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.

12–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)

12–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)

Chapter 13

High-Level Language Interface

Objectives

- To review motivation for writing mixed-mode programs
- To discuss the principles of mixed-mode programming
- To describe how assembly language procedures are called from C
- To illustrate how C functions are called from assembly language procedures
- To explain how inline assembly language code is written

Thus far we have written stand-alone assembly programs. Except in Chapter 1, our discussion has focused on the mechanics of Pentium assembly language programming. This last chapter considers mixed-mode programming. In this mode, part of a program is written in a high-level language and part in assembly language. We use C and Pentium assembly languages to illustrate how such mixed-mode programs are written. The motivation for mixed-mode programming is discussed in Section 13.1. Section 13.2 gives an overview of mixed-mode programming. Mixed-mode programming can be done either by inline assembly code or by separate assembly modules. The inline assembly method is discussed in Section 13.6. Other sections focus on the separate assembly module method.

Section 13.3 gives a detailed discussion of the mechanics involved in calling assembly language procedures from a C program. This section presents details about parameter passing, returning values to C functions, and so on.

S. P. Dandamudi, *Introduction to Assembly Language Programming*

Section 13.4 details how a C function can be called from an assembly language procedure. Section 13.5 discusses some simplifications that we can use in interfacing assembly programs to C. The last section summarizes the chapter.

13.1 Why Program in Mixed-Mode?

Mixed-mode programming refers to writing parts of a program in different languages. In this chapter we focus on programming in the high-level C language and assembly language. Thus, in our case, part of a program is written in C and the other part in Pentium assembly language. We use the Borland C++ compiler and Turbo Assembler to explain the principles involved in mixed-mode programming. This discussion can be easily extended to a different set of languages—for example, mixed-mode programming involving Pascal and assembly language.

In Chapter 1 we discussed several reasons why one would want to program in assembly language. We identified the following three main reasons:

- Access to hardware
- Time-efficiency
- Space-efficiency

Access to hardware refers to having direct control over the input/output devices and other system hardware. High-level languages, on purpose, do not provide mechanisms to directly access system hardware. As we have seen in Chapter 12, we can use software interrupts to access various I/O devices. For example, `int 21H` software interrupt provides a variety of services to access devices like a keyboard, printer, display screen, disk, and so on. In addition, we can use `in` and `out` instructions to write our own assembly language routines to access I/O devices. These features of assembly language are very important if one is involved in writing systems software.

Time-efficiency refers to the fact that a carefully written assembly language program tends to execute faster than an equivalent program written in a high-level language. We demonstrated this advantage of programming in assembly language in Chapter 1 by using the bubble sort example.

Space-efficiency refers to the space requirements of the executable code. A more efficient program requires less space to do the same task. In general, a cleverly crafted assembly language program tends to take less memory than the equivalent code produced by a compiler. We should note, however, that space efficiency is not critical in most applications. It is important only in handheld devices and other memory-constrained systems.

While it is possible to write a program entirely in assembly language, there are several disadvantages in doing so. These include:

- Low productivity
- High maintenance cost
- Lack of portability

Low productivity is due to the fact that assembly language is a low-level language. That is, in most cases, each assembly language instruction accomplishes only a fraction of the task typically done by a high-level language instruction. As a result, a single high-level language instruction may require several assembly language instructions. We discussed this aspect in Chapter 1 by means of examples. It has been observed that programmers tend to produce the same number of lines of debugged and tested source code per unit time irrespective of the level of the language used. As the assembly language requires more lines of source code, programmer productivity tends to be low.

Programs written in assembly language are also difficult to maintain. This is also a direct consequence of assembly language being a low-level language.

High-level programs are portable in the sense that they can be compiled to a target architecture without making any changes (or sometimes with slight modifications) to the source code. Assembly language programs are not portable and are suitable to run only on the target architecture.

As a result of these pros and cons, some programs are written in mixed-mode using both high-level and low-level languages. System software often requires mixed-mode programming. In such programs, it is possible for a high-level procedure to call a low-level procedure and vice versa. The remainder of the chapter discusses how mixed-mode programming is done in C and assembly languages. Our goal is to illustrate only the principles involved. Once these principles are understood, the discussion can be generalized to any type of mixed-mode programming.

13.2 Overview

There are two ways of writing mixed-mode C and assembly programs—inline assembly code, or separate assembly modules. In the inline assembly method, the C program module can contain assembly language instructions. The Borland C++ compiler allows embedding assembly language instructions within a C program by prefixing them with **asm** to let the compiler know that it is an assembly language instruction. This method is useful if you have only a small amount of assembly code to be embedded. Otherwise, separate assembly

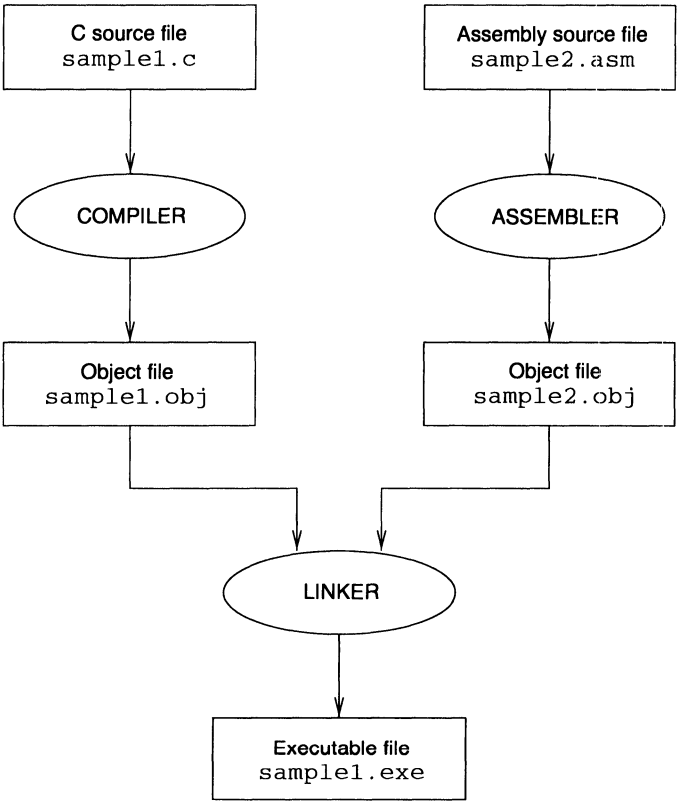


Figure 13.1 Steps involved in compiling mixed-mode programs.

modules are preferred. Section 13.6 discusses how the inline assembly method works with an example. Until then, we focus on separate assembly modules.

When separate modules are used for C and assembly languages, each module can be translated into the corresponding object (.obj) file. To do this translation, we use a C compiler for the C modules and an assembler for the assembly modules, as shown in Figure 13.1. Then the linker can be used to produce the executable (.exe) file from these object files.

Suppose our mixed-mode program consists of two modules—one C module (file sample1.c), and one assembly module (file sample2.asm). The process involved in producing the executable file is shown in Figure 13.1. We can instruct the Borland C++ compiler to initiate this cycle with

```
bcc sample1.c sample2.asm
```

This command instructs the Borland C++ compiler to first compile `sample1.c` to `sample1.obj` and then invoke the Turbo Assembler TASM to assemble `sample2.asm` to `sample2.obj`. The linker TLINK is finally invoked to link `sample1.obj` and `sample2.obj` to produce `sample1.exe`.

13.3 Calling Assembly Procedures from C

Let us now discuss how we can call an assembly language procedure from a C program. The first thing we have to know is what communication medium is used between the C and assembly language procedures. We need to figure this out, as the two procedures may exchange parameters and results. You are right if you guessed it to be the stack.

Given that the stack is used for communication purposes, we still need to know how the C function places the parameters on the stack, and where it expects the assembly language procedure to return the result. In addition, we should also know which registers we can use freely without worrying about preserving their values. Next we discuss these issues in detail.

13.3.1 Parameter Passing

There are two ways in which arguments (i.e., parameter values) are pushed onto the stack: from left to right or from right to left. Most high-level languages such as Basic, Fortran, and Pascal push the arguments from left to right. These are called *left-pusher* languages. C, on the other hand, pushes arguments from right to left. Thus, C is a *right-pusher* language. Right-pushing offers one advantage over left-pushing if the language allows procedures with a variable number of arguments. In this case, the number of arguments pushed onto the stack is available just below the return address pointer on the stack, independent of the number of arguments passed. The stack state after executing

```
sum(a,b,c,d)
```

is shown in Figure 13.2. From now on, we consider only right-pushing of arguments as we focus on the C language.

To see how Borland C++ pushes arguments onto the stack, take a look at the following C program (this is a partial listing of Program 13.1).

```
int main(void)
{
    int     x=25, y=70;
    int     value;
    extern  int test(int, int, int);
```

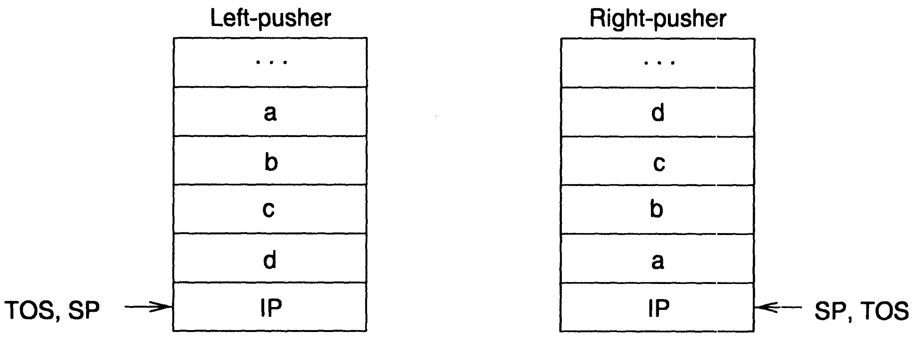


Figure 13.2 Two ways of pushing parameters onto the stack.

```
        value = test (x, y, 5);
        . . .
        . . .
}
```

This program is compiled (use -S option to generate the assembly source code) as:

```
;            int      x=25, y=70;
;
        mov     word ptr [bp-2],25
        mov     word ptr [bp-4],70
;
;            int      value;
;            extern   int test(int, int, int);
;
;            value = test (x, y, 5);
;
        push    5
        push    word ptr [bp-4]
        push    word ptr [bp-2]
        call    near ptr _test
        add     sp,6
        mov     word ptr [bp-6],ax
```

The compiler assigns space for variables x, y, and value on the stack at BP−2, BP−4, and BP−6, respectively. When the test function is called, the arguments are pushed from right to left, starting with the constant 5. Also notice that the stack is cleared of the arguments by the C program after the call by

```
add   sp,6
```

So when we write our assembly procedures, we should not bother clearing the arguments from the stack as we did in our programs in the previous chapters. The rationale for using this convention to clear arguments from the stack is that C allows a variable number of arguments in a function call. On the other hand, parameters are cleared by the called procedure if we are using Pascal instead of C. The Borland C++ compiler allows you to specify the desired parameter passing mechanism (C or Pascal). For example, by using -p option to use Pascal calls, the same program is compiled as

```
;          int     x=25, y=70;
;
       mov     si,25
       mov     word ptr [bp-2],70
;
;          int     value;
;          extern  int test(int, int, int);
;
;          value = test (x, y, 5);
;
       push    si
       push    word ptr [bp-2]
       push    5
       call    near ptr TEST
       mov     di,ax
```

We can clearly see that left-pushing of arguments is used. In addition, the stack is not cleared of the arguments. Thus, in this case, it is the responsibility of the called procedure to clear the stack of the arguments, which is what we have been doing in our assembly programs in the previous chapters.

13.3.2 Returning Values

We can see from the C and Pascal assembly codes given in the last sub-section that the AX register returns the value of the `test` function. In fact, AX is used to return 8- and 16-bit values. To return a 32-bit value, use DX:AX pair with DX holding the upper 16 bits. Table 13.1 shows how various values are returned to the Borland C++ function. This list does not include floats and doubles. These are returned via the 8087 stack. We will not discuss these details here.

Table 13.1 Registers used to return values

Return value type	Register used
unsigned char	AX
char	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX
long	DX:AX
near pointer	AX
far pointer	DX:AX

13.3.3 Preserving Registers

In general, the called assembler procedure can use the registers as needed, except that the following register contents should be preserved:

```
BP, SP, CS, DS, SS
```

In addition, if register variables are enabled, both SI and DI registers should also be preserved. When register variables are enabled, both SI and DI registers are used for variable storage, as shown below:

```
;          int     x=25, y=70;
;
     mov     si,25
     mov     word ptr [bp-2],70
;
;          int     value;
;          extern  int test(int, int, int);
;
;          value = test (x, y, 5);
;
     push    5
     push    word ptr [bp-2]
     push    si
     call    near ptr _test
     add     sp,6
     mov     di,ax
```

Compare this version, with register variables enabled, to the previous version given on page 494. Instead of using the stack, SI and DI are used to map variables x and value, respectively. Since one never knows whether the C code was compiled with or without enabling the register variables, it is good practice to preserve SI and DI registers as well.

13.3.4 Publics and Externals

Mixed-mode programming with separate assembly modules involves at least two program modules—one C module and one assembly module. Thus, we have to declare those functions and procedures that are not defined in the same module as external. Similarly, those procedures that are accessed by another module should be declared as public, as discussed in Chapter 4. Before proceeding further, you may want to refresh your memory by reviewing the material on multimodule programs presented in Chapter 4. Here we mention only those details that are specific to the mixed-mode programming involving C and assembly language.

In C, all external labels should start with an underscore character (_). The C and C++ compilers automatically append the required underscore character to all external functions and variables. For example, when we called the `test` function in our example C program, the corresponding assembly code shown on page 494 used `_test`, as we have declared `test` as an external function. A consequence of this characteristic is that when we write an assembly procedure that is called from C, we have to make sure that we prefix an underscore character to its name. Next we present a few examples to illustrate mixed-mode programming.

13.3.5 Illustrative Examples

We now look at three examples to illustrate the interface between C and assembly programs. We start with a simple example, whose C part has been dissected in the previous subsections. For an additional example, see the bubble sort example discussed in Chapter 1.

Example 13.1 *Our first mixed-mode example*

This example passes three parameters to the assembly language function `test`. The C code is shown in Program 13.43 and the assembly code in Program 13.44. Since the `test` procedure is called from the C program, we have to prefix an underscore character to the procedure name. The function `test` is declared as external in the C program (line 11) and public in the assembly program (line 7). Since C clears the arguments from the stack, the assembly procedure uses a simple `ret` to transfer control back to the C program. Other than these differences, the assembly procedure is similar to several others we have written before.

Program 13.43 An example illustrating assembly calls from C—C code (in file `testex_c.c`)

```
 1: /**********************************************************
 2:  * A simple example to illustrate C and assembly language *
 3:  * interface. The test function is written in assembly    *
 4:  * language (in file testex_a.asm).                       *
 5:  **********************************************************/
 6: #include <stdio.h>
 7: int main(void)
 8: {
 9:         int     x=25, y=70;
10:         int     value;
11:         extern  int test(int, int, int);
12:
13:         value = test (x, y, 5);
14:         printf("result = %d\n", value);
15:         return 0;
16: }
```

Program 13.44 An example illustrating assembly calls from C—Assembly code (in file `testex_a.asm`)

```
 1: ;-----------------------------------------------------------
 2: ; Assembly program for the test function - called from the
 3: ; C program in file testex_c.c
 4: ;-----------------------------------------------------------
 5: .MODEL SMALL
 6: .CODE
 7: PUBLIC _test
 8: _test   PROC
 9:         push    BP
10:         mov     BP,SP
11:         mov     AX,[BP+4] ; get argument1 x
12:         add     AX,[BP+6] ; add argument2 y
13:         sub     AX,[BP+8] ; subtract argument3 from sum
```

```
        pop     BP
        ret               ; stack cleared by C function
_test   ENDP
        END
```

Example 13.2 *An example to show parameter passing by call-by-value as well as call-by-reference*

This example shows how pointer parameters are handled. The C main function requests three integers and passes them to the assembly procedure. The C program is given in Program 13.45. The assembly procedure min_max, shown in Program 13.46, receives the three integer values and two pointers to variables minimum and maximum. It finds the minimum and maximum of the three integer values and returns them to the main C function via the two pointer variables. The minimum value is kept in AX and the maximum in DX. The code given on lines 29–32 in Program 13.46 stores the return values by using the BX register in the indirect addressing mode.

Program 13.45 An example with C program passing pointers to assembly program—C code (in file minmax_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 file minmax_a.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 13.46 An example with C program passing pointers to assembly program—Assembly code (in file minmax_a.asm)

```
;-----------------------------------------------------------
; Assembly program for the min_max function - called from
; the C program in file minmax_c.c. This function finds the
; minimum and maximum of the three integers received by it.
;-----------------------------------------------------------
.MODEL SMALL
.CODE
PUBLIC _min_max
_min_max    PROC
        push    BP
        mov     BP,SP
        ; AX keeps minimum number and DX maximum
        mov     AX,[BP+4]     ; get value 1
        mov     DX,[BP+6]     ; get value 2
        cmp     AX,DX         ; value 1 < value 2?
        jl      skip1         ; if so, do nothing
        xchg    AX,DX         ; else, exchange
skip1:
        mov     CX,[BP+8]     ; get value 3
        cmp     CX,AX         ; value 3 < min in AX?
        jl      new_min
        cmp     CX,DX         ; value 3 < max in DX?
        jl      store_result
        mov     DX,CX
        jmp     store_result
new_min:
        mov     AX,CX
store_result:
        mov     BX,[BP+10]    ; BX := &minimum
        mov     [BX],AX
        mov     BX,[BP+12]    ; BX := &maximum
        mov     [BX],DX
        pop     BP
        ret
_min_max   ENDP
        END
```

Example 13.3 *String processing example*

This example illustrates how global variables, declared in C, are accessed by assembly procedures. The string variable is declared as a global variable in the C program, as shown in Program 13.47 (line 9). The assembly language procedure computes the string length by accessing the global string variable, as shown in Program 13.48. The procedure call is parameter-less in this example (see line 16 of the C program). The string variable is declared as an external variable in the assembly code (line 7) with an underscore, as it is an external variable.

Program 13.47 A string processing example—C code (in file `string_c.c`)

```
 1: /**********************************************************
 2:  * A string processing example. Demonstrates processing   *
 3:  * global variables. Calls the string_length              *
 4:  * assembly language program in file string_a.asm file.  *
 5:  **********************************************************/
 6: #include <stdio.h>
 7: #define LENGTH 256
 8:
 9: char string[LENGTH];
10: int main(void)
11: {
12:  extern int string_length (char a[]);
13:
14:  printf("Enter string: ");
15:  scanf("%s", string);
16:  printf("string length = %d\n", string_length());
17:  return 0;
18: }
```

Program 13.48 A string processing example—Assembly code (in file string_a.asm)

```
;-----------------------------------------------------------
; String length function works on the global string
; (defined in the C function). It returns string length.
;-----------------------------------------------------------
.MODEL SMALL
.DATA
        EXTRN   _string:byte
.CODE
PUBLIC  _string_length
_string_length  PROC
        mov     AX,0              ; AX keeps the character count
        mov     BX,OFFSET _string ; load BX with string address
repeat:
        cmp     BYTE PTR[BX],0    ; compare with NULL character
        jz      done
        inc     AX                ; increment string length
        inc     BX                ; inc. BX to point to next char.
        jmp     repeat
done:
        ret
_string_length  ENDP
        END
```

13.4 Calling C Functions from Assembly

So far we have considered how a C function can call an assembler procedure. Sometimes it is desirable to call a C function from an assembler procedure. This scenario often arises when we want to avoid writing assembly code for performing complex tasks. Instead, a C function could be written for those tasks. This section illustrates how we can access C functions from assembly procedures. Essentially, the mechanism is the same—we use the stack as the communication medium, as shown in the next example.

Example 13.4 *An example to illustrate a C function call from an assembly procedure*

The main C function requests a set of marks of a class and passes this array to the assembly procedure `stats`, as shown in Program 13.49. The assembly procedure `stats` computes the minimum, maximum and rounded average marks and returns these three values to the C main function (see Program 13.50). To compute the rounded average mark of the class, the C function `find_avg` is called from the assembly procedure. The required arguments `total` and `size` are pushed onto the stack (lines 42 and 43) before calling the C function on line 44. Since the convention for C calls for the caller to clear the stack, line 45 adds 4 to SP to clear the two arguments passed onto `find_avg` C function. The rounded average integer is returned in the AX register.

Program 13.49 An example to illustrate C calls from assembly programs—C code (in file `marks_c.c`)

```
/***********************************************************
 * An example to illustrate C program calling assembly    *
 * procedure and assembly procedure calling a C function. *
 * This program calls the assembly language procedure     *
 * in file MARKS_A.ASM. The program outputs minimum,      *
 * maximum, and rounded average of a set of marks.        *
 **********************************************************/
#include <stdio.h>

#define  CLASS_SIZE  50

int main(void)
{
      int     marks[CLASS_SIZE];
      int     minimum, maximum, average;
      int     class_size, i;
      int     find_avg(int, int);
      extern  void stats(int*, int, int*, int*, int*);

      printf("Please enter class size (<50): ");
      scanf("%d", &class_size);
      printf("Please enter marks:\n");
      for (i=0; i<class_size; i++)
          scanf("%d", &marks[i]);

      stats(marks, class_size, &minimum, &maximum, &average);
      printf("Minimum = %d, Maximum = %d, Average = %d\n",
                        minimum, maximum, average);
      return 0;
}
 /**********************************************************
  * Returns the rounded average required by the assembly
  * procedure STATS in file MARKS_A.ASM.
  **********************************************************/
int find_avg(int total, int number)
{
      return((int)((double)total/number + 0.5));
}
```

Program 13.50 An example to illustrate C calls from assembly programs—Assembly code (in file `marks_a.asm`)

```
;-----------------------------------------------------------
; Assembly program example to show call to a C function.
; This procedure receives a marks array and class size
; and returns minimum, maximum, and rounded average marks.
;-----------------------------------------------------------
.MODEL SMALL
EXTRN   _find_avg:PROC
.CODE
PUBLIC _stats
_stats  PROC
        push    BP
        mov     BP,SP
        push    SI
        push    DI
        ; AX keeps minimum number and DX maximum
        ; Marks total is maintained in SI
        mov     BX,[BP+4]     ; BX := marks array address
        mov     AX,[BX]       ; min := first element
        mov     DX,AX         ; max := first element
        xor     SI,SI         ; total := 0
        mov     CX,[BP+6]     ; CX := class size
repeat1:
        mov     DI,[BX]       ; DI := current mark
        ; compare and update minimum
        cmp     DI,AX
        ja      skip1
        mov     AX,DI
skip1:
        ; compare and update maximum
        cmp     DI,DX
        jb      skip2
        mov     DX,DI
skip2:
        add     SI,DI           ; update marks total
        add     BX,2
        loop    repeat1
        mov     BX,[BP+8]       ; return minimum
        mov     [BX],AX
        mov     BX,[BP+10]      ; return maximum
        mov     [BX],DX
        ; now call find_avg C function to compute average
        push    WORD PTR[BP+6] ; push class size
        push    SI              ; push total marks
        call    _find_avg       ; returns average in AX
        add     SP,4            ; clear stack
        mov     BX,[BP+12]      ; return average
        mov     [BX],AX
        pop     DI
        pop     SI
        pop     BP
        ret
_stats  ENDP
        END
```

13.5 Simplified Calling Mechanisms

The task of interfacing C and assembly language programs can be simplified by using features of TASM. We have so far, on purpose, not used these features in order to focus on the fundamental mechanisms involved. In this section we discuss three features of TASM that simplify high-level language interface.

13.5.1 The ARG Directive

When writing an assembly language program such as minmax_a.asm, we have to calculate the offsets of the arguments (relative to BP). By using the ARG directive, we can let the assembler do this job for us. All we have to do is list the arguments passed on to the procedure by the C program in an ARG directive. The order of these arguments should be the same as that in the C call. Also make sure that all arguments are listed in a single ARG statement. For each argument, its type field should be specified. If a type field is not specified, TASM assumes WORD for 16-bit models and DWORD for 32-bit models. The next example illustrates how the ARG directive can be used.

Example 13.5 *An* ARG *directive example*

In order to show how the ARG directive can be used, we have rewritten the minmax procedure using the ARG directive (see Program 13.51). Notice that the arguments are listed in the same order as in the C call. The ARG statement on line 11 uses \ so that we can continue it on the next line. Remember that \ can be used to extend an assembly language line beyond 80 characters. The ARG directive computes the required offsets. We can refer to these offsets by their names. For example, see lines 16, 17, 22, 32, and 34 in Program 13.51.

Program 13.51 An example to demonstrate the use of the ARG directive (in file minmax2a.asm)

```
 1:  ;-----------------------------------------------------------
 2:  ; Assembly program for the min_max function -- called from
 3:  ; the C program in file minmax_c.c. This function finds the
 4:  ; minimum and maximum of the three integers received by it.
 5:  ; Uses ARG to simplify offset calculations of arguments.
 6:  ;-----------------------------------------------------------
 7:  .MODEL SMALL
 8:  .CODE
 9:  PUBLIC _min_max
10:  _min_max   PROC
11:          ARG     v1:WORD, v2:WORD, v3:WORD,\
12:                  min_ptr:PTR WORD, max_ptr:PTR WORD
13:          push    BP
14:          mov     BP,SP
15:          ; AX keeps minimum number and DX maximum
16:          mov     AX,[v1]       ; get value 1
17:          mov     DX,[v2]       ; get value 2
18:          cmp     AX,DX         ; value 1 < value 2?
19:          jl      skip1         ; if so, do nothing
20:          xchg    AX,DX         ; else, exchange
21:  skip1:
22:          mov     CX,[v3]       ; get value 3
23:          cmp     CX,AX         ; value 3 < min in AX?
24:          jl      new_min
25:          cmp     CX,DX         ; value 3 < max in DX?
```

```
        jl      store_result
        mov     DX,CX
        jmp     store_result
new_min:
        mov     AX,CX
store_result:
        mov     BX,[min_ptr]  ; BX := &minimum
        mov     [BX],AX
        mov     BX,[max_ptr]  ; BX := &maximum
        mov     [BX],DX
        pop     BP
        ret
_min_max   ENDP
        END
```

13.5.2 Extended CALL Instruction

In this section, we discuss two other simplifications of high-level language interface. The first one is a minor one that eliminates the need for prefixing underscores to external functions and variables. We can let the assembler do this for us by specifying that the C language is used. We do this by writing PUBLIC C ... instead of PUBLIC ... as shown in Program 13.52 (see line 10). We can follow the same method for the EXTRN directive, as shown on line 8 of Program 13.52.

Program 13.52 An example showing the use of the extended CALL instruction (in file marks2a.asm)

```
 1:  ;-----------------------------------------------------------
 2:  ; Assembly program example to show call to a C function.
 3:  ; This procedure receives a marks array and class size
 4:  ; and returns minimum, maximum, and rounded average marks.
 5:  ; Uses TASM's extended procedure call instruction.
 6:  ;-----------------------------------------------------------
 7:  .MODEL SMALL
 8:  EXTRN C  find_avg:PROC
 9:  .CODE
10:  PUBLIC C stats
11:  stats   PROC
12:          ARG     marks:PTR WORD, class_size:WORD, min:PTR WORD,\
                max:PTR WORD, avg:PTR WORD
        push    BP
        mov     BP,SP
        push    SI
        push    DI
        ; AX keeps minimum number and DX maximum
        ; Marks total is maintained in SI
        mov     BX,[marks]   ; BX := marks array address
        mov     AX,[BX]      ; min := first element
        mov     DX,AX        ; max := first element
        xor     SI,SI        ; total := 0
        mov     CX,[class_size]
repeat1:
        mov     DI,[BX]      ; DI := current mark
        ; compare and update minimum
        cmp     DI,AX
        ja      skip1
        mov     AX,DI
skip1:
        ; compare and update maximum
        cmp     DI,DX
        jb      skip2
        mov     DX,DI
skip2:
        add     SI,DI        ; update marks total
        add     BX,2
        loop    repeat1
        mov     BX,[min]     ; return minimum
        mov     [BX],AX
        mov     BX,[max]     ; return maximum
        mov     [BX],DX
        ; now call find_avg C function to compute average
        ; returns the rounded average value in AX
        call    find_avg C, SI, class_size
        mov     BX,[avg]     ; return average
        mov     [BX],AX
        pop     DI
        pop     SI
        pop     BP
        ret
stats   ENDP
        ENDs
```

The other feature relieves us from pushing the arguments onto the stack before a procedure call. We can use the CALL instruction extensions to let the assembler insert the necessary push instructions to place the arguments in the correct order (right-pushing for C and left-pushing for Pascal) and to clear the stack (in case of C). The syntax of the CALL instruction is

```
CALL    destination [language[,arg1]...]
```

where `language` is C, CPP, Pascal, FORTRAN, etc. and `arg` is any valid argument that can be pushed onto the stack. We use this extended CALL on line 46 to call the C function `find_avg`.

To summarize, the extended CALL instruction does the following three things to simplify procedure calls:

- Pushes the arguments in the correct order (based on the language specified);
- Prefixes an underscore if required (as in the C language);
- Clears the stack of the arguments if needed (as in the C language).

13.6 Inline Assembly Code

In the inline assembly method, assembly language statements can be embedded into the C code. Such assembly statements are identified by placing the `asm` keyword before the assembly language instruction, as shown in Program 13.53. The end of an inline `asm` statement is indicated by either a semicolon (;) or a newline. Multiple assembly language instructions can be written on the same `asm` line provided they are separated by semicolons, as shown below:

```
asm xor    AX,AX;  mov    AL,DH
```

If there are multiple assembly language instructions, we can also use braces to compound them, as shown below:

```
asm {
        xor    AX,AX
        mov    AL,DH
    }
```

Make sure to place the first brace on the same line as the `asm` keyword. To include comments on `asm` statements, use C-style comments. You cannot use assembly language-style comments (that start with a semicolon).

Example 13.6 *Example with inline assembly code*

This is a simple example to illustrate how inline assembly code can be written. The C function current_month is entirely written in assembly language. It uses int 21H to read the current month. The date information is provided by service 2AH under int 21H, as detailed 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)

Program 13.53 An example showing inline assembly code (in file inlineex.c)

```
/*****************************************************************
 * This program illustrates how inline assembly code can be      *
 * written. It uses the interrupt service of DOS (int 21H)       *
 * to get the current month information.                         *
 *****************************************************************/
#include       <stdio.h>

int current_month(void);

int main(void)
{
      printf ("Current month is: %d\n", current_month());
      return 0;
}
int current_month(void)
{
      asm  mov   AH,2AH
      asm  int   21H
      asm  xor   AX,AX   /* we really want to clear AH */
      asm  mov   AL,DH
}
```

Compiling Inline Assembly Programs

Borland C++ can handle inline assembly code in one of two ways:

- Convert the C code first into assembly language and then invoke TASM to produce an object (`.obj`) file. We call this the TASM method.
- Use the built-in assembler (BASM) to assemble the `asm` statements in the C code. We call this the BASM method.

The process involved in compiling using these two methods is shown in Figure 13.3. The BASM approach is restricted in the sense that only 16-bit instructions can be used. If you use 32-bit instructions (e.g., 80486 or Pentium instructions), the Borland C++ compiler generates an error message. Then you can either simplify the inline code to avoid the instructions that BASM will not accept, or use the other method that invokes TASM. Using the BASM method does not require any special attention to compile an inline program.

In the TASM method, you can use the `-B` compiler option so that the compiler first generates an assembly language file and then invokes TASM to assemble it into the `.obj` file. Alternatively, you can include

```
#pragma   inline
```

at the beginning of the C file to instruct the compiler to use the `-B` option. The TASM method has the advantage of utilizing the capability of TASM, and hence you are not restricted to a subset of the assembly language instructions as in the BASM method.

13.7 Summary

We introduced the principles involved in mixed-mode programming. We discussed the main motivation for writing mixed-mode programs. This chapter focused on mixed-mode programming involving C and the assembly language. Using the Borland C++ compiler and Turbo Assembler software, we demonstrated how assembly language procedures are called from C, and vice versa. Once you understand the principles discussed in this chapter, you can easily handle any type of mixed-mode programming activity.

13.8 Exercises

13–1 Why do we need to write mixed-mode programs?

13–2 Find out details about how you can compile mixed-mode programs with your compiler (if it is other than the Borland C++ compiler).

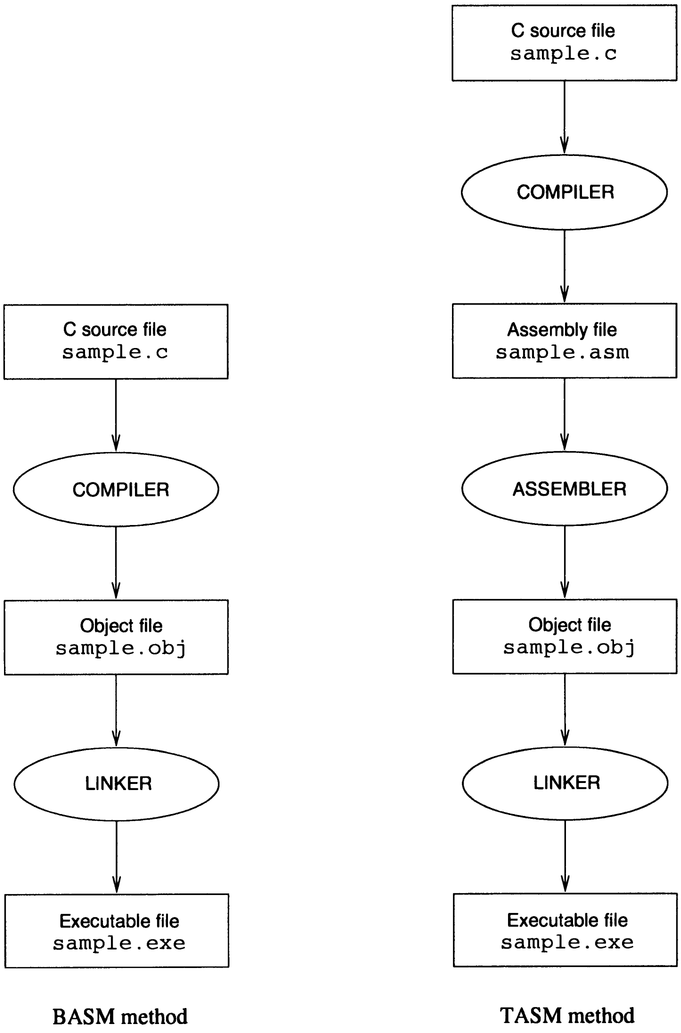


Figure 13.3 Steps involved in compiling mixed-mode programs with inline assembly code.

13–3 Describe how parameters are passed from a C calling function to an assembly language procedure.

13–4 For your compiler, describe how 8-, 16-, and 32-bit values are returned to a C function.

13–5 For your compiler, which registers should be preserved by an assembly procedure?

13–6 What is the difference between a right-pusher and a left-pusher language (as far as parameter passing is concerned)?

13–7 Why does C use right-pushing while Pascal uses left-pushing of arguments?

13–8 Explain why in C, the calling function is responsible for clearing the stack.

13–9 What are the pros and cons of inline assembly as opposed to separate assembly modules?

13–10 What are the significant differences between the BASM and TASM methods for writing inline assembly code?

13.9 Progamming Exercises

13–P1 Write a mixed-mode program to display the time in the format `hh:mm:ss`. The C main program should call the assembly procedure `get_time` and pass on to it three pointers for returning hours, minutes, and seconds. The assembly procedure uses the function 2CH of interrupt 21H to get the current time. Details of this service 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)

13–P2 Write a program that requests the user for a string and a substring 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. The C main program then calls an assembly language procedure to find the location of the substring. This procedure receives two pointers to strings `string` and `substring` and searches 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 is returned if no match is found. For example, if

`string` = Good things come in small packages.

and

`substring` = in

the procedure should return 8, indicating a match of `in` in `things`.

13–P3 Write a mixed-mode inline assembly 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.). The C main program is responsible for displaying the date. However, the C function calls the function `get_date` that receives three pointers to variables day, month, and year. To get the current date, you can use the function 2AH of interrupt 21H. Details of this service 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)

13–P4 Write a program to read a matrix (maximum size 10×10) from the user and display the transpose of the matrix. To obtain the transpose of matrix **A**, write rows of **A** as columns. Here is an example:
If the input matrix is

$$\begin{bmatrix} 12 & 34 & 56 & 78 \\ 23 & 45 & 67 & 89 \\ 34 & 56 & 78 & 90 \\ 45 & 67 & 89 & 10 \end{bmatrix}$$

the transpose of the matrix is

$$\begin{bmatrix} 12 & 23 & 34 & 45 \\ 34 & 45 & 56 & 67 \\ 56 & 67 & 78 & 89 \\ 78 & 89 & 90 & 10 \end{bmatrix}$$

The C part of your program is responsible for getting the matrix and for displaying the result. The transpose should be done by an assembly procedure. Devise an appropriate interface between the two procedures.

13–P5 Write a mixed-mode program that reads a string of characters as input and displays the number of alphabetic characters (i.e., A–Z and a–z) and number of digit characters (i.e., 0–9). 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 back. The assembly procedure `count` calls the C library functions `isalpha` and `isdigit` to determine if a character is an alpha or digit character, respectively.

13–P6 We know that

$$1 + 2 + 3 + \ldots + N = \frac{N \times (N + 1)}{2}$$

Write a program that requests N as input and computes the lefthand side and the righthand side of the equation and 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 back to the C main function. The assembly program computes the lefthand side and calls a C function to compute the righthand side (it passes the N value to the C function). If the lefthand side is equal to the righthand side, the assembly procedure returns the result of the calculation. Otherwise, a negative value is returned to the main C function.

Part IV

Appendices

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 character representation

Modern computer systems are built around millions of tiny switches implemented by what are known as transistors. Each switch can be in one of two states: open or closed. These two states are used to represent 0 and 1, the basic alphabet of any digital computer. Everything that a computer should understand must be expressed in this simple alphabet—computer instructions, data, etc. This appendix examines how data are represented internally in a computer system.

We consider two types of data—numbers and characters. Representing numbers is a two-step process. First, we have to select a number system to use. 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, in Section A.1. Section A.2 discusses conversion of numbers among the number systems. We will 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. A *brief discussion of character representation is given in Section A.6. 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 ten 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 have 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 is in 10's place, and 5 is 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

$$d_n b^n + d_{n-1} b^{n-1} + \ldots + d_1 b^1 + d_0 b^0 \tag{A.1}$$

where b is the base of the number system. 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 ten fingers and toes to count. The alphabet consists of ten 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 $(2^n - 1)$—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 shall see 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, etc.—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 520, i.e., multiply each digit by its weight and add the results together. Note that these arithmetic calculations are done in the decimal system.

Example A.1 *Conversion from binary to decimal*

Convert the binary number 10100111B into its equivalent in the decimal system.

$$\begin{aligned} 10100111\text{B} &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + \\ &\qquad 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 167\text{D} \end{aligned}$$

Notice that the positional weights increase from right to left as

$$1, 2, 4, 8, 16, 32, 64, 128.$$

□□□□□□

Example A.2 *Conversion from octal to decimal*

Convert the octal number 247Q into its equivalent in the decimal system.

$$\begin{aligned} 247\text{Q} &= 2 \cdot 8^2 + 4 \cdot 8^1 + 7 \cdot 8^0 \\ &= 167\text{D} \end{aligned}$$

□□□□□□

Example A.3 *Conversion from hexadecimal to decimal*

Convert the hexadecimal number A7H into its equivalent in the decimal system.

$$\begin{aligned} \text{A7H} &= \text{A} \cdot 16^1 + 7 \cdot 16^0 \\ &= 10 \cdot 16^1 + 7 \cdot 16^0 \\ &= 167\text{D} \end{aligned}$$

□□□□□□

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:

$$\begin{aligned} d_1d_0 &= d_1 \times b^1 + d_0 \times b^0 \\ &= (d_1 \times b) + d_0 \\ d_2d_1d_0 &= d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 \\ &= ((d_2 \times b) + d_1)b + d_0 \\ d_3d_2d_1d_0 &= d_3 \times b^3 + d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 \\ &= (((d_3 \times b) + d_2)b + d_1)b + d_0 \end{aligned}$$

The following algorithm summarizes this process.

Algorithm: Conversion from base b to the decimal system

Input: A number $d_{n-1}d_n \cdots d_1d_0$ 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:

> *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 $d_{n-1}d_n \cdots d_1d_0$ in decimal

Output: Equivalent number in the target base b number system

Procedure: Result digits are obtained from left to right. MOD is the modulo operator and DIV is 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 in the binary system.

		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 523.

□□□□□□

To understand why this algorithm works, let M be the decimal number that we want to convert into its equivalent representation in the base-b target number system. Let $d_n d_{n-1} \ldots d_1 d_0$ be the equivalent number in the target system. Then

$$\begin{aligned} M &= d_n d_{n-1} \ldots d_1 d_0 \\ &= d_n \cdot b^n + d_{n-1} \cdot b^{n-1} + \cdots + d_1 \cdot b^1 + d_0 \cdot b^0 \end{aligned}$$

Now, to get d_0, divide M by b.

$$\begin{aligned} \frac{M}{b} &= (d_n \cdot b^{n-1} + d_{n-1} \cdot b^{n-2} + \cdots + d_1) + \frac{d_0}{b} \\ &= Q_1 + \frac{d_0}{b} \end{aligned}$$

Since d_0 is less than b, it represents the remainder of M/b division. To obtain the d_1 digit, divide Q_1 by b. The 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 = 247Q. From Example A.2 on page 524, 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 = 7AH, which is the correct answer (see Example A.3 on page 524). □□□□□□

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 3 bits at a time, while binary to hexadecimal conversion requires converting 4 bits at a time.

Binary/Octal Conversion

To convert a binary number into an equivalent octal number, form 3-bit groups starting from the right. Add extra 0's at the lefthand 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 0's on the lefthand side does not change the value of a

Table A.1 3-bit binary to octal conversion

3-bit Binary	Octal digit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

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 0's. For example, for octal digit 0, write the three bits 000.

Example A.8 *Conversion from octal to binary*

The following two examples illustrate conversion from octal to binary.

$$105\text{Q} = \overbrace{001}^{1}\ \overbrace{000}^{0}\ \overbrace{101}^{5}\,\text{B}$$

$$247\text{Q} = \overbrace{010}^{2}\ \overbrace{100}^{4}\ \overbrace{111}^{7}\,\text{B}$$

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 0's at the left.

Table A.2 4-bit binary to hexadecimal conversion

4-bit Binary	Hex digit	4-bit Binary	Hex digit
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

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 0's 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 the 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.

$$\begin{array}{l}\text{Hex} \Longrightarrow \text{Binary} \Longrightarrow \text{Octal}\\ \text{Octal} \Longrightarrow \text{Binary} \Longrightarrow \text{Hex}\end{array}$$

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., non-negative) 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 0's 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

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.*

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 8-bit numbers*

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

$$\begin{array}{rl}
 & \texttt{001110} \leftarrow \mathrm{C_{in}} \\
\texttt{174D} & = \texttt{10101110B} \\
\underline{\texttt{75D}} & = \underline{\texttt{01001011B}} \\
\texttt{249D} & = \texttt{11111001B}
\end{array}$$

□□□□□□

An overflow is said to have occurred if there is a carry-out of the leftmost bit position, as shown in the following example.

Example A.12 *Binary addition with overflow*

Addition of 174D and 91D results in an overflow, as the result is outside the range of the numbers represented by using 8 bits.

$$\begin{array}{rcl}
 & \overset{\text{indicates overflow}}{\downarrow} & \\
 & \texttt{11111110} \leftarrow \mathrm{C_{in}} & \\
\texttt{174D} & = & \texttt{10101110B} \\
\underline{\texttt{91D}} & = & \underline{\texttt{01011011B}} \\
\texttt{265D} & \neq & \texttt{00001001B}
\end{array}$$

The overflow condition implies that the sum is not in the range of numbers that can be represented using 8 bits—which is 0 through 255D. To represent 265D, we need 9 bits. You can verify that 100001001B is the binary equivalent of 265D. □□□□□□

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

Subtraction of Unsigned Binary Numbers

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 8-bit numbers*

Perform binary subtraction of 110D from 201D.

```
         1111110 ← Bin
201D  = 11001001B
110D  = 01101110B
-----   ---------
 91D  = 01011011B
```

□□□□□□

If a 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
            ↓
            11111110 ← 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 shall see 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, multiplying two 64-bit numbers is not supported by Pentium. While 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 consider decimal multiplication like 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 have started with the least significant digit of the multiplier, and the partial product 123×6 = 738 is computed. The next higher digit (5) of the multiplier is used to generate the next partial product 123 ×5 = 615. But since digit 5 has a positional weight of 10, we should actually do 123×50 = 6150. This is implicitly done by left shifting the partial product 615 by one digit position. The process is repeated until all digits of the multiplier are processed. Binary multiplication follows exactly the same procedure except that the base-2 arithmetic is performed, as shown in the next example.

Example A.16 *Binary multiplication of unsigned integers*

```
14D ⇒            1110B   ← multiplicand
11D ⇒      ×     1011B   ← multiplier
1110 × 1 ⇒       1110
1110 × 1 ⇒      1110
1110 × 0 ⇒     0000
1110 × 1 ⇒    1110
Product ⇒    10011010B   = 154D
```

□□□□□□

The following algorithm formalizes this procedure with a slight change.

Algorithm: Multiplication of unsigned binary number
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
```

□□□□□□

Binary division is simpler than decimal division because binary numbers are restricted to 0's and 1's—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: dividend is 1011B (11D) and the divisor is 0010B (2D). The dividend is extended to 8 bits by padding 0's at the lefthand 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
- 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, for signed integers. 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 Number representation using 4-bit binary (All numbers except in Binary column are 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

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 non-negative 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

Like in the excess-M representation, negative values are biased as well in the next two representations—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 merely complementing individual bits (converting 0's to 1's 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 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 $\longrightarrow$complement bits$\longrightarrow$ 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—the carry out of the leftmost bit (i.e., sign bit) should be added to the result. Since 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 of 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 of 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 of 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.

$$\begin{array}{rcl}
\texttt{+5D} & = & \texttt{0101B} \\
\underline{\texttt{+3D}} & = & \underline{\texttt{0011B}} \\
\texttt{+8D} & \neq & \texttt{1000B} \quad (= -7\text{D})
\end{array}
\qquad\qquad
\begin{array}{rcl}
\texttt{-5D} & = & \texttt{1010B} \\
\texttt{-4D} & = & \underline{\texttt{1011B}} \\
 & & \texttt{10101B} \\
 & & \underline{\quad\hookrightarrow \texttt{1}} \\
\texttt{-9D} & \neq & \texttt{0110B} \quad (= +6\text{D})
\end{array}$$

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 notice that in both cases, the sign bit of the result is reversed. In fact, this is the condition to detect overflow when signed numbers are added. Also note that overflow can only occur with addition if both operands have the same sign.

Subtraction

Subtraction can be treated as the addition of a negative number. We have already seen this in Example A.20.

Example A.22 *Subtraction in 1's complement representation*

To subtract 7 from 4 (i.e., to perform 4 − 7), get 1's complement representation of −7 and add this to 4.

$$\begin{array}{rl}
\texttt{+4D} = \texttt{0100B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{0100B} \\
\underline{\texttt{-7D}} = \texttt{0111B} \xrightarrow{\text{1's complement}}\longrightarrow\longrightarrow & \underline{\texttt{1000B}} \\
\texttt{-3D} = & \texttt{1100B}
\end{array}$$

The result is 1100B = −3, which is the correct answer. □□□□□□

Overflow

The overflow condition cannot arise with subtraction if the operands involved are of the same sign. The overflow condition can be detected here if the sign of the result is the same as that of the subtrahend (i.e., second operand).

Example A.23 *A subtraction example with overflow*

Subtract −3 from 5 (i.e., perform 5 − (−3)).

$$\begin{array}{lll}
\texttt{+5D} & = \texttt{0101B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{0101B} \\
\texttt{-(-3D)} & = \texttt{1100B} \xrightarrow{\text{1's complement}}\longrightarrow\longrightarrow & \underline{\texttt{0011B}} \\
\overline{\texttt{+8D}} & \neq & \texttt{1000B}
\end{array}$$

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)).

$$\begin{array}{lll}
\texttt{-5D} & = \texttt{1010B} \longrightarrow\longrightarrow\longrightarrow\longrightarrow & \texttt{1010B} \\
\texttt{-(+3D)} & = \texttt{0011B} \xrightarrow{\text{1's complement}}\longrightarrow\longrightarrow & \underline{\texttt{1100B}} \\
 & & \texttt{10110B} \\
 & & \underline{\hookrightarrow \ \texttt{1}} \\
\overline{\texttt{-8D}} & \neq & \texttt{0111B}
\end{array}$$

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.

$$\begin{array}{llr}
6D = \texttt{0110B} \longrightarrow & \text{complement} \longrightarrow & \texttt{1001B}\\
 & \text{add 1} & \texttt{1B}\\ \cline{3-3}
 & & \texttt{1010B}
\end{array}$$

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, like 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 magnitude of a negative number in 2's complement representation*

Find the magnitude of the 2's complement integer 1010B.
Since the most significant bit is 1, we know that it is a negative number.

$$\begin{array}{llrl}
\texttt{1010B} \longrightarrow & \text{complement} \longrightarrow & \texttt{0101B} & \\
 & \text{add 1} & \texttt{1B} & \\ \cline{3-3}
 & & \texttt{0110B} & (= 6D)
\end{array}$$

The magnitude is 6. That is, 1010B = −6D. □□□□□□

Addition and Subtraction

Both of these operations work in the same manner as in the case of 1's complement representation except that any carry out of 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 in a similar manner as 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 right of the decimal point has the weight 10^{-1}, the next position has the weight 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 weights 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

0.11001B

the decimal value it represents is

$$1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 0.78125\text{D}$$

The period in the binary system is referred to as the *binary point*. Thus, the algorithm to convert a binary fraction to its equivalent in decimal is straightforward.

Algorithm: Binary fraction to decimal
Input: A fractional binary number $0.d_1d_2 \cdots d_{n-1}d_n$ with n bits
(trailing 0's 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.

$$\begin{array}{rcllc}
0.78125 \times 2 & = & 1.5625 & \longrightarrow & 1 \\
0.5625 \times 2 & = & 1.125 & \longrightarrow & 1 \\
0.125 \times 2 & = & 0.25 & \longrightarrow & 0 \\
0.25 \times 2 & = & 0.5 & \longrightarrow & 0 \\
0.5 \times 2 & = & 1.0 & \longrightarrow & 1 \\
\text{Terminate} & & & &
\end{array}$$

The binary fraction is 0.11001B, which is obtained by taking numbers from 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 goes as the first digit immediately 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 you 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 octal equivalent.

$$\begin{array}{rcllc}
0.78125 \times 8 & = & 6.25 & \longrightarrow & 6 \\
0.25 \times 8 & = & 2.0 & \longrightarrow & 2 \\
\text{Terminate} & & & &
\end{array}$$

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 0's at 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.

$$0.\underbrace{1100}_{12=\text{C}}\,\underbrace{1\mathbf{000}}_{8} = 0.\text{C8H}$$

Notice that we have to add three 0's on the right to make the number of bits equal to 8 (a multiple of 4). □□□□□□

This conversion process is given by the following algorithm.

Algorithm: Conversion of fractions from decimal to base *b* system
Input: Decimal fractional number
Output: Its equivalent in base *b* with a maximum of *F* digits
Procedure: The function `integer` returns the integer part of the argument and the function `fraction` returns the fractional part.

```
value := fraction to be converted
digit_count := 0
repeat
    next digit of the result := integer (value × $b$)
    value := fraction (value × $b$)
    digit_count := digit_count + 1
 ((value = 0) OR (digit_count = $F$))
```

We will 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 553.

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 $+127$D (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 ($F = 3$), then we have to round the fractional part of a number to the nearest 0.125 $(= 2^{-3})$ to represent. 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 $\frac{1}{2^{F+1}}$.

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 the 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 $\frac{1}{2^{3+1}} = 0.0625$.

If we increase F by 1 bit to 4 bits, the range decreases approximately by half to $0 \leq X < 2^4$ (i.e., $0 \leq X < 16$). The precision, on the other hand, improves to $\frac{1}{2^{4+1}} = 0.03125$. □□□□□□

The fixed-point representation is simple but suffers from the serious disadvantage of limited range. This may not be acceptable to most applications. In particular, fixed-point's inability to represent very small and very large numbers without requiring a large number of bits is unacceptable in many applications.

Using scientific notation, we can make better use of the given number of bits to represent a real number. The next section discusses the *floating-point* representation that 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, 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:

$$\begin{array}{rcl} 1.2345 & \times & 10^{45} \\ 123.45 & \times & 10^{43} \\ 0.00012345 & \times & 10^{49} \end{array}$$

This causes implementation problems to perform arithmetic operations, comparisons, etc. This problem can be avoided by introducing a standard form—called *normal form*. Reverting back 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

$$+1101.101 \times 2^{+11010}$$

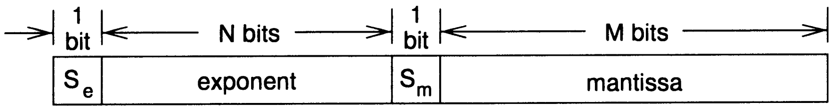
is

$$+1.101101 \times 2^{+11101}$$

We normally write such numbers as

$$+1.101101E11101$$

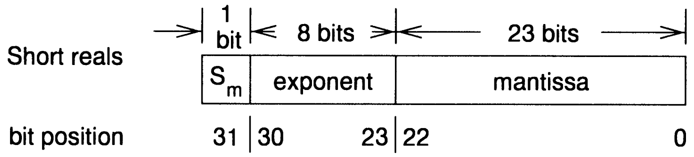
To represent such normalized numbers, we might use the format shown below:

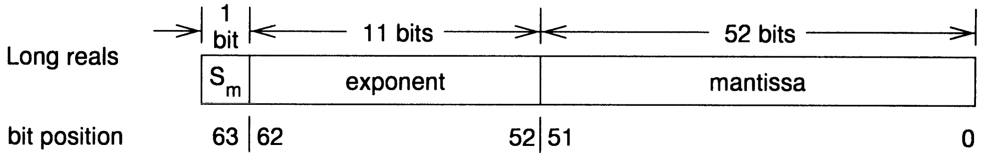


where S_m and S_e represent the sign of mantissa and exponent, respectively.

Implementation of floating-point numbers on computer systems vary from this generic format—usually for efficiency reasons or to conform to a standard. From here on, we discuss the specific format used by Pentium, which also 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.

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 will not discuss this format. The remaining two formats are shown below:





Two 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 shall see 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 525).
1b: Convert the fractional part to binary using the procedure described in Section A.5.1 (page 548).
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.

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

Subtraction of Unsigned Binary Numbers

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 8-bit numbers***

Perform binary subtraction of 110D from 201D.

$$\begin{array}{rl} & \texttt{ 1111110} \leftarrow \mathrm{B_{in}} \\ \texttt{201D} & = \texttt{11001001B} \\ \texttt{110D} & = \texttt{01101110B} \\ \hline \texttt{91D} & = \texttt{01011011B} \end{array}$$

□□□□□□

If a 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.

$$\begin{array}{rcll}
 & & \overset{\substack{\text{indicates}\\ \text{underflow}}}{\downarrow} & \\
 & & 11111110 \leftarrow \mathrm{B_{in}} & \\
201D & = & 11001001B & \\
202D & = & 11001010B & \\
\hline
-1D & \neq & 11111111B & \quad (=255D)
\end{array}$$

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 shall see 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, multiplying two 64-bit numbers is not supported by Pentium. While 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 consider decimal multiplication like when you first learned multiplication. Here is an example.

Example A.15 *Decimal multiplication*

$$\begin{array}{rrl}
 & 123 & \leftarrow \text{multiplicand} \\
 & \times\ \ 456 & \leftarrow \text{multiplier} \\
\hline
123 \times 6 \Rightarrow & 738 & \\
123 \times 5 \Rightarrow & 615 & \\
123 \times 4 \Rightarrow & 492 & \\
\hline
\text{Product} \Rightarrow & 56088 &
\end{array}$$

□□□□□□

We have started with the least significant digit of the multiplier, and the partial product 123×6 = 738 is computed. The next higher digit (5) of the multiplier is used to generate the next partial product 123 ×5 = 615. But since digit 5 has a positional weight of 10, we should actually do 123×50 = 6150. This is implicitly done by left shifting the partial product 615 by one digit position. The process is repeated until all digits of the multiplier are processed. Binary multiplication follows exactly the same procedure except that the base-2 arithmetic is performed, as shown in the next example.

Example A.16 *Binary multiplication of unsigned integers*

```
14D ⇒                 1110B   ← multiplicand
11D ⇒          ×      1011B   ← multiplier
                     --------
1110 × 1 ⇒             1110
1110 × 1 ⇒            1110
1110 × 0 ⇒           0000
1110 × 1 ⇒          1110
                   ----------
Product ⇒          10011010B   = 154D
```

□□□□□□

The following algorithm formalizes this procedure with a slight change.

Algorithm: Multiplication of unsigned binary number
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
```

□□□□□□

Binary division is simpler than decimal division because binary numbers are restricted to 0's and 1's—either subtract the divisor or do nothing. Here is an example of binary division.

Floating-point representation used on the IBM PC follows the IEEE 754 standard. There are three components of a floating-point number: mantissa, exponent, and the sign of 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 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	(c) AB15H	(e) 1234Q
(b) 11010011B	(d) 1234H	(f) 100100B

A–4 To represent numbers 0 through 300 (both inclusive), how many digits are required in the following number systems:

1. binary
2. octal
3. hexadecimal

A–5 What are the advantages of the octal and hexadecimal number systems over the binary system?

A–6 Perform the following number conversions:

1. 1011010011B = ______________ Q
2. 1011010011B = ______________ H
3. 1204Q = ______________ B
4. ABCDH = ______________ B

A–7 Perform the following number conversions:

1. 56D = ______________ B

2. 217D = ______________ Q
3. 150D = ______________ H

Verify your answer by converting your answer back to decimal.

A–8 Assume that 16 bits are available to store a number. Specify the range of numbers that can be represented by the following number systems:

1. unsigned integer
2. signed-magnitude
3. excess-1023
4. 1's complement
5. 2's complement

A–9 What is the difference between a half-adder and a full-adder?

A–10 Perform the following operations assuming that the numbers are unsigned integers. Make sure to identify the presence or absence of the overflow or underflow condition.

1. 01011010B + 10011111B
2. 10110011B + 01101100B
3. 11110001B + 00011001B
4. 10011101B + 11000011B
5. 01011010B – 10011111B
6. 10110011B – 01101100B
7. 11110001B – 00011001B
8. 10011101B – 11000011B

A–11 Repeat the above exercise assuming that the numbers are signed integers that use the 2's complement representation.

A–12 Find the decimal equivalent of the following binary numbers assuming that the numbers are expressed in:

1. unsigned integer
2. signed-magnitude
3. excess-1023
4. 1's complement
5. 2's complement

(a) 01101110 (b) 11011011 (c) 00111101
(d) 11010011 (e) 10001111 (f) 01001101

A–13 Convert the following decimal numbers into signed-magnitude, excess-127, 1's complement, and 2's complement number systems. Assume that 8 bits are used to store the numbers.

(a) 60 (b) 0 (c) −120
(d) −1 (e) 100 (f) −99

A–14 Find the decimal equivalent of the following binary numbers:

(a) 10101.0101011 (b) 10011.1101 (c) 10011.1010
(d) 1011.1011 (e) 1101.001101 (f) 110.111001

A–15 Convert the following decimal numbers into the short floating-point format:

1. 19.3125
2. −250.53125

A–16 Convert the following decimal numbers into the long floating-point format:

1. 19.3125
2. −250.53125

A–17 Find the decimal equivalent of the following numbers, which are in the short floating-point format:

1. 7B59H
2. A971H
3. BBC1H

A.9 Progamming Exercises

A–P1 Implement the algorithm on page 524 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 525 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 552 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 Assembly Language Programs

Objectives

- To present the structure of the stand-alone 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 assembly language programs. We begin by taking a look at the structure of assembly language programs that we use in this book. To make the task of writing assembly language programs easier, we make use of the simplified segment directives provided by the assembler. Section B.1 describes the structure of the stand-alone assembly language programs used in this book.

Unlike high-level languages, assembly language does not provide a convenient mechanism to do input/output. To overcome this deficiency, we have provided a set of I/O routines to facilitate character, string, and numeric input/output. These routines are described in Section B.2.

Once we have written an assembly language program, we have to transform it into its executable form. Typically, this takes two steps: we use an assembler to translate the source program into what is called an object program and then use *a linker to transform the object program into an executable version. Section B.3 gives details of these steps. The appendix concludes with a summary.*

B.1 Structure of Assembly Language Programs

Writing an assembly language program is a complicated task, particularly for a beginner. We will make this daunting task simple by hiding those details that are irrelevant. A typical assembly language program consists of three parts. The code part of the program defines the program's functionality by a sequence of assembly language instructions. The code part of the program, after translating it to the machine language code, is placed in the code segment. The data part reserves memory space for the program's data. The data part of the program is mapped to the data segment. Finally, we also need the stack data structure, which is mapped to the stack segment. The stack serves two main purposes: it provides temporary storage and acts as the medium to pass parameters in procedure calls. We will use the template shown in Figure B.1 for writing stand-alone assembly language programs. These are the programs that are written completely in assembly language.

Now let us dissect the statements in this template. This template consists of two types of statements: executable instructions and assembler directives. Executable instructions generate machine code for Pentium to execute when the program is run. Assembler directives, on the other hand, are meant only for the assembler. They provide information to the assembler on the various aspects of the assembly process. In this book, all assembler directives are shown in uppercase letters, while instructions are shown in lowercase.

The TITLE line is optional and when included, usually contains a brief heading of the program and the disk file name. The TITLE information can be up to 128 characters. To understand the purpose of the TITLE directive, you should know that the assembler produces, if you want, a nicely formatted listing file (with extension `.lst`) after the source file has been assembled. In the listing file, each page heading contains the information provided in the TITLE directive.

The COMMENT assembler directive is useful to include several lines of text in assembly language programs. The format of this directive is

```
COMMENT  delimiter [text]
[text]
[text] delimiter [text]
```

where the brackets [ ] indicate optional text. The delimiter is used to delineate the comment block. The delimiter is any nonblank character after the COMMENT directive. The assembler ignores the text following the delimiter until

```
TITLE    brief title of program      file-name
COMMENT  |
           Objectives:
               Inputs:
              Outputs:
         |
.MODEL SMALL

.STACK    100H              ; defines a 256-byte stack

.DATA
(data goes here)

.CODE
.486                        ; not necessary if only 8086
                            ; instructions are used
INCLUDE  io.mac             ; include I/O routines
main  PROC
      .STARTUP              ; setup segments
            .
            .
    (code goes here)
            .
            .
      .EXIT                 ; returns control
main  ENDP
      END    main
```

Figure B.1 Structure of the stand-alone assembly language programs used in this book

the second occurrence of the delimiter. It also ignores any text following the second delimiter on the same line. We use the COMMENT directive to include objectives of the program and its inputs and outputs. For an example, see `sample.asm` given on page 568.

The .MODEL directive specifies a standard memory configuration for the assembly language program. For our purposes, a small model is sufficient. A restriction of this model is that our program's code should be $\leq$ 64K, and the total storage for the data should also be $\leq$ 64K. This directive should precede the .STACK, .DATA, and .CODE directives.

The .STACK directive defines the stack segment to be used with the program. The size of the stack can be specified. By default, we always use a 100H byte (256 bytes or 128 words) stack.

The .DATA directive defines the data segment for the assembly language program. The program's variables are defined here. Chapter 3 discusses various directives to define and initialize variables used in assembly language programs.

The .CODE directive terminates the data segment and starts the code segment. You need to use `.486` only if the code contains instructions of 32-bit processors such as 80486 and Pentium. This line is not necessary if the assembly language code uses only the instructions of the 8086 processor. The INCLUDE directive causes the assembler to include source code from another file (`io.mac` here). The code

```
main  PROC
        .
        .
main  ENDP
```

defines a procedure called `main` using the directives PROC (PROCedure) and ENDP (END Procedure).

The last statement uses the END directive for two distinct purposes:

1. By using the label `main`, it identifies the entry point into the program (first instruction of `main` procedure here);
2. It signals the assembler that the end of the source file has been reached.

The choice of `main` in the template is arbitrary. You can use any other name with the restriction that the same name should appear in all three places.

The .STARTUP assembler directive sets up the data and stack segments appropriately. In its place you can write code to set up the data segment yourself. To do this, use the following code:

```
mov    AX,@DATA
mov    DS,AX
```

These two lines initialize the DS register so that it points to the program's data segment. Note that @DATA points to the data segment.

To return control from the assembly program, use the .EXIT assembler directive. This directive places the code to call the `int 21H` function 4CH to return control. In this directive's place, you can write your own code to call `int 21H`, as shown below:

```
mov    AX,4C00H
int    21H
```

Control is returned to the operating system by interrupt 21H service 4CH. The service required under interrupt 21H is indicated by moving 4CH into the AH register. This service also returns an error code that is given in the AL register. It is good practice to set AL to 0 to indicate normal termination of the program.

B.2 Input/Output Routines

We rarely write programs that do not input and/or output data. High-level languages provide facilities to input and output data. For example, C provides `scanf` and `printf` functions to input and output data, respectively. Analogously, Pascal has `read` and `write` for similar purposes. Typically, high-level languages can read numeric data (integers, floating-point numbers), characters, and strings.

Assembly language, however, does not provide a convenient mechanism to input/output data. The operating system provides some basic services to read and write data, but these are fairly limited. For example, there is no function to read an integer from the keyboard.

In order to facilitate I/O in assembly language programs, it is necessary to write the required procedures. We have written a set of I/O routines to read and display signed integers, characters, and strings. The remainder of the section describes these routines. Each I/O routine call looks like an assembly language instruction. This is achieved by using macros. Each macro call typically expands to several assembly language statements and includes a call to an appropriate procedure. These macros are all defined in the `io.mac` file and actual assembled procedures that perform I/O are in the `io.obj` file. Table 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:

```
PutCh   'A'        ; displays character A
PutCh   AL         ; displays the character in AL
PutCh   response   ; displays the byte located in
                   ; memory (labeled response)
```

The format of `GetCh` is

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

```
GetCh    destination
```

where destination can be either an 8-bit, general-purpose register or a byte in memory. Some examples are:

```
GetCh    DH
GetCh    response
```

In addition, a `nwln` macro is defined to display a newline, which sends a carriage return (CR) and a line feed (LF). It takes no operands.

String I/O

PutStr and GetStr are defined to display and read strings, respectively. The strings are assumed to be in NULL-terminated format. That is, the last character of the string is the NULL ASCII character, which signals the end of the string. Strings are discussed in Chapter 9.

The format of PutStr is

```
PutStr   source
```

where source is the name of the buffer containing the string to be displayed. For example,

```
PutStr   message
```

displays the string stored in the buffer message. Strings are limited to 80 characters. If the buffer does not contain a NULL-terminated string, a maximum of 80 characters are displayed.

The format of GetStr is

```
GetStr   destination [, buffer_size]
```

where destination is the buffer name into which the string from the keyboard is read. The input string can be terminated by a CR. You can also specify the optional buffer_size value. If not specified, a buffer size of 81 is assumed. Thus, in the default case, a maximum of 80 characters are read into the string. If a value is specified, buffer_size−1 characters are read. The string is stored as a NULL-terminated string. You can backspace to correct input. Here are some examples:

```
GetStr   in_string     ; reads at most 80 characters
GetStr   TR_title,41   ; reads at most 40 characters
```

Numeric I/O

There are four macro definitions for performing integer I/O: two are defined for 16-bit integers and two for 32-bit integers. First we look at the 16-bit integer I/O routines—PutInt and GetInt. The formats of these routines are

```
PutInt   source
GetInt   destination
```

where source and destination can be a 16-bit, general-purpose register or the label of a memory word.

PutInt displays the signed number at the source. It suppresses all leading 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 label of a memory doubleword (i.e., 32 bits). For example, if `total` is a 32-bit number in memory, we can display it by

```
PutLint   total
```

and read a long integer from the keyboard into `total` by

```
GetLint   total
```

Some examples that use registers are:

```
PutLint   EAX
GetLint   EDX
```

An Example

Program B.54 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. In both cases, ? indicates that the data is not initialized.

Program B.54 An example assembly program

```
1:  TITLE    An example assembly language program     SAMPLE.ASM
2:  COMMENT  |
3:           Objective: To demonstrate the use of some I/O
4:                      routines and to show the structure
5:                      of assembly language programs.
6:              Inputs: As prompted.
```

```
|          Outputs: As per input.
.MODEL SMALL
.STACK  100H
.DATA
name_msg      DB  'Please enter your name: ',0
query_msg     DB  'How many times to repeat welcome message? ',0
confirm_msg1  DB  'Repeat welcome message ',0
confirm_msg2  DB  ' times? (y/n) ',0
welcome_msg   DB  'Welcome to Assembly Language Programming ',0

user_name    DB  16 DUP (?)   ; buffer for user name
response     DB  ?

.CODE
INCLUDE io.mac

main  PROC
      .STARTUP
      PutStr  name_msg       ; prompt user for his/her name
      nwln
      GetStr  user_name,16   ; read name (max. 15 characters)
      nwln
ask_count:
      PutStr  query_msg      ; prompt for repeat count
      GetInt  CX             ; read repeat count
      nwln
      PutStr  confirm_msg1   ; confirm repeat count
      PutInt  CX             ; by displaying its value
      PutStr  confirm_msg2
      GetCh   response       ; read user response
      nwln
      cmp     response,'y'   ; if 'y', display welcome message
      jne     ask_count      ; otherwise, request repeat count
display_msg:
      PutStr  welcome_msg    ; display welcome message
      PutStr  user_name      ; display the user name
      nwln
      loop    display_msg    ; repeat count times
      .EXIT
main  ENDP
      END     main
```

The program requests the name of the user and a repeat count. After confirming the repeat count, it displays a welcome message repeat count times. We use `PutStr` on line 25 to prompt for the user name. The name is read as a string using `GetStr` into the `user_name` buffer. Since we have allocated only 16 bytes for the buffer, the name cannot be more than 15 characters. We enforce this by specifying the optional buffer size parameter in `GetStr` (line 27). The `PutStr` on line 30 requests a repeat count, which is read by `GetInt` on line 31. The confirmation message is displayed by lines 33–35. The response of the user `y/n` is read by `GetCh` on line 36. If the response is `y`, the loop (lines 40–44) displays the welcome message repeat count times. A sample interaction with the program is shown below:

```
Please enter your name:
Veda
How many times to repeat welcome message? 4
Repeat welcome message 4 times? (y/n) y
Welcome to Assembly Language Programming Veda
Welcome to Assembly Language Programming Veda
Welcome to Assembly Language Programming Veda
Welcome to Assembly Language Programming Veda
```

B.3 Assembling and Linking

Figure B.2 shows the steps involved in converting an assembly language program into an executable program. The source assembly language file (e.g., `sample.asm`) is given as input to the assembler. The assembler translates the assembly language program into an object program (e.g., `sample.obj`). The linker takes one or more object programs (e.g., `sample.obj` and `io.obj`) and combines them into an executable program (e.g., `sample.exe`). The following subsections describe each of these steps in detail.

B.3.1 The Assembly Process

To assemble a program, you need to have an assembler (e.g., TASM.EXE or MASM.EXE). In the remainder of this section we describe Turbo assembler TASM. MASM also works in a similar way (see your assembler documentation). The general format to assemble an assembly language program is

```
TASM  [options] source-file [,obj-file] [,list-file] [,xref-file]
```

where the specification of fields in [ ] is optional. If we simply specify only the source file, TASM just produces only the object file. Thus to assemble our example source file `sample.asm`, type

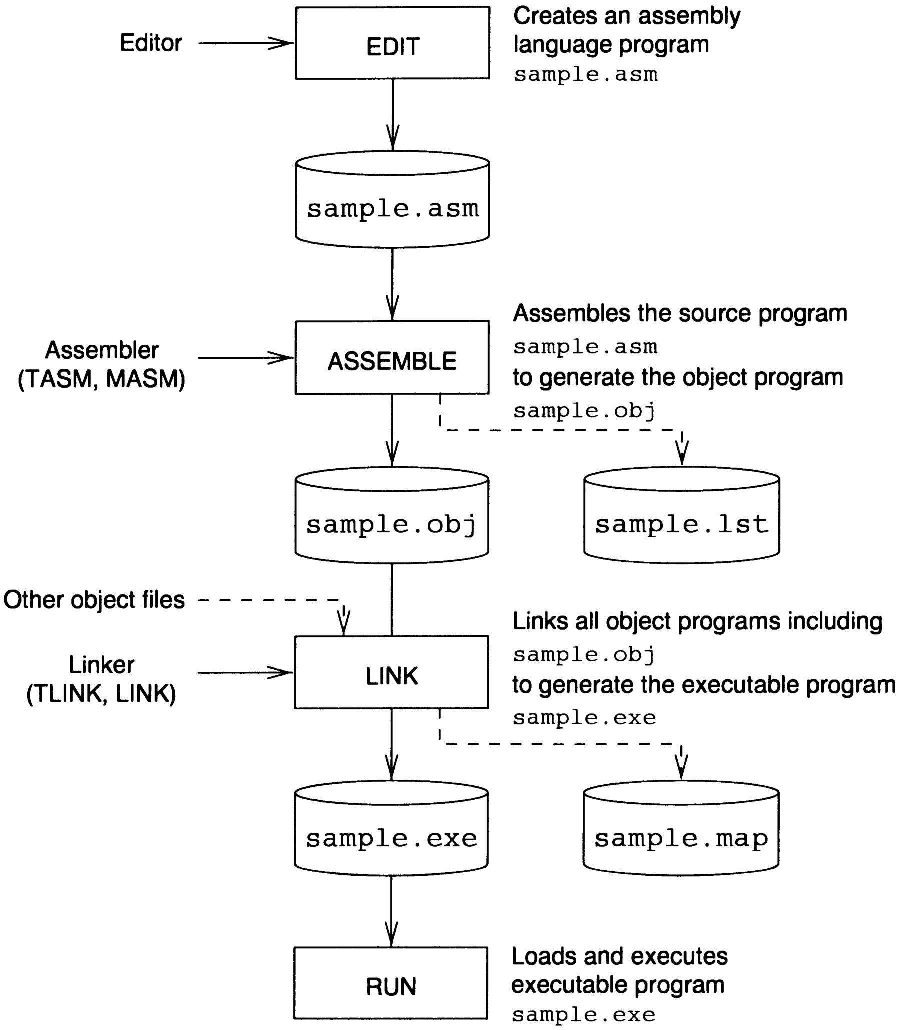

Editor
EDIT
Creates an assembly language program
sample.asm
sample.asm
Assembler (TASM, MASM)
ASSEMBLE
Assembles the source program
sample.asm
to generate the object program
sample.obj
sample.obj
sample.lst
Other object files
Linker (TLINK, LINK)
LINK
Links all object programs including
sample.obj
to generate the executable program
sample.exe
sample.exe
sample.map
RUN
Loads and executes executable program
sample.exe

Figure B.2 Assembling, linking, and executing assembly language programs (optional inputs and outputs are shown by dashed lines).

```
TASM  sample
```

You don't have to type the extension. By default, TASM assumes the `.asm` extension. During the assembly process, TASM displays error messages (if any). After successfully assembling the source program, TASM generates an object file with the same file name as the source file but with the `.obj` extension. Thus, in our example, it generates the `sample.obj` file.

If you want the assembler to generate the listing file, you can use

```
TASM  sample,,
```

This produces two files: `sample.obj` and `sample.lst`. The list file contains detailed information about the assembly process, as we shall see shortly. If you want to use a different file name for the listing file, you have to specify the file name (the extension `.lst` is assumed), as in the following example:

```
TASM  sample,,myprog
```

which generates two files: `sample.obj` and `myprog.lst`.

If the fourth field `xref-file` is specified, TASM generates a listing file containing cross-reference information (discussed shortly).

Options

You can also use command line option L to produce the listing file. For example,

```
TASM  /L sample
```

produces `sample.obj` and `sample.lst` files. During the assembly process, TASM displays error messages but does not display the corresponding source lines. You can use option `Z` to force TASM to display the error source lines. Other interesting options are `N` to suppress symbol table information in the listing file, and `ZI` to include complete debugging information for debuggers (such as Turbo debugger TD). A complete list of options is displayed by typing TASM.

The List File

Program B.55 gives a simple program that reads two signed integers from the user and displays their sum if there is no overflow; otherwise, it displays an error message. The input numbers should be in the range −2,147,483,648 to +2,147,483,647, which is the range of a 32-bit signed number. The program uses `PurStr` and `GetLInt` to prompt and read input numbers (see lines 24, 25 and 29, 30). The sum of the input numbers is computed on lines 34–36. If the resulting sum is outside the range of a signed 32-bit integer, the overflow flag is set by the `add` instruction. In this case, the program displays the overflow message (line 40). If there is no overflow, the sum is displayed (lines 46 and 47).

Program B.55 An assembly language program to add two integers sumprog.asm

```
TITLE   Assembly language program to find sum   SUMPROG.ASM
COMMENT  |
         Objective: To add two integers.
            Inputs: Two integers.
|           Output: Sum of input numbers.
.MODEL SMALL
.STACK  100H
.DATA
prompt1_msg  DB  'Enter first number: ',0
prompt2_msg  DB  'Enter second number: ',0
sum_msg      DB  'Sum is: ',0
error_msg    DB  'Overflow has occurred!',0

number1      DD  ?  ; stores first number
number2      DD  ?  ; stores first number
sum          DD  ?  ; stores sum

.CODE
INCLUDE io.mac
.486
main  PROC
      .STARTUP
      ; prompt user for first number
      PutStr  prompt1_msg
      GetLint number1
      nwln

      ; prompt user for second number
      PutStr  prompt2_msg
      GetLint number2
      nwln

      ; 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
main    ENDP
        END     main
```

The list file for the source program `sumprog.asm` is shown in Program B.56. It contains, in addition to the original source code lines, a lot of useful information about the results of the assembly. This additional information includes the actual machine code generated for the executable statements, offsets of each statement, and tables of information about symbols and segments.

The top line of each page consists of a header that identifies the assembler, its version, date, time, and page number. If TITLE is used, the title line is printed on each page of the listing. There are two parts to the listing file: the first part consists of annotated source code, and the second part gives tables of information about the symbols and segments used by the program.

Source Code Lines

The format of the source code lines is:

```
nesting-level  line#  offset  machine-code  source-line
```

`nesting-level`: the level of nesting of "include files" and macros. We discuss this in Chapter 10, which discusses macros in detail.

`line#`: the number of the listing file line numbers. These numbers are different from the line numbers in the source file. This can be due to include files, macros, etc., as shown in Program B.56.

`offset`: a 4-digit hexadecimal offset value of the machine code for the source statement. For example, the offset of the first instruction (line 31) is 0000, and that of the add instruction on line 45 is 0044H. Source lines such as comments do not generate any offset.

machine-code: the hexadecimal representation of the machine code for the assembly language instruction. For example, the machine language encoding of

```
mov    EAX,number1
```

is 66|A1004B (line 44) and requires 4 bytes (66 is the operand size override prefix). Similarly, the machine language encoding of

```
jmp    done
```

is EB1990 (line 52), requiring 3 bytes of memory. Again, source code lines such as comments do not generate any machine code for obvious reasons.

source-line: a copy of the original source code line. As you can see from Program B.56, the number of bytes required for the machine code depends on the source instruction. When operands are in memory like number1, their relative address value is appended with r (see line 44) to indicate that the actual value is fixed up by the linker when the segment is combined with other segments (for example, io.obj in our example). You will see an e instead of r if the symbol is defined external to the source file (thus available only at link time). For segment values, an s is appended to the relative addresses.

Program B.56 The list file for the example assembly program sumprog.asm

```
\begin{verbatim}
 1:  Turbo Assembler    Version 4.0        07/31/95 14:38:34       Page 1
 2:  sumprog.ASM
 3:  Assembly language program to find sum    SUMPROG.ASM
 4:
 5:
 6:        1                              COMMENT  |
 7:        2                                        Objective: To add two integers.
 8:        3                                           Inputs: Two integers.
 9:        4                              |            Output: Sum of input numbers.
10:        5                              DOSSEG
11:        6  0000                        .MODEL SMALL
12:        7  0000                        .STACK  100H
13:        8  0000                        .DATA
14:        9  0000  45 6E 74 65 72 20 66+ prompt1_msg  DB  'Enter first number: ',0
15:       10        69 72 73 74 20 6E 75+
16:       11        6D 62 65 72 3A 20 00
17:       12  0015  45 6E 74 65 72 20 73+ prompt2_msg  DB  'Enter second number: ',0
18:       13        65 63 6F 6E 64 20 6E+
19:        14        75 6D 62 65 72 3A 20+
20:        15        00
21:        16  002B  53 75 6D 20 69 73 3A+  sum_msg      DB  'Sum is: ',0
22:        17        20 00
23:        18  0034  4F 76 65 72 66 6C 6F+  error_msg    DB  'Overflow has occurred!',0
24:        19        77 20 68 61 73 20 6F+
25:        20        63 63 75 72 72 65 64+
26:        21        21 00
27:        22
28:        23  004B  ????                   number1      DW  ?  ; stores first number
29:        24  004D  ????                   number2      DW  ?  ; stores first number
30:        25  004F  ????                   sum          DW  ?  ; stores sum
31:        26
32:        27  0051                         .CODE
33:        28                               INCLUDE io.mac
34:  1     29
35:  1     30
36:        31
37:        32  0000                         main  PROC
38:        33                                     ; initialize data segment
39:        34  0000  B8 0000s                     mov     AX,@DATA
40:        35  0003  8E D8                        mov     DS,AX
41:        36
42:        37                                     ; prompt the user for first number
43:        38                                     PutStr  prompt1_msg
44:        39                                     GetInt  number1
45:        40                                     nwln
46:        41
47:        42                                     ; prompt the user for second number
48:        43                                     PutStr  prompt2_msg
49:        44                                     GetInt  number2
50:        45                                     nwln
51:        46
52:        47                                     ; find sum of two 16-bit numbers
53:        48  002F  A1 004Br                     mov     AX,number1
54:        49  0032  03 06 004Dr                  add     AX,number2
55:        50  0036  A3 004Fr                     mov     sum,AX
56:        51
57:        52                                     ; check for overflow
58:        53  0039  71 10                        jno     no_overflow
59:        54                                     PutStr  error_msg
60:        55                                     nwln
61:        56  0048  EB 16 90                     jmp     done
62:        57
63:  Turbo Assembler    Version 4.0       07/31/95 14:38:34       Page 2
64:  sumprog.ASM
65:  Assembly language program to find sum     SUMPROG.ASM
66:

     58                                   ; display sum
     59  004B                       no_overflow:
     60                                   PutStr  sum_msg
     61                                   PutInt  sum
     62                                   nwln
     63
     64                                   ; return to DOS
     65  0060                       done:
     66  0060  B8 4C00                    mov     AX,4C00H
     67  0063  CD 21                      int     21H
     68  0065                       main  ENDP
     69                                   END     main
Turbo Assembler    Version 4.0       07/31/95 14:38:34       Page 3
Symbol Table
Assembly language program to find sum     SUMPROG.ASM



Symbol Name                         Type   Value

??DATE                              Text   "07/31/95"
??FILENAME                          Text   "sumprog "
??TIME                              Text   "14:38:32"
??VERSION                   Number 0400
@32BIT                              Text   0
@CODE                               Text   _TEXT
@CODESIZE                   Text   0
@CPU                                Text   0101H
@CURSEG                             Text   _TEXT
@DATA                               Text   DGROUP
@DATASIZE                   Text   0
@FILENAME                   Text   SUMPROG
@INTERFACE                          Text   00H
@MODEL                              Text   2
@STACK                              Text   DGROUP
@WORDSIZE                   Text   2
DONE                                Near   _TEXT:0060
ERROR_MSG                   Byte   DGROUP:0034
MAIN                                Near   _TEXT:0000
NO_OVERFLOW                         Near   _TEXT:004B
NUMBER1                             Word   DGROUP:004B
NUMBER2                             Word   DGROUP:004D
PROC_GETCH                          Near   _TEXT:---- Extern
PROC_GETINT                         Near   _TEXT:---- Extern
PROC_GETLINT                        Near   _TEXT:---- Extern
PROC_GETSTR                         Near   _TEXT:---- Extern
PROC_NWLN                   Near   _TEXT:---- Extern
```

```
PROC_PUTCH                          Near   _TEXT:---- Extern
PROC_PUTINT                         Near   _TEXT:---- Extern
PROC_PUTLINT                        Near   _TEXT:---- Extern
PROC_PUTSTR                         Near   _TEXT:---- Extern
PROMPT1_MSG                         Byte   DGROUP:0000
PROMPT2_MSG                         Byte   DGROUP:0015
SUM                                 Word   DGROUP:004F
SUM_MSG                             Byte   DGROUP:002B
TEMP                                Byte   _TEXT:---- Extern

Macro Name

GETCH
GETINT
GETLINT
GETSTR
NWLN
PUTCH
PUTINT
PUTLINT
PUTSTR

Groups & Segments           Bit Size Align  Combine Class

DGROUP                              Group
  STACK                             16  0100 Para   Stack   STACK
  _DATA                             16  0051 Word   Public  DATA
Turbo Assembler    Version 4.0        07/31/95 14:38:34       Page 4
Symbol Table
Assembly language program to find sum     SUMPROG.ASM


_TEXT                               16  0065 Word   Public  CODE
\end{verbatim}
```

Symbol Table

The second part of the listing file consists of two tables of information. The first one lists all the symbols used in the program in alphabetical order. These include the variables and labels used in the program. For each symbol, the symbol table gives its type and value. For example, number1 and number2 are words with offsets 4BH and 4FH, respectively, in the DGROUP segment group. This segment group has _DATA and STACK segments.

The I/O procedures (PROC_GETCH, etc.) are near procedures that are defined as external in io.mac. Procedures are discussed in Chapter 4. The object code for these procedures is available at the time of linking (`io.obj` file). The macros listed are defined in `io.mac`.

If the fourth field `xref-file` on the TASM command line is specified, the listing file would have contained cross-reference information for each symbol. The cross-reference information gives where (i.e., line number) the symbol was defined and the line numbers of all the lines in the program on which that symbol was referenced.

Group and Segment Table

The other table gives information on groups and segments. Segment groups do not have any attributes and are listed with the segments making up the group. For example, the DGROUP consists of _DATA and STACK segments. Segments, however, have attributes. For each segment, five attributes are listed.

Bit: gives the data size, which is 16 in our case.

Size: indicates the segment size in hex. For example, the STACK segment is 100H (i.e., 256) bytes long.

Align: indicates the type of alignment. This refers to the memory boundaries that a segment can begin. Some alignment types are:

BYTE	Segment can begin at any address
WORD	Segment can begin only at even addresses
PARA	Segment can begin only at an address that is a multiple of 16 (para = 16 bytes)

For example, STACK is para-aligned, while _DATA and _TEXT are word-aligned.

Combine: specifies how segments of the same name are combined. With the PUBLIC combine type, identically named segments are concatenated into a larger segment. The combine type STACK is special and can only be used for the stack.

Class: refers to the segment class—e.g., CODE, DATA, or STACK. The linker uses this information to order segments.

B.3.2 Linking Object Files

Linker is a program that takes one or more object programs as its input and produces an executable program. In our example, since I/O routines are defined separately, we need two object files—`sample.obj` and `io.obj`—to generate the executable file `sample.exe`. To do this, we use the command

```
TLINK sample io
```

The syntax of TLINK is given by

```
TLINK [options] obj-files,exe-file,map-file,lib-file
```

where `obj-files` is a list of object files to be linked, and `exe-file` is the name of the executable file. If no executable file name is given, the name of the first object file specified is used with the `.exe` extension. TLINK, by default, also generates a map file. If no map file name is given on the command line, the first object file name is used with the `.map` extension. `lib-file` specifies library files, and we will not discuss them here.

The map file provides information on segments. The map file generated for the `sample` program is shown below:

```
  Start  Stop   Length Name                  Class

  00000H 0037FH 00380H _TEXT                 CODE
  00380H 0053FH 001C0H _DATA                 DATA
  00540H 0063FH 00100H STACK                 STACK

 Program entry point at 0000:0000
```

For each segment, it gives the starting and ending addresses along with the length of the segment in bytes, its name and class. For example, the CODE segment is named _TEXT and starts at address 0 and ends at 37FH. The length, therefore, is 380H.

If you intend to debug your program using Turbo Debugger, you should use `V` in order to link the necessary symbolic information. For example, the `sample.obj` object program, along with `io.obj`, can be linked by

```
TLINK /V sample io
```

You have to make sure that the `ZI` option has been used during the assembly.

B.4 Summary

Assembly language programs consist of three parts: stack, data, and code segments. These three segments can be defined using simplified segment directives provided by both TASM and MASM assemblers. By means of simple examples, we have seen how a typical stand-alone assembly language program looks using these directives.

Since assembly language does not provide a convenient mechanism to do input/output, we defined a set of I/O routines to help us in performing simple character, string, and numeric input and output. The numeric I/O routines provided can input/output both 16-bit and 32-bit signed integers.

To execute an assembly language program, we have to first translate it into an object program by using an assembler. Then we have to pass this object program, along with any other object programs needed by the program, to a linker to produce an executable program. Both the assembler and linker generate additional files that provide information on the assembly and link processes.

B.5 Exercises

B–1 What is the purpose of the TITLE directive?

B–2 How is the stack defined in the assembly language programs used in this book?

B–3 In the assembly language program structure used in this book, how are the data and code parts specified?

B–4 What is meant by a "stand-alone" assembly language program?

B–5 What is an assembler? What is the purpose of it?

B–6 What files are generated by your assembler? What is the purpose of each of these files?

B–7 What is the function of the linker? What is the input to the linker?

B–8 Why is it necessary to define our own I/O routines in assembly language?

B–9 What is a NULL-terminated string?

B–10 Why is buffer size specification necessary in `GerStr` but not in `PutStr`?

B–11 What happens if the buffer size parameter is not specified in `GetStr`?

B–12 What happens if the buffer specified in `PutStr` does not contain a NULL-terminated string?

B–13 What is the range of numbers that `GetInt` can read from the keyboard? Give an explanation for the range.

B–14 Repeat the last exercise for `GetLint`.

B.6 Progamming 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 the `sample.asm` by deliberately introducing errors into the program. Assemble the program and see the type of errors reported by your assembler. Also, generate the listing file and briefly explain its contents.

B–P4 Assemble the `sample.asm` program to generate cross-reference information. Comment on how this information is presented by your assembler.

Appendix C

Debugging Assembly Language Programs

Objectives

- To present some basic strategies to debug assembly language programs
- To describe the DOS debugger DEBUG
- To explain the basic features of the Turbo debugger (TD)
- To provide a brief discussion of the Microsoft debugger (CodeView)

Debugging assembly language programs is more difficult and time-consuming than debugging high-level language programs. However, the fundamental strategies that work for high-level languages also work for assembly language programs. Section C.1 gives a discussion of these strategies. Since you are familiar with debugging in a high-level language, this discussion is rather brief.

The following three sections discuss three popular debuggers. While the DOS DEBUG is a line-oriented debugger, the other two—Turbo Debugger and CodeView—are window-oriented and are much better. All three share some basic commands required to support debugging assembly language programs.

Our goal in this appendix is to introduce the three debuggers briefly, as the best way to get familiar with these debuggers is to try them. We use a simple example to explain some of the commands of DEBUG (in Section C.2) and Turbo Debugger (in Section C.3). Since CodeView is similar in spirit to the Turbo Debugger, we give only a brief overview of it in Section C.4. The appendix concludes with a summary.

C.1 Strategies to Debug Assembly Language Programs

Programming is a complicated task. Very few real-life programs are ever written that work perfectly the very first time. Loosely speaking, a program can be thought of as mapping a set of input values to a set of output values. The functionality of the mapping performed by a program is given as the specification for the programming task. It goes without saying that when the program is written, it should be verified to meet the specifications. In programming parlance, this activity is referred to as testing and validating the program.

Testing a program itself is a complicated task. Typically, test cases, selected to validate the program, should test each possible path in the program, boundary cases, etc. 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. You have to rely on your intuition and experience. However, there are tools that can help you in this process. We will look at three such tools in this chapter—DEBUG, Turbo Debugger TD, and Microsoft CodeView.

Finding bugs in a program is very much dependent on the individual program. Once an error is detected, there are some general ways of locating the source code lines causing the error. The basic principle that helps 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 single or a few procedures are added to the program to add some specific functionality and test it before adding other functions to the program. In general, it is a bad idea to write the whole program and start the testing process, unless the program is "small." The best strategy is to write code that has as few bugs as possible. This can be achieved by using pseudocode and verifying the logic of the pseudocode even before 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 pseudo-code in your head! Furthermore, don't be in a hurry to write some assembly code that appears to work. This is short sighted, as you will end up spending more time in the debugging phase.

To isolate a bug, program execution should be observed in slow motion. Most debuggers provide a command to execute a program in single-step mode. In this mode, a program executes one statement at a time and pauses. Then you can examine contents of registers, data in memory, stack contents, etc. In this mode, a procedure call is treated as a single statement and the entire procedure is executed before pausing the program. This is useful if you know that the called procedure works correctly. Debuggers also provide another command to trace even the statements of procedure calls, which is useful for testing procedures as well.

Often we know that some parts of the program work correctly. In this case, it is a sheer waste of time to single step or trace the code. What we would like is to execute this part of the program and then stop for more careful debugging (perhaps by single stepping). Debuggers provide commands to set up breakpoints and to execute up to a breakpoint. Another helpful feature that most debuggers provide is the watch facility. By using watches, it is possible to monitor the state (i.e., values) of the variables in the program as the execution progresses.

In the following three sections, we discuss three debuggers and how they are useful in debugging the program `addigits.asm` discussed in Chapter 3. We limit our discussion to 16-bit segments and operands. The program used in our debugging sessions is shown in Program C.57. This program does not use the .STARTUP and .EXIT assembler directives. As explained in Appendix B, we use

```
mov    AX,@DATA
mov    DS,AX
```

in place of the .STARTUP directive and

```
mov    AX,4C00H
int    21H
```

in place of the .EXIT directive.

Program C.57 An example program used to explain debugging

```
TITLE   Add individual digits of a number   ADDIGITS.ASM
COMMENT |
        Objective: To find the sum of individual digits of
                   a given number. Shows character to binary
                   conversion of digits.
            Input: Requests a number from keyboard.
|          Output: Prints the sum of the individual digits.
.MODEL SMALL
.STACK 100H
.DATA
number_prompt  DB  'Please type a number (<11 digits): ',0
```

```
out_msg        DB  'The sum of individual digits is: ',0
number         DB  11 DUP (?)

.CODE
INCLUDE io.mac
main    PROC
        .STARTUP
        PutStr  number_prompt  ; request an input number
        GetStr  number,11    ; read input number as a string
        nwln
        mov     BX,OFFSET number  ; BX := address of number
        sub     DX,DX        ; DX := 0 -- DL keeps the sum
repeat_add:
        mov     AL,[BX]      ; 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     BX           ; increment BX to point to next digit
        jmp     repeat_add   ;  and jump back
done:
        PutStr  out_msg
        PutInt  DX           ; write sum
        nwln
        .EXIT
main    ENDP
        END   main
```

C.2 DEBUG

DEBUG is invoked by

DEBUG file_name

For example, to debug the `addigits` program, we can use

DEBUG addigits.exe

DOS loads DEBUG into memory, which in turn loads `addigits.exe`. It is necessary to enter the extension `.exe`, as DEBUG does not assume any extension. DEBUG displays a hyphen (-) as a prompt. At this prompt, it can accept one of several commands. Table C.1 shows some of the commands useful in debugging programs.

Table C.1 Summary of DEBUG commands

command	function
Display Commands:	
`U`	Unassembles next 32 bytes
`U address`	Unassembles next 32 bytes at `address`
`U range`	Unassembles the bytes in the specified `range`
`D`	Displays the next 128 bytes of memory in hex and ASCII
`D address`	Displays the next 128 bytes of memory at `address`
`D range`	Displays the contents of memory in the specified `range`
`R`	Displays the contents of all registers and shows the next instruction
`R register`	Displays the contents of `register` and accepts hex data to update `register`
`E address`	Displays the contents of the memory location specified by `address`
`E address value-list`	Copies the hex data from `value-list` into memory from `CS:address`
Execution Commands:	
`T`	Traces (i.e., single-step mode) execution; executes one instruction and displays the register contents and the next instruction
`T count`	Executes next `count` instructions
`T =address`	Executes the instruction at `CS:address`
`T =address count`	Executes `count` instructions at `CS:address`
`P`	Like trace but proceeds through `call`, `loop`, `int`
`P count`	Proceeds through the next `count` statements
`P =address`	Executes the statement at `CS:address`
`P =address count`	Executes `count` statements at `CS:address`
`G`	Executes program to completion or until a breakpoint is encountered
`G bkpt-address`	Executes program until the breakpoint specified by `bkpt-address`
`G =address bkpt-address`	Executes program until the breakpoint specified by `bkpt-address` starting from `address`
Miscellaneous Commands:	
`L`	Reloads program after termination
`Q`	Quits DEBUG

Display Group

U (Unassemble)

Unassembles the next 32 bytes. The general format is

```
U [address]
or
U [range]
```

If no address is specified in the command, the next 32 bytes since the last U command are unassembled. If there was no U command, the default address CS:IP is used. The address should be specified in hex. The range can be specified in one of two ways—either by giving a start and end address, or by giving a start address and length in bytes. When specifying length, the prefix L should be used, as shown in the following example.

```
U           ; unassembles the next 32 bytes
U 3B        ; unassembles 32 bytes from CS:3BH
U 3B 4B     ; unassembles from CS:3BH to CS:4BH
U 3B L10    ; unassembles 16 (= 10H) bytes from CS:3BH
```

Note that in the last example, length is specified as L10, where 10H = 16D is the length.

D (Display or Dump)

Displays the contents of the specified memory locations both in hex and ASCII. The general format is similar to that of the U command and is given by

```
D [address]
or
D [range]
```

The default segment is the segment pointed by DS and the default range is 128 (i.e., 80H) bytes.

```
D           ; displays the next 128 bytes from last display
D CS:0      ; displays 128 bytes from CS:0
D 10 17     ; displays from DS:10H to DS:17H
D 3B LB     ; displays 11 (= BH) bytes from DS:3BH
```

E (Enter)

This command can be used to enter data. The general format is

```
E address
or
E address values
```

If the first format is used (i.e., with no values), it displays the contents of the addressed location. The default segment is the data segment pointed by DS. For example,

```
E 12
```

displays the contents of DS:12H. In the second format, the list of values specified replaces the contents of the addressed memory locations. For example,

```
E 46 31 32 33
```

changes the contents of memory locations 46H through 48H to 31H, 32H, and 33H, respectively. We can also do the same with the following command:

```
E 46 '123'
```

The same command can be used to replace machine code. For example,

```
E CS:5 8B D8
```

replaces the machine code by 8BD8, which represents

```
mov    BX,AX
```

R (Register)

Displays the contents of registers and the next instruction. The general format is

```
R
```

or

```
R register
```

If no register is specified, it displays the contents of all registers, including the flags, instruction pointer, and segment registers. The flags register contents are displayed as follows:

Flag	Set	Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary carry	AC	NA
Parity	PE	PO
Carry	CY	NC

When a register name is specified in the command, it displays the contents of the register and prompts (displays ':') for a replacement value. For example,

```
-R AX
AX 0000   ; displays the contents of AX (here 0000)
:7FFF     ; prompts for a replacement value
          ; here we want to write 7FFFH into AX
```

modifies AX to 7FFFH. If you do not want to change the contents of the register, simply type return. You can also use this command to modify the IP register.

Execution Group

T (Trace)

Executes the program in single-step mode; after the execution, it displays the contents of the registers and the next instruction. The general format is

```
T   or   T count    or
T =address    or    T =address count
```

If a count value is specified, it traces count instructions. It displays contents of registers and the next instruction after the execution of each instruction. If an address is specified, tracing starts at the specified address. Here are some examples:

```
T =5D     ; trace the instruction at CS:5DH
T 3       ; trace the next 3 instructions
T =5D 3   ; trace 3 instructions from CS:5DH
```

P (Proceed)

Similar to trace except it considers an interrupt call (`int`), procedure call (`call`), loop, etc., as single instructions. Normally you use this command unless you want to debug a procedure, interrupt routine, etc.

G (Go)

Executes program to a specified breakpoint. The format is

```
G   or   G bkpt-address   or
G =address bkpt-address
```

This command is useful in setting breakpoints. You can specify up to 10 breakpoint addresses. If the optional start address (`=address`) is given, execution begins from this address. This, for example, is useful in debugging a procedure or a part of the program, without executing it from the beginning. Some examples are given below:

```
G          ; execute program to completion
G 31       ; execute up to CS:31H
G =31 45   ; execute from CS:31H to CS:45H
```

C.2.1 Miscellaneous Group

The other two commands in Table C.1 are useful to reload the program (L) and exit the DEBUG (Q).

C.2.2 An Example

A sample DEBUG run on `addigits.exe` is shown in Program C.58. The U command on line 2 displays the code by unassembling the first 32 bytes. A drawback with this is that there is no symbolic information. For example,

```
mov    AX,@DATA
```

is displayed as

```
mov    AX,3F09
```

where 3F09 (in hex) is the data segment value. Similarly, procedure calls include the offset values but not the procedure names. This deficiency is remedied by the other two debuggers.

Notice that the code shown here does not exactly correspond to the code of Program C.57. The reason is that each macro call (such as PutStr, GetStr, and nwln) is expanded by using the macro definitions in io.mac. For example, the PutStr macro call is expanded by the four lines of code (lines 5–8). Using symbolic information, we can write these four lines of code as

```
push   AX
mov    AX,OFFSET number_prompt
call   proc_PutStr
pop    AX
```

As discussed in Appendix B, these macros are defined in io.mac. The GetStr macro is expanded to lines 9–15 and nwln to lines 16–18.

Now let us examine the data segment contents. In order to use the default DS register, we have to set up this register to point to our data segment. This is done by the first two lines of the code. One way to execute these two lines of code is to use the T command (line 20). It makes no difference whether you use the P or T command, as there are no procedure calls or loop instructions. Note that the trace command executes in single-step mode. Thus, after executing each instruction, it displays the contents of the registers, status of the flags, and the next instruction to be executed. From line 27, we can see that DS is initialized to the data segment.

Now we can use the D command (line 29) to display the first 128 bytes starting at offset 0. The data segment contains the two message strings:

```
Please type a number (<10 digits):
The sum of individual digits is:
```

and the storage space for number starts after these two message strings at 3F09:0046. Since we have not initialized it, the contents do not matter at this point.

Now let us execute the program until after reading an input number. That is, we will set up a breakpoint at the instruction

```
mov    BX,0046
```

at offset 001FH. We can do this by using the G command on line 38. The prompt and the input number are shown on line 39. At the breakpoint, it displays the contents of the registers, flags, and the next instructions, as in the trace command we have seen before. While the G command allows us to set up breakpoints in the program, the other two symbolic debuggers provide a much better screen-oriented user interface, as we shall see later in this appendix.

Now let us verify that the input has been read properly. We use the D command

```
D 46 LB
```

on line 44 to examine the contents of number. In this D command, we are not only specifying the address (46H), but also indicating that 11 (=BH) bytes are to be displayed. Thus, we will just see the contents (11 bytes) of number (lines 45 and 46).

Let us suppose that we want to check the logic of the loop (lines 25–32 in Program C.57). We can do this by executing the loop in single-step mode using the T command on line 47. This gives us an opportunity to check the logic one instruction at a time. An interesting point is that, on line 55, when we are using the indirect addressing mode, it displays the address and its contents:

```
DS:0046=31
```

At the end of the loop, DX = 1, which is what it should be for the given input.

Having checked the logic of the loop, let us run the whole loop without any interruption. This is done by setting a breakpoint using the G command on line 80. (In this example, it is useful to have the list file handy to know the offset values of the code at various points.) This breakpoint is set at line 34 in Program C.57. We note that the sum in the DX register is the correct value (2DH = 45D) for the input given in this sample run.

The rest of the DEBUG output is straightforward to follow. Notice that after the program has terminated, we have used the L command to reload the application for another execution, this time without any breakpoints. Finally, on line 100, we have used the Q command to exit DEBUG.

Program C.58 A sample DEBUG session

```
 1:  A:\>debug addigits.exe
 2:  -U
 3:  3ED1:0000 B8093F        MOV AX,3F09
 4:  3ED1:0003 8ED8          MOV DS,AX
 5:  3ED1:0005 50            PUSH AX
 6:  3ED1:0006 B80000        MOV AX,0000
 7:  3ED1:0009 E85600        CALL 0062
 8:  3ED1:000C 58            POP AX
 9:  3ED1:000D 51            PUSH CX
10:  3ED1:000E B90B00        MOV CX,000B
11:  3ED1:0011 50            PUSH AX
12:  3ED1:0012 B84600        MOV AX,0046
13:  3ED1:0015 E88101        CALL 0199
14:  3ED1:0018 58            POP AX
15:  3ED1:0019 59            POP CX
16:  3ED1:001A 50            PUSH AX
17:  3ED1:001B E83500        CALL 0053
```

```
3ED1:001E 58            POP AX
3ED1:001F BB4600        MOV BX,0046
-T 2

AX=3F09  BX=0000  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3EC1  ES=3EC1  SS=3F20  CS=3ED1  IP=0003   NV UP EI PL NZ NA PO NC
3ED1:0003 8ED8          MOV DS,AX

AX=3F09  BX=0000  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0005   NV UP EI PL NZ NA PO NC
3ED1:0005 50            PUSH AX
-D 0
3F09:0000  50 6C 65 61 73 65 20 74-79 70 65 20 61 20 6E 75   Please type a nu
3F09:0010  6D 62 65 72 20 28 3C 31-30 20 64 69 67 69 74 73   mber (<10 digits
3F09:0020  29 3A 20 00 54 68 65 20-73 75 6D 20 6F 66 20 69   ): .The sum of i
3F09:0030  6E 64 69 76 69 64 75 61-6C 20 64 69 67 69 74 73   ndividual digits
3F09:0040  20 69 73 3A 20 00 00 00-00 00 00 00 00 00 00 00    is: ..........
3F09:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
3F09:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
3F09:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-G 1F
Please type a number (<10 digits): 1234567890

AX=3F09  BX=0000  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=001F   NV UP EI PL NZ NA PO NC
3ED1:001F BB4600        MOV BX,0046
-D 46 LB
3F09:0040                    31 32-33 34 35 36 37 38 39 30         1234567890
3F09:0050  00
-T 8

AX=3F09  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0022   NV UP EI PL NZ NA PO NC
3ED1:0022 2BD2          SUB DX,DX

AX=3F09  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0024   NV UP EI PL ZR NA PE NC
3ED1:0024 8A07          MOV AL,[BX]                          DS:0046=31

AX=3F31  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0026   NV UP EI PL ZR NA PE NC
3ED1:0026 3C00          CMP AL,00

AX=3F31  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0028   NV UP EI PL NZ NA PO NC
3ED1:0028 7407          JZ 0031

AX=3F31  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002A   NV UP EI PL NZ NA PO NC
3ED1:002A 240F          AND AL,0F

AX=3F01  BX=0046  CX=04EC  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002C   NV UP EI PL NZ NA PO NC
```

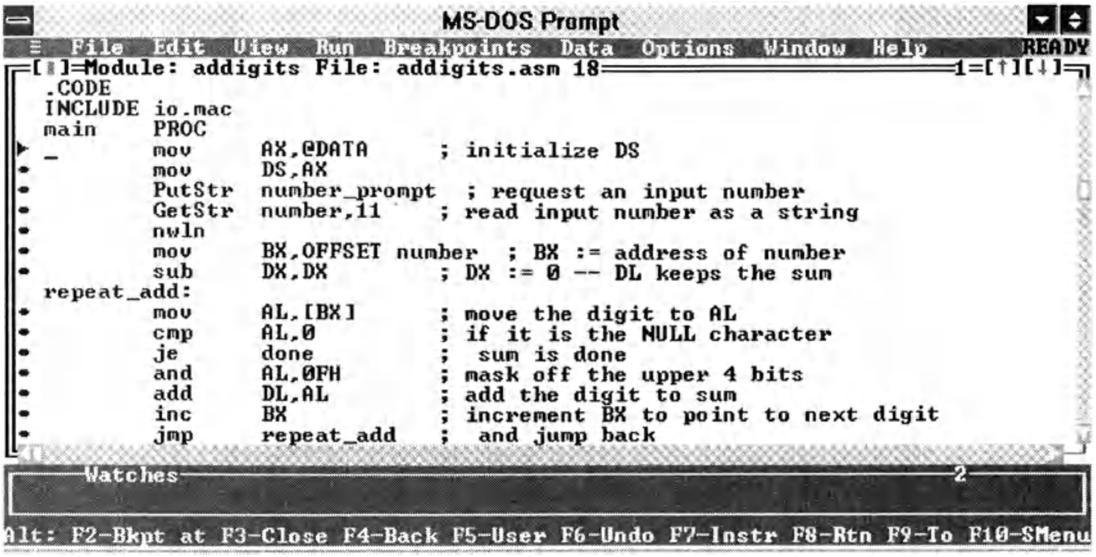
```
3ED1:002C 02D0        ADD DL,AL

AX=3F01  BX=0046  CX=04EC  DX=0001  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002E   NV UP EI PL NZ NA PO NC
3ED1:002E 43          INC BX

AX=3F01  BX=0047  CX=04EC  DX=0001  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=002F   NV UP EI PL NZ NA PE NC
3ED1:002F EBF3        JMP 0024
-G 31

AX=3F00  BX=0050  CX=04EC  DX=002D  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0031   NV UP EI PL ZR NA PE NC
3ED1:0031 50          PUSH AX
-G45
The sum of individual digits is: 45

AX=3F00  BX=0050  CX=04EC  DX=002D  SP=0100  BP=0000  SI=0000  DI=0000
DS=3F09  ES=3EC1  SS=3F20  CS=3ED1  IP=0045   NV UP EI PL NZ NA PO NC
3ED1:0045 B8004C      MOV AX,4C00
-G

Program terminated normally
-L
-G
Please type a number (<10 digits): 456
The sum of individual digits is: 15

Program terminated normally
-Q

A:\>
```

C.3 Turbo Debugger TD

Turbo Debugger is a window-oriented debugger that facilitates symbolic debugging at the source-code level. TD can be used to debug programs written in high-level languages like C and Pascal as well as in assembly language. In this section, we briefly discuss some of the features of TD relevant to debugging assembly language programs.

In order for TD to use symbolic information during debugging, you have to assemble your program with the ZI option and link with the V option. For example, to debug `addigits.asm`, use the following commands to prepare your program:

```
TASM /zi addigits
TLINK /v addigits io
```


MS-DOS Prompt
≡ File Edit View Run Breakpoints Data Options Window Help READY
Watches
2
Alt: F2-Bkpt at F3-Close F4-Back F5-User F6-Undo F7-Instr F8-Rtn F9-To F10-SMenu

```
[■]=Module: addigits File: addigits.asm 18                          1=[↑][↓]
 .CODE
 INCLUDE io.mac
 main      PROC
▶ _        mov     AX,@DATA       ; initialize DS
•          mov     DS,AX
•          PutStr  number_prompt  ; request an input number
•          GetStr  number,11      ; read input number as a string
•          nwln
•          mov     BX,OFFSET number  ; BX := address of number
•          sub     DX,DX          ; DX := 0 -- DL keeps the sum
 repeat_add:
•          mov     AL,[BX]        ; 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     BX             ; increment BX to point to next digit
•          jmp     repeat_add     ;  and jump back
```

Figure C.1 TD window at the start of `addigits.asm` program.

The Turbo Debugger can then be invoked by

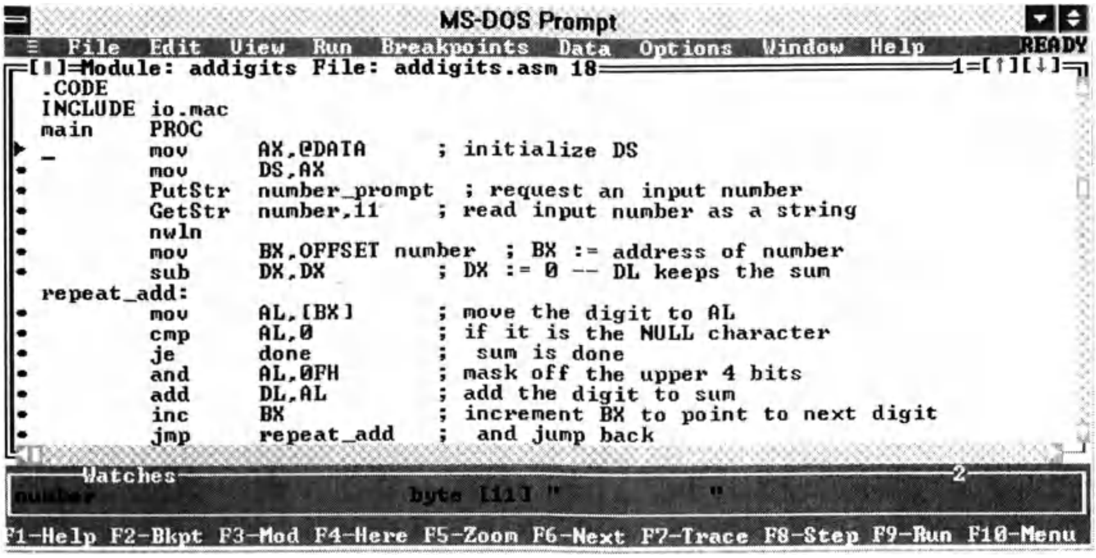
```
TD addigits
```

Figure C.1 shows the screen that you would see after invoking TD as indicated. The screen consists of a menu bar (called main menu) at the top, and a quick reference help line at the bottom. In addition, it displays two windows: a module window and a watches window. Each window that the TD opens has a number associated with it. The window number appears in the upper-righthand corner of the window. For example, the module window is window 1 and the watches window is window 2. The active window, the module window in Figure C.1, has a double-line border around it and inactive windows have a single-line border (e.g., see watches window).

The module window shows the source program. The arrow at the left points to the next instruction to be executed. Since we haven't yet run the program, the arrow points to the first line of the `main` procedure in Figure C.1.

You can make an inactive window active by pressing Alt-x, where x is the window number. For example, Alt-2 makes the watches window active.

The main menu can be activated by F10. Press carriage return to open the selected pull-down menu. You can then use the arrow keys to navigate the menu items. Here we will take a brief look at the `View` and `Run` menu options.


MS-DOS Prompt
≡ File Edit View Run Breakpoints Data Options Window Help READY
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
[■]=Module: addigits File: addigits.asm 18                    1=[↑][↓]
 .CODE
 INCLUDE io.mac
 main    PROC
          mov     AX,@DATA       ; initialize DS
          mov     DS,AX
          PutStr  number_prompt  ; request an input number
          GetStr  number,11      ; read input number as a string
          nwln
          mov     BX,OFFSET number  ; BX := address of number
          sub     DX,DX          ; DX := 0 -- DL keeps the sum
 repeat_add:
          mov     AL,[BX]        ; 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     BX             ; increment BX to point to next digit
          jmp     repeat_add     ;  and jump back
Watches                                                        2
number                   byte [11] "          "
```

Figure C.2 TD window after adding `number` to watch list.

The `View` pull-down menu provides several options to view the status of the program. Some of the options available are listed in Table C.2.

As we indicated in Section C.1, watches are useful to monitor the state of a set of variables as the program execution progresses. In Turbo Debugger, a variable or an expression can be added to the watch list by using `add watch` in the Data menu (Ctrl-F7). Figure C.2 shows the watches window when the variable `number` is added to the watch list. Notice that the TD shows the name of the variable, its type and contents. Now, for example, we can test the initial part of our program by placing a breakpoint after reading the input number by `GetStr`. This can be done by using an option under the `Run` menu, which we will discuss next.

Program execution is controlled by the `Run` menu. Some of the options in this menu are shown in Table C.3. Now let us execute the program until

```
mov     BX,OFFSET number
```

One way is to move the cursor to this line and press F4. This execution prompts you for a number (we have given 1234567890 as input in this example execution) that is stored in variable `number`. As shown in Figure C.3, the watches window shows that `number` has properly received the input value. Breakpoints in a program can also be set by the `Breakpoints` menu.

Table C.2 Selected `View` menu options

Breakpoints	Displays a list of breakpoints set in the program
Stack	Displays the active procedures
Watches	Displays the values of the variables and expressions in the watch list
Variables	Shows the names and values of all variables accessible from current location of the program
CPU	Shows the status of the program (discussed in text)
Dump	Shows the contents of a part of memory (similar to DEBUG's Dump command)
Register	Shows the contents of all registers including the flags

Table C.3 Selected `Run` menu options

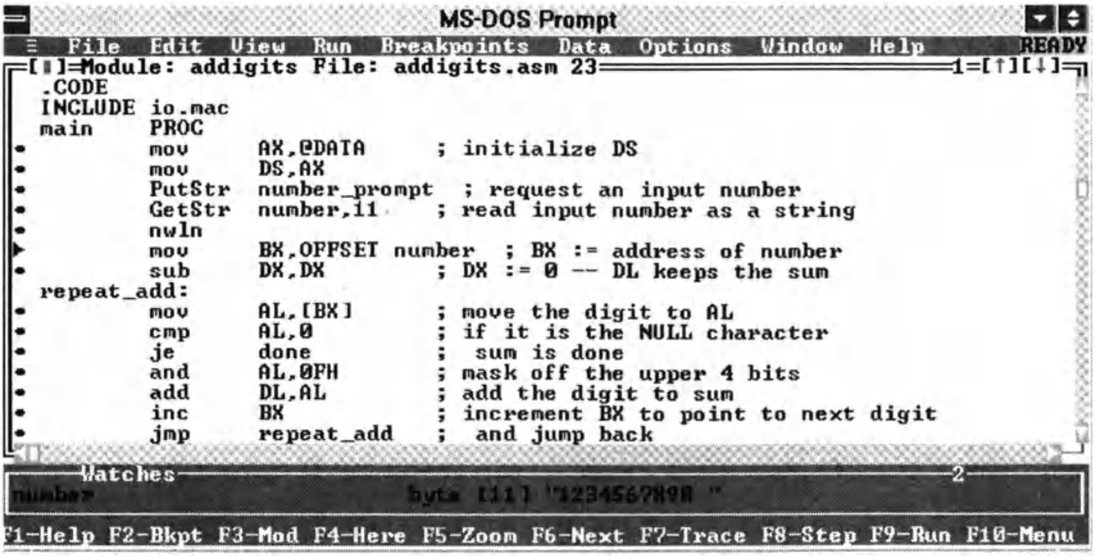
Run (F9)	Execute program until completion or until a breakpoint is encountered
Goto cursor (F4)	Execute program up to the line that the cursor is on
Trace into (F7)	Execute one instruction at a time in single-step mode (similar to DEBUG Trace command)
Step over (F8)	Execute one statement at a time (a procedure call, interrupt, loop are treated as a single statement as in the Proceed command of DEBUG)

The module window is useful in debugging programs at the source-code level. This is particularly helpful in debugging programs written in high-level languages like C and Pascal. While the source-code level of debugging is also useful in debugging assembly language programs (for example, we can set convenient watches to monitor the progress), the CPU window is much more useful for low-level debugging. The remainder of the section focuses on the CPU window.

The CPU window provides a snapshot view of the program state. The CPU window after executing the program until

```
mov     BX,OFFSET number
```

is shown in Figure C.4. The window is divided into five panes. The code pane (top left pane) shows the CS:IP, along with the machine code and source-code


MS-DOS Prompt
≡ File Edit View Run Breakpoints Data Options Window Help READY
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
[■]=Module: addigits File: addigits.asm 23                    1=[↑][↓]
 .CODE
 INCLUDE io.mac
 main    PROC
•        mov     AX,@DATA      ; initialize DS
•        mov     DS,AX
•        PutStr  number_prompt  ; request an input number
•        GetStr  number,11     ; read input number as a string
•        nwln
▶        mov     BX,OFFSET number  ; BX := address of number
•        sub     DX,DX         ; DX := 0 -- DL keeps the sum
 repeat_add:
•        mov     AL,[BX]       ; 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     BX            ; increment BX to point to next digit
•        jmp     repeat_add    ;  and jump back
```

```
Watches                                                   2
number                     byte [11] "1234567890 "
```

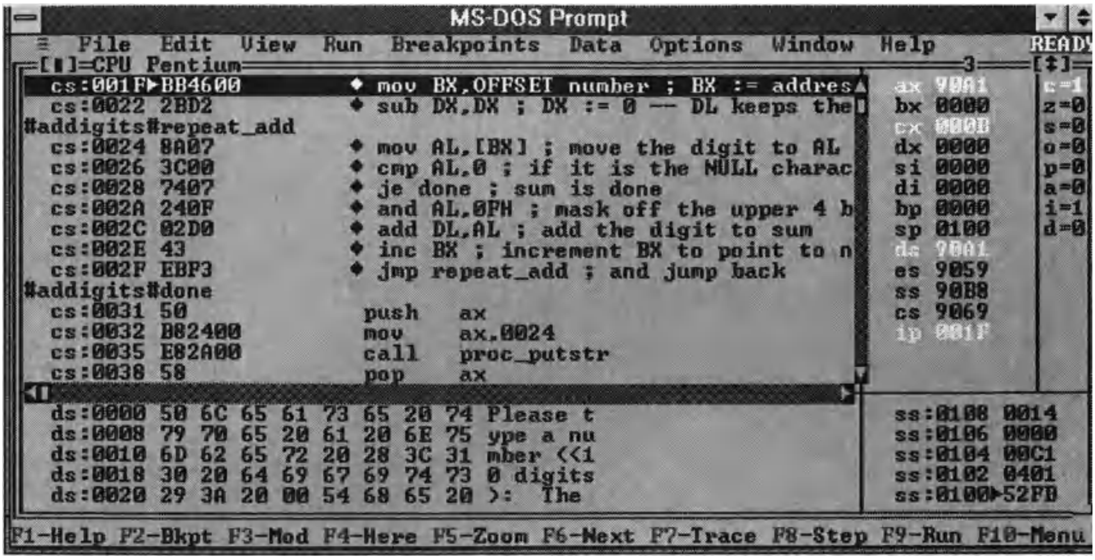
Figure C.3 TD window after reading the value 1234567890 into `number`.

lines. The current instruction is indicated by the arrow and also by highlighting the line if the code pane is active.

The next pane is the register pane and shows the contents of all 16-bit registers except flags. (You can display 32-bit registers by using an option in the local menu.) The status of flags is shown in the top right pane (flags pane). Unlike the DEBUG, the flag values are shown as 1 or 0 to indicate whether the flag is set or cleared, respectively. Also, changes in register values and flags are highlighted. For example, see registers AX, CX, and the carry flag.

The bottom left pane (data pane) shows the contents of the data segment. As shown in Figure C.4, the data pane shows the contents both in hex and ASCII. The fifth pane (stack pane) shows SS:SP and the contents of the stack. The top of the stack is indicated by an arrow. Remember that the stack grows toward low memory addresses. Therefore, SP values are displayed in decreasing order from top to bottom. You can use the tab to move the cursor from one pane to the next.

An interesting feature of TD is its context-sensitive local menus. Depending on where the cursor is, a local pop-up menu can be activated by Alt-F10 or Ctrl-F10. Figure C.5 shows the pop-up local menu of the data pane. For example, we can use the option


MS-DOS Prompt
≡ File Edit View Run Breakpoints Data Options Window Help READY
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
=[■]=CPU Pentium==========================================3======[↕]=
  cs:001F►BB4600          ♦ mov BX,OFFSET number ; BX := addres   ax 90A1   c=1
  cs:0022 2BD2            ♦ sub DX,DX ; DX := 0 -- DL keeps the   bx 0000   z=0
#addigits#repeat_add                                              cx 000B   s=0
  cs:0024 8A07            ♦ mov AL,[BX] ; move the digit to AL    dx 0000   o=0
  cs:0026 3C00            ♦ cmp AL,0 ; if it is the NULL charac   si 0000   p=0
  cs:0028 7407            ♦ je done ; sum is done                 di 0000   a=0
  cs:002A 240F            ♦ and AL,0FH ; mask off the upper 4 b   bp 0000   i=1
  cs:002C 02D0            ♦ add DL,AL ; add the digit to sum      sp 0100   d=0
  cs:002E 43              ♦ inc BX ; increment BX to point to n   ds 90A1
  cs:002F EBF3            ♦ jmp repeat_add ; and jump back        es 9059
#addigits#done                                                    ss 90B8
  cs:0031 50                push    ax                            cs 9069
  cs:0032 B82400            mov     ax,0024                       ip 001F
  cs:0035 E82A00            call    proc_putstr
  cs:0038 58                pop     ax
  ds:0000 50 6C 65 61 73 65 20 74 Please t                        ss:0108 0014
  ds:0008 79 70 65 20 61 20 6E 75 ype a nu                        ss:0106 0000
  ds:0010 6D 62 65 72 20 28 3C 31 mber (<1                        ss:0104 00C1
  ds:0018 30 20 64 69 67 69 74 73 0 digits                        ss:0102 0401
  ds:0020 29 3A 20 00 54 68 65 20 ):  The                         ss:0100►52FB
```

Figure C.4 CPU window before the `repeat_add` loop.

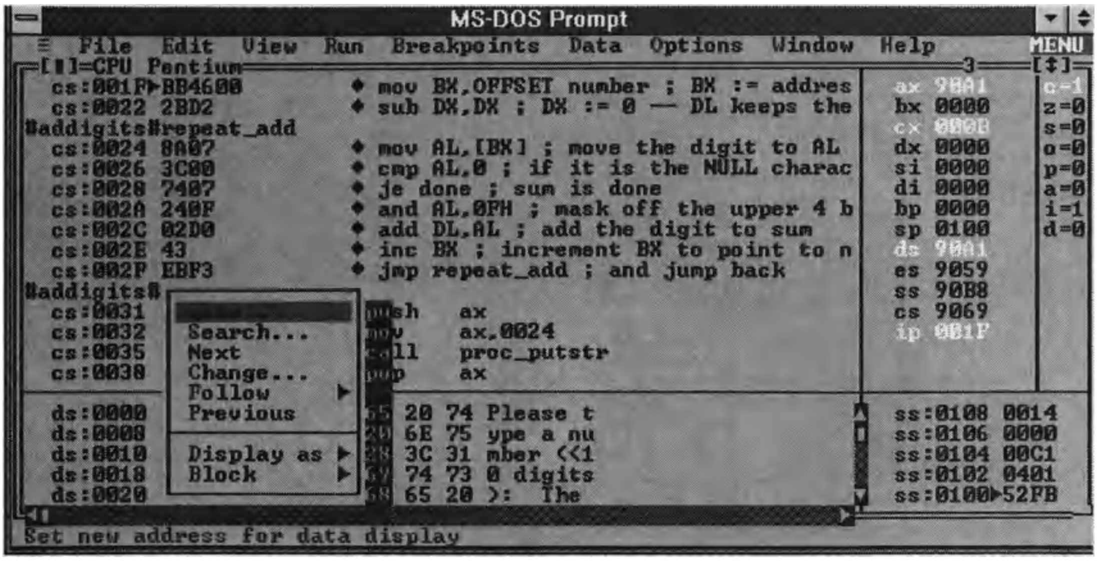
```
Goto...
```

to specify an address to change the area of a data segment memory to be displayed. If we want to see the contents of `number` (whose offset is 46H), we can use this option of the local menu. The resulting data pane is shown in Figure C.6. As you can see, it shows the input number that we have given to the program. You also see another similar sequence starting at DS:0064. This is actually the buffer into which `GetStr` reads the input number first before copying it into `number`.

Now if you want to check the logic of the `repeat_add` loop, you can use Trace Into (F7) or Step Over (F8) to single step while monitoring the contents of the registers and flags. Since there are no procedure calls or loop instructions, both F7 and F8 behave the same way for our example program. To see the complete execution of `repeat_add` loop, move the cursor to the

```
push    AX
```

instruction at CS:0031 and press F4. The resulting state is shown in Figure C.7. Now notice that the sum in the DX register is 2DH, which is the hexadecimal equivalent of 45D.



```
[▮]=CPU Pentium                                                  3  [↕]
  cs:001F▶BB4600     ◆ mov BX,OFFSET number ; BX := addres    ax 90A1   c=1
  cs:0022 2BD2       ◆ sub DX,DX ; DX := 0 — DL keeps the     bx 0000   z=0
#addigits#repeat_add                                          cx 000B   s=0
  cs:0024 8A07       ◆ mov AL,[BX] ; move the digit to AL     dx 0000   o=0
  cs:0026 3C00       ◆ cmp AL,0 ; if it is the NULL charac    si 0000   p=0
  cs:0028 7407       ◆ je done ; sum is done                  di 0000   a=0
  cs:002A 240F       ◆ and AL,0FH ; mask off the upper 4 b    bp 0000   i=1
  cs:002C 02D0       ◆ add DL,AL ; add the digit to sum       sp 0100   d=0
  cs:002E 43         ◆ inc BX ; increment BX to point to n    ds 90A1
  cs:002F EBF3       ◆ jmp repeat_add ; and jump back         es 9059
#addigits#                                                    ss 90B8
  cs:0031                 push   ax                           cs 9069
  cs:0032                 mov    ax,0024                      ip 001F
  cs:0035                 call   proc_putstr
  cs:0038                 pop    ax

  ds:0000              55 20 74 Please t                      ss:0108 0014
  ds:0008              20 6E 75 ype a nu                      ss:0106 0000
  ds:0010              28 3C 31 mber <<1                      ss:0104 00C1
  ds:0018              37 74 73 0 digits                      ss:0102 0401
  ds:0020              68 65 20 ): The                        ss:0100▶52FB
Set new address for data display
```

Figure C.5 CPU window with Data Pane local menu.

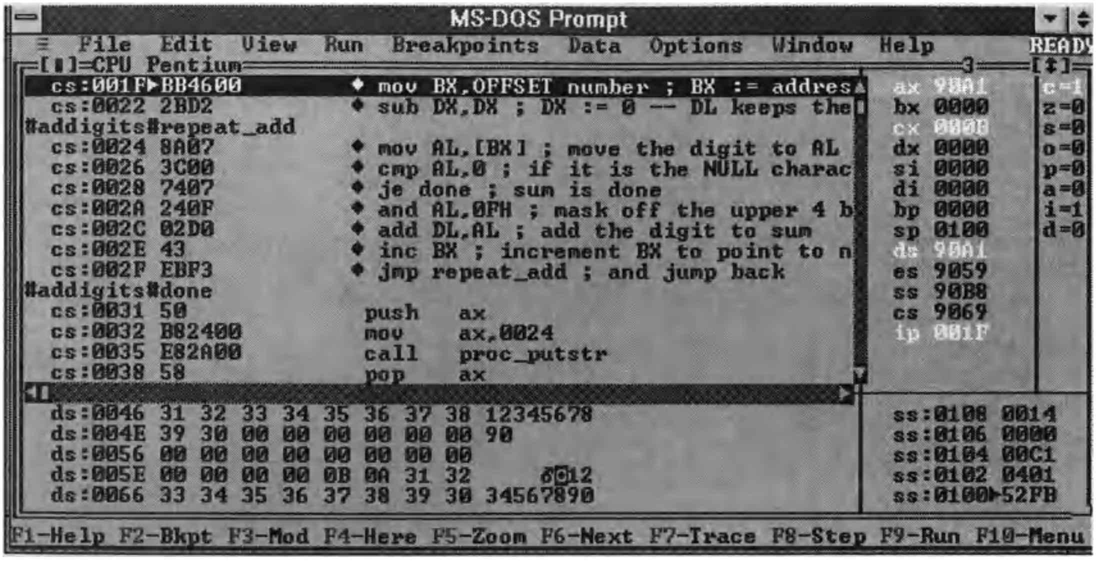
In this brief discussion, we have glossed over numerous features available in TD. Now it is up to you to fully utilize the help offered by TD in debugging your assembly language programs.

C.4 CodeView

Microsoft's CodeView is similar in spirit to the Turbo Debugger. As in TD, you have to assemble your program using the ZI option and link with the CO option. This causes the symbolic information to be placed in the execution file.

Depending on the version of CodeView you are using, some of the details might vary. Here we briefly discuss some generic features.

As in TD, you can add a variable or an expression to the watch list. The values of variables in the watch list are displayed in the watch window. The watch menu can be used to either add or delete an expression or a variable to the watch list. Also, breakpoints can be set or edited, i.e., added, deleted, etc. Go (F5) can be used to execute from the next instruction to the completion of the program or until a breakpoint is encountered.


MS-DOS Prompt
File Edit View Run Breakpoints Data Options Window Help
READY
[▮]=CPU Pentium
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
  cs:001F►BB4600         ♦ mov BX,OFFSET number ; BX := addres   ax 90A1   c=1
  cs:0022 2BD2           ♦ sub DX,DX ; DX := 0 -- DL keeps the   bx 0000   z=0
#addigits#repeat_add                                             cx 000B   s=0
  cs:0024 8A07           ♦ mov AL,[BX] ; move the digit to AL    dx 0000   o=0
  cs:0026 3C00           ♦ cmp AL,0 ; if it is the NULL charac   si 0000   p=0
  cs:0028 7407           ♦ je done ; sum is done                 di 0000   a=0
  cs:002A 240F           ♦ and AL,0FH ; mask off the upper 4 b   bp 0000   i=1
  cs:002C 02D0           ♦ add DL,AL ; add the digit to sum      sp 0100   d=0
  cs:002E 43             ♦ inc BX ; increment BX to point to n   ds 90A1
  cs:002F EBF3           ♦ jmp repeat_add ; and jump back        es 9059
#addigits#done                                                   ss 90B8
  cs:0031 50              push   ax                              cs 9069
  cs:0032 B82400          mov    ax,0024                         ip 001F
  cs:0035 E82A00          call   proc_putstr
  cs:0038 58              pop    ax

  ds:0046 31 32 33 34 35 36 37 38 12345678                       ss:0108 0014
  ds:004E 39 30 00 00 00 00 00 00 90                             ss:0106 0000
  ds:0056 00 00 00 00 00 00 00 00                                ss:0104 00C1
  ds:005E 00 00 00 00 0B 0A 31 32     ♂◙12                       ss:0102 0401
  ds:0066 33 34 35 36 37 38 39 30 34567890                       ss:0100►52FB
```

Figure C.6 CPU window after executing `Goto...` command.

Trace (F8) and step (F10) commands are also available to control program execution. These are similar to trace into and step over commands available in Turbo Debugger.

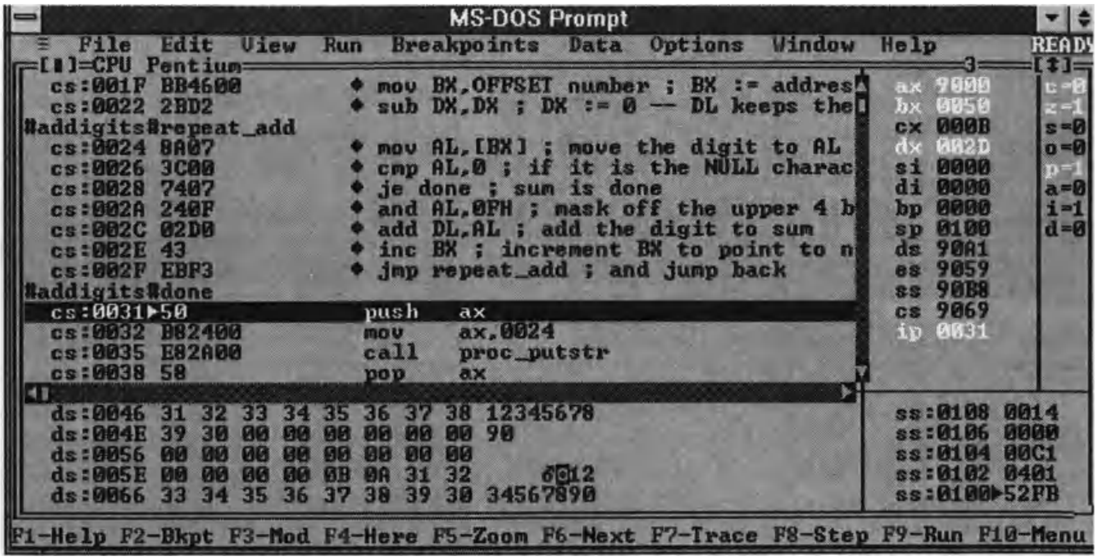
The register window displays the contents of all registers including the flags register. The flags values are reported using the two-letter encoding used in DEBUG (see page 590).

The `view` menu provides several options to open other windows. For example, the `memory` option under this menu can be used to open the memory window and the `output` option switches to the program output window.

C.5 Summary

We started this appendix with a brief discussion of the basic debugging techniques. Since assembly language is a low-level programming language, debugging tends to be even more tedious than debugging a program written in a high-level language. It is, therefore, imperative to follow good programming practices in order to help debug and maintain assembly language programs.

There are several tools available for debugging programs. We discussed three debuggers—DEBUG, Turbo Debugger, and CodeView—in this appendix.


MS-DOS Prompt
≡ File Edit View Run Breakpoints Data Options Window Help
READY
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
=[■]=CPU Pentium=====================================================3=====[↕]=
  cs:001F BB4600        ♦ mov BX,OFFSET number ; BX := addres   ax 9000   c=0
  cs:0022 2BD2          ♦ sub DX,DX ; DX := 0 — DL keeps the    bx 0050   z=1
#addigits#repeat_add                                            cx 000B   s=0
  cs:0024 8A07          ♦ mov AL,[BX] ; move the digit to AL    dx 002D   o=0
  cs:0026 3C00          ♦ cmp AL,0 ; if it is the NULL charac   si 0000   p=1
  cs:0028 7407          ♦ je done ; sum is done                 di 0000   a=0
  cs:002A 240F          ♦ and AL,0FH ; mask off the upper 4 b   bp 0000   i=1
  cs:002C 02D0          ♦ add DL,AL ; add the digit to sum      sp 0100   d=0
  cs:002E 43            ♦ inc BX ; increment BX to point to n   ds 90A1
  cs:002F EBF3          ♦ jmp repeat_add ; and jump back        es 9059
#addigits#done                                                  ss 90B8
  cs:0031►50             push   ax                              cs 9069
  cs:0032 B82400         mov    ax,0024                         ip 0031
  cs:0035 E82A00         call   proc_putstr
  cs:0038 58             pop    ax
  ds:0046 31 32 33 34 35 36 37 38 12345678                      ss:0108 0014
  ds:004E 39 30 00 00 00 00 00 00 90                            ss:0106 0000
  ds:0056 00 00 00 00 00 00 00 00                               ss:0104 00C1
  ds:005E 00 00 00 00 0B 0A 31 32     ♂◙12                      ss:0102 0401
  ds:0066 33 34 35 36 37 38 39 30 34567890                      ss:0100►52FB
```

Figure C.7 CPU window after completing the `repeat_add` loop.

While DEBUG is a line-oriented debugger, the other two are window-oriented and offer a much better user interface. The best way to learn to use these debuggers is by hands-on experience.

C.6 Exercises

C–1 Discuss some general techniques useful in debugging programs.

C–2 How are window-oriented debuggers like Turbo Debugger better than line-oriented debuggers like DEBUG?

C–3 What is the difference between T and P commands of DEBUG?

C–4 Discuss how breakpoints are useful in debugging programs.

C–5 It has been said that the CPU window of the Turbo Debugger is more useful in debugging assembly language programs. Explain the reasons for this.

C.7 Progamming Exercises

C–P1 Take a program from Chapter 3 and ask your friend to deliberately introduce some logical errors into the program. Then use your debugger to locate and fix errors. Discuss the features of your debugger that you found most useful.

C–P2 Using your debugger's capability to modify flags, verify the conditions mentioned for conditional jumps in Section 7.3 on page 263.

Appendix D

Pentium Instruction Set

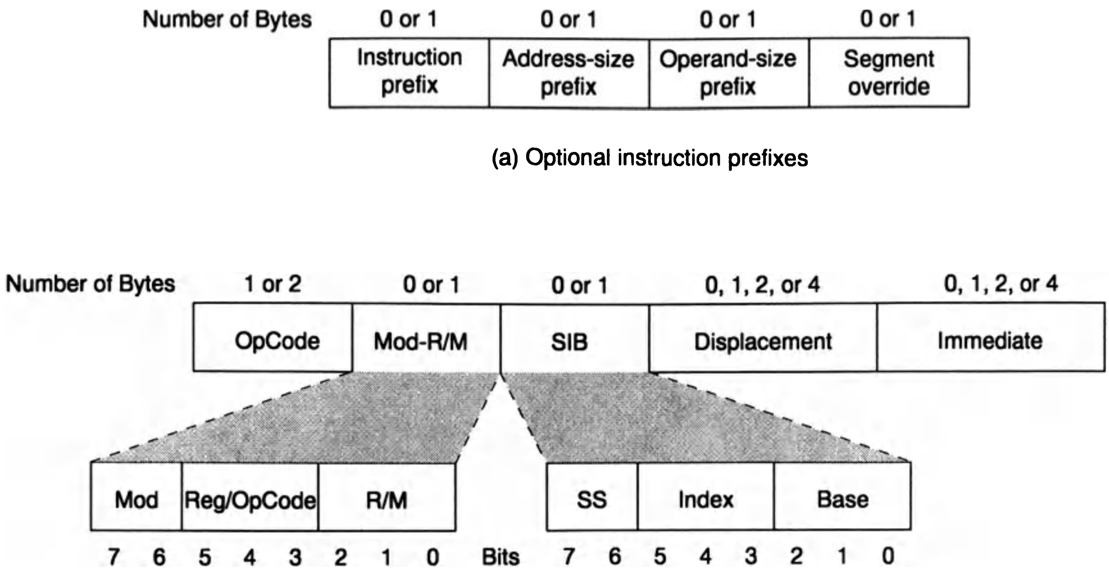
Objectives

- To describe the Pentium instruction format
- To present selected Pentium instructions

Instruction format and encoding encompass a variety of factors: addressing modes, number of operands, number of registers, sources of operands, etc. Instructions can be of fixed length or variable length. In a fixed-length instruction set, all instructions are of the same length. In processors like Pentium that use variable-length instructions, instruction length can vary to accommodate the complexity of an instruction. Section D.1 discusses the instruction format of the Pentium processor. A subset of the Pentium instruction set is presented in Section D.2.

D.1 Pentium Instruction Format

Pentium uses variable-length instructions. Instruction length can be between 1 and 16 bytes. The instruction format of Pentium is shown in Figure D.1. The general instruction format is shown in Figure D.1*b*. In addition, instructions can have several optional instruction prefixes shown in Figure D.1*a*. The next two subsections discuss the instruction format in detail.



(a) Optional instruction prefixes

(b) General instruction format

Figure D.1 Pentium instruction format.

D.1.1 Instruction Prefixes

There are four instruction prefixes, as shown in Figure D.1*a*. These prefixes can appear in any order. All four prefixes are optional. When a prefix is present, it takes a byte.

- *Instruction Prefixes*: Instruction prefixes such as `rep` were discussed in Chapter 9. This group of prefixes consists of `rep`, `repe/repz`, `repne/repnz`, and `lock`. The three repeat prefixes were discussed in detail in Chapter 9. The `lock` prefix is useful in multiprocessor systems to ensure exclusive use of shared memory.
- *Segment Override Prefixes*: These prefixes are used to override the default segment association. For example, DS is the default segment for accessing data. We can override this by using a segment prefix. We saw an example of this in Chapter 4 (see Program 4.14 on page 151). 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 2, 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 one 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.

D.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    AX
push    CX
push    DX
push    BX
```

  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 (AX, CX, DX, or BX).

- *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 D.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 D.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   BX
done:
```

generates the code 7F 01 for the `jg` conditional jump instruction. The opcode for `jg` is 7FH and the displacement is 01 because the pop instruction encoding takes only a single byte.

- *Immediate*: The immediate field is 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     AX,256
```

is encoded as `B8 0100`. Note that the first bye B8 not only identifies the instruction as `mov` but also specifies the destination register as AX (by the least significant three bits of the opcode byte). Pentium uses the following encoding for the 16-bit registers:

AX = 0
CX = 1
DX = 2
BX = 3
SP = 4
BP = 5
SI = 6
DI = 7

The last two bytes represent the immediate value 256, which is equal to 100H. If we change the register from AX to BX, the opcode byte changes

D.2 Selected Pentium Instructions

This section gives selected Pentium instructions in alphabetical order. For each instruction, the instruction mnemonic, flags affected, format, and a description are given. For a more detailed discussion, please refer to the *Pentium Processor Family Developer's Manual—Volume 3: Architecture and Programming Manual.* While most of the components are self-explanatory, the flags section requires some explanation regarding the notation used. An instruction can affect a flag bit in one of several ways. We use the following notation to represent the effect of an instruction on a flag bit.

0	—	Cleared
1	—	Set
–	—	Unchanged
M	—	Updated according to the result
*	—	Undefined

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 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 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 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.

bts — Bit test and set

C	O	Z	S	P	A
M	–	–	–	–	–

Format: `bts src1,src2`

Description: The value of the bit in `src1`, whose position is indicated by `src2`, is saved in the carry flag and then the bit in `src1` is set (i.e., stores 1). The first operand `src1` can be a 16- or 32-bit value that is either in a register or memory. The second operand `src2` can be a 16- or 32-bit value located in a register or an 8-bit immediate value. Clock cycles: 7–13.

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 4. For details on other forms of call, see the Pentium data book. Clock cycles: vary depending on the type of call.

cbw — Convert byte to word

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `cbw`

Description: Converts the signed byte in AL to a signed word in AX by copying the sign bit of AL (the most significant bit) to all bits of AH. Clock cycles: 3.

cdq — Convert doubleword to quadword

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `cdq`

Description: Converts the signed doubleword in EAX to a signed quadword in EDX:EAX by copying the sign bit of EAX (the most significant bit) to all bits of EDX. Clock cycles: 2.

clc — Clear carry flag

C	O	Z	S	P	A
0	–	–	–	–	–

Format: `clc`

Description: Clears the carry flag. Clock cycles: 2.

cld — Clear direction flag

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `cld`

Description: Clears the direction flag. Clock cycles: 2.

cli — Clear interrupt flag

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `cli`

Description: Clears the interrupt flag. Note that maskable interrupts are disabled when the interrupt flag is cleared. Clock cycles: 7.

cmc — Complement carry flag

C	O	Z	S	P	A
M	–	–	–	–	–

Format: `cmc`

Description: Complements the carry flag. Clock cycles: 2.

cmp — Compare two operands

C	O	Z	S	P	A
M	M	M	M	M	M

Format: `cmp    dest,src`

Description: Compares the two operands specified by performing `dest` – `src`. However, the result of this subtraction is not stored (unlike the `sub` instruction), but only the flags are updated to reflect the result of the subtract operation. This instruction is typically used in conjunction with conditional jumps. If an operand greater than 1 byte is compared to an immediate byte, the byte value is first sign-extended. Clock cycles: 1 if no memory operand is involved; 2 if one of the operands is in memory.

cmps — Compare string operands

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
cmps    dest,src
cmpsb
cmpsw
cmpsd
```

Description: Compares the byte, word, or doubleword pointed by the source index register (SI or ESI) with an operand of equal size pointed by the destination index register (DI or EDI). If the address size is 16 bits, SI and DI registers are used; ESI and EDI registers are used for 32-bit addresses. The comparison is done by subtracting the operand pointed by the DI or EDI register from that by the SI or ESI register. That is, the `cmps` instructions performs either [SI]−[DI] or [ESI]−[EDI]. The result is not stored but used to update the flags, as in the `cmp` instruction. After the comparison, both source and destination index registers are automatically updated. Whether these two registers are incremented or decremented depends on the direction flag (DF). The registers are incremented if DF is 0 (see the `cld` instruction to clear the direction flag); if the DF is 1, both index registers are decremented (see the `std` instruction to set the direction flag). The two registers are incremented or decremented by 1 for byte comparisons, 2 for word comparisons, and 4 for doubleword comparisons.

Note that the specification of the operands in `cmps` is not really required, as the two operands are assumed to be pointed by the index registers. The `cmpsb`, `cmpsw`, and `cmpsd` are synonyms for the byte, word, and doubleword `cmps` instructions, respectively.

The repeat prefix instructions (i.e., `rep`, `repe`, or `repne`) can precede the `cmps` instructions for array or string comparisons. See `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 the 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 either 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 6 for details. Clock cycles: 17 for an 8-bit divisor, 25 for a 16-bit divisor, and 41 for a 32-bit divisor.

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 the `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 either 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 6 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 memory, `dest` must be a register.

In the three-operand format, a constant can be specified as an immediate operand. The result (`src × constant`) is stored in `dest`. As in the two-operand format, the `dest` operand must be a register. The `src` can be either in a register or memory. The immediate constant can be an 8-, 16-, or 32-bit value. For additional restrictions, refer to the Pentium data book. Clock cycles: 10 (11 if the one-operand format is used with either 8- or 16-bit operands).

in — Input from a port

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
in    dest,port
in    dest,DX
```

Description: This instruction has two formats. In both formats, `dest` must be the AL, AX, or EAX register. In the first format, it reads a byte, word, or doubleword from `port` into the AL, AX, or EAX register, respectively. Note that `port` is an 8-bit immediate value. This format is restrictive in the sense that only the first 256 ports can be accessed. The other format is more flexible and allows access to the complete I/O space (i.e., any port between 0 and 65,535). In this format, the port number is assumed to be in the DX register. Clock cycles: varies—see Pentium data book.

inc — Increment by 1

C	O	Z	S	P	A
–	M	M	M	M	M

Format:

```
inc    dest
```

Description: The `inc` instruction increments the `dest` operand by 1. The carry flag is not affected. Clock cycles: 1 if `dest` is a register; 3 if `dest` is in memory.

ins — Input from a port to string

C	O	Z	S	P	A
–	–	–	–	–	–

Format:
```
insb
insw
insd
```

Description: This instruction transfers 8-, 16-, or 32-bit data from the input port specified in the DX register to a location in memory pointed by ES:(E)DI. The DI index register is used if the address size is 16 bits, and the EDI index register is used for 32-bit addresses. Unlike the `in` instruction, the `ins` instruction does not allow the specification of the port number as an immediate value. After the data transfer, the index register is updated automatically. The index register is incremented if DF is 0; it is decremented if DF is 1. The index register is incremented or decremented by 1, 2, or 4 for byte, word, or doubleword operands, respectively. The repeat prefix can be used along with the `ins` instruction to transfer a block of data (the number of data transfers is indicated by the CX register—see the `rep` instruction for details). Clock cycles: varies—see Pentium data book.

int — Interrupt

C	O	Z	S	P	A
–	–	–	–	–	–

Format:
```
int    interrupt-type
```

Description: The `int` instruction calls an interrupt service routine or handler associated with `interrupt-type`. The `interrupt-type` is an immediate 8-bit operand. This value is used as an index into the Interrupt Descriptor table (IDT). See Chapter 12 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 to be used. See Chapter 12 for more details. This instruction affects all flags as the flags register is popped from the 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 7. See Chapter 7 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 7 for an example. For other forms of the `jmp` instruction, see the Pentium data book. Note: Flags are not affected unless there is a task switch, in which case all flags are affected. Clock cycles: 1 for direct jumps, 2 for indirect jumps (more clock cycles for other types of jumps).

lahf — Load flags into AH register

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `lahf`

Description: The `lahf` instruction loads the AH register with the low byte of the flags register. AH := SF, ZF, *, AF, *, PF, *, CF, where * represents an 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 corresponding segment registers (e.g., the DS register for the `lds` instruction, the 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 is 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 is 16 bits, the effective address is zero-extended to 32 bits. Clock cycles: 1.

lods — Load string operand

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
lodsb
lodsw
lodsd
```

Description: The `lods` instruction loads the AL, AX, or EAX register with the memory byte, word, or doubleword at the location pointed by DS:SI or DS:ESI. The address size attribute determines whether the SI or ESI register is used. The `lodsw` and `loadsd` instructions share the same opcode (ADH). The operand size is used to load either a word or doubleword. After loading, the source index register is updated automatically. The index register is incremented if DF is 0; it is decremented if DF is 1. The index register is incremented or decremented by 1, 2, or 4 for byte, word, or doubleword operands, respectively. The `rep` prefix can be used with this instruction but is not useful, as explained in Chapter 9. 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 the `target` if ZF is 0 (instead of 1 as in the `loope` instruction). See Chapter 7 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, the SI and DI registers are used; ESI and EDI registers are used for 32-bit addresses. The default segment for the source is DS and ES for the destination. The segment override prefix can be used only for the source operand. After the move, both source and destination index registers are automatically updated as in the `cmps` instruction.

The `rep` prefix instruction can precede the `movs` instruction for block movement of data. See the `rep` instruction for details. Clock cycles: 4.

movsx — Copy with sign extension

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
movsx    reg16,src8
movsx    reg32,src8
movsx    reg32,src16
```

Description: Copies the sign-extended source operand `src8/src16` into the destination `reg16/reg32`. The destination can be either a 16-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 the `movsx` instruction except `movzx` copies the zero-extended source operand into destination. Clock cycles: 3.

mul — Unsigned multiplication

C	O	Z	S	P	A
M	M	*	*	*	*

Format:

```
mul    AL,src8
mul    AX,src16
mul    EAX,src32
```

Description: Performs unsigned multiplication of two 8-, 16-, or 32-bit operands. Only one of the operands 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, `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 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 65535). In this format, the port number is assumed to be in the DX register. Clock cycles: varies—see Pentium data book.

outs — Output from a string to a port

C	O	Z	S	P	A
–	–	–	–	–	–

Format:

```
outsb
outsw
outsd
```

Description: This instruction transfers 8-, 16-, or 32-bit data from a string (pointed by the source index register) to the output port specified in the DX register. Similar to the `ins` instruction, it uses the SI index register for 16-bit addresses and the ESI register if the address size is 32. The (E)SI register is automatically updated after the transfer of a data item. The index register is incremented if DF is 0; it is decremented if DF is 1. The index register is incremented or decremented by 1, 2, or 4 for byte, word, or doubleword operands, respectively. The repeat prefix can be used with `outs` for block transfer of data. Clock cycles: varies—see Pentium data book.

pop — Pop a word from the stack

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `pop dest`

Description: Pops a word or doubleword from the top of the stack. If the address size attribute is 16 bits, SS:SP is used as the top of the stack pointer; otherwise, SS:ESP is used. `dest` can be a register or memory operand. In addition, it can also be a segment register DS, ES, SS, FS, or GS (e.g., pop DS). The stack pointer is incremented by 2 (if the operand size is 16 bits) or 4 (if the operand size is 32 bits). Note that pop CS is not allowed. This can be done only indirectly by the `ret` instruction. Clock cycles: 1 if `dest` is a general register; 3 if `dest` is a segment register or memory operand.

popa — Pop all general registers

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `popa`
`popad`

Description: Pops all eight 16-bit (popa) or 32-bit (popad) general registers from the top of the stack. The popa loads the registers in the order DI, SI, BP, and discards the 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, (ii) a memory operand, (iii) a segment register (CS, SS, DS, ES, FS, or GS), or (iv) an immediate byte, word, or doubleword operand. The stack pointer is decremented by 2 (if the operand size is 16 bits) or 4 (if the operand size is 32 bits). The `push ESP` instruction pushes the ESP register value before it 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; it 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. It 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 support 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 8. 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 8. The OF flag is affected only for single bit rotates; it is undefined for multi-bit rotates. Clock cycles: `rol` and `ror` take 1 (if `src` is a register) or 3 (if `src` is a memory operand) for the immediate mode (constant 1 or `count`) and 4 for the CL version; for the other two instructions, it can take as many as 27 clock cycles—see the 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 `rep cmps` and `rep scas` instructions. For more details, see Chapter 9. Clock cycles: varies—see the 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 4 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 the 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 support 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 8. The second operand gives the number of times `src` is to be shifted. This operand can be given as an immediate value (a constant 1 or a byte value `count`) or preloaded into the CL register. The `shr` (shift right) is similar to `shl` except for the direction of shift. The `sar` (shift arithmetic right) is similar to `sal` except for two differences: the shift direction is right and the sign bit is copied into the vacated bits. If the shift count is zero, no flags are affected. The CF flag contains the last bit shifted out. The OF flag is defined only for single shifts; it is undefined for multi-bit shifts. Clock cycles: 1 (if `src` is a register) or 3 (if `src` is a memory operand) for the immediate mode (constant 1 or `count`) and 4 for the CL version.

sbb — Subtract with borrow

C	O	Z	S	P	A
M	M	M	M	M	M

Format: `sbb dest,src`

Description: Performs integer subtraction with borrow. The `dest` is assigned the result of `dest -(src+CF)`. Clock cycles: 1–3.

scas — Compare string operands

C	O	Z	S	P	A
M	M	M	M	M	M

Format:

```
scas    operand
scasb
scasw
scasd
```

Description: Subtracts the memory byte, word, or doubleword pointed by the destination index register (DI or EDI) from the AL, AX, or EAX register, respectively. The result is not stored but used to update the flags. The memory operand must be addressable from the ES register. Segment override is not allowed in this instruction. If the address size is 16 bits, the DI register is used; the EDI register is used for 32-bit addresses. After the subtraction, the destination index register is updated automatically. Whether the register is incremented or decremented depends on the direction flag (DF). The register is incremented if DF is 0 (see the `cld` instruction to clear the direction flag); if the DF is 1, the index register is decremented (see the `std` instruction to set the direction flag). The amount of increment or decrement is 1 (for byte operands), 2 (for word operands), or 4 (for doubleword operands).

Note that the specification of the operand in `scas` is not really required, as the memory operand is assumed to be pointed by the index register. The `scasb`, `scasw`, and `scasd` are synonyms for the byte, word, and doubleword `scas` instructions, respectively.

The repeat prefix instructions (i.e., `repe` or `repne`) can precede the `scas` instructions for array or string comparisons. See the `rep` instruction for details. Clock cycles: 4.

setCC — Byte set on condition operands

C	O	Z	S	P	A
–	–	–	–	–	–

Format: `setCC dest`

Description: Sets `dest` byte to 1 if the condition CC is met; otherwise, sets to zero. The operand `dest` must be either an 8-bit register or memory operand. The conditions tested are similar to the conditional jump instruction (see the `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`. The `dest` and `src` operands can both be either 16- or 32-bit operands. While `dest` can be a register or memory operand, `src` must be a register of the same size as `dest`. The third operand `count` can be an immediate byte value, or the CL register can be used as in the shift instructions. The contents of the `src` register are not altered.

The `shrd` instruction (double precision shift right) is similar to `shld` except for the direction of shift.

If 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 multi-bit 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`

```
stosb
stosw
stosd
```

Description: Stores the contents of the AL, AX, or EAX register at the memory byte, word, or doubleword pointed by the destination index register (DI or EDI), respectively. If the address size is 16 bits, the DI register is used; the EDI register is used for 32-bit addresses. After the load, the destination index register is automatically updated. Whether this register is incremented or decremented depends on the direction flag (DF). The register is incremented if DF is 0 (see the `cld` instruction to clear the direction flag); if 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 E

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 635.

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	^	7E	126	~
3F	63	?	5F	95	_			

Note that 7FH (127 in decimal) is a control character listed on the previous page.

Index