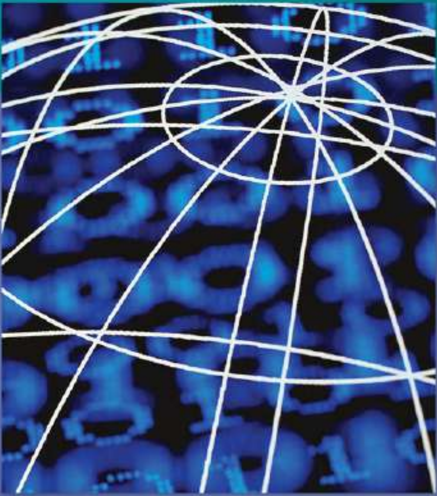
Carl Hamacher · Zvonko Vranesic · Safwat Zaky · Naraig Manjikian



COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS

Sixth Edition

This page intentionally left blank

This page intentionally left blank

COMPUTER ORGANIZATION
AND EMBEDDED SYSTEMS

This page intentionally left blank

COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS

SIXTH EDITION

Carl Hamacher
Queen's University

Zvonko Vranesic
University of Toronto

Safwat Zaky
University of Toronto

Naraig Manjikian
Queen's University

Mc Graw Hill
Connect Learn Succeed™

The McGraw·Hill Companies

COMPUTER ORGANIZATION AND EMBEDDED SYSTEMS, SIXTH EDITION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2012 by The McGraw-Hill Companies, Inc. All rights reserved. Previous editions 2002, 1996, and 1990. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 DOC/DOC 0 9 8 7 6 5 4 3 2 1

ISBN 978–0–07–338065–0
MHID 0–07–338065–2

Vice President & Editor-in-Chief: *Marty Lange*
Vice President EDP/Central Publishing Services: *Kimberly Meriwether David*
Publisher: *Raghothaman Srinivasan*
Senior Sponsoring Editor: *Peter E. Massar*
Developmental Editor: *Darlene M. Schueller*
Senior Marketing Manager: *Curt Reynolds*
Senior Project Manager: *Lisa A. Bruflodt*
Buyer: *Laura Fuller*
Design Coordinator: *Brenda A. Rolwes*
Media Project Manager: *Balaji Sundararaman*
Cover Design: *Studio Montage, St. Louis, Missouri*
Cover Image: *© Royalty-Free/CORBIS*
Compositor: *Techsetters, Inc.*
Typeface: *10/12 Times Roman*
Printer: *R. R. Donnelley & Sons Company/Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Computer organization and embedded systems / Carl Hamacher ... [et al.]. – 6th ed.
p. cm.
Includes bibliographical references.
ISBN-13: 978-0-07-338065-0 (alk. paper)
ISBN-10: 0-07-338065-2 (alk. paper)
1. Computer organization. 2. Embedded computer systems. I. Hamacher, V. Carl.
QA76.9.C643.H36 2012
004.2'2–dc22
2010050243

www.mhhe.com

To our families

This page intentionally left blank

About the Authors

Carl Hamacher received the B.A.Sc. degree in Engineering Physics from the University of Waterloo, Canada, the M.Sc. degree in Electrical Engineering from Queen's University, Canada, and the Ph.D. degree in Electrical Engineering from Syracuse University, New York. From 1968 to 1990 he was at the University of Toronto, Canada, where he was a Professor in the Department of Electrical Engineering and the Department of Computer Science. He served as director of the Computer Systems Research Institute during 1984 to 1988, and as chairman of the Division of Engineering Science during 1988 to 1990. In 1991 he joined Queen's University, where is now Professor Emeritus in the Department of Electrical and Computer Engineering. He served as Dean of the Faculty of Applied Science from 1991 to 1996. During 1978 to 1979, he was a visiting scientist at the IBM Research Laboratory in San Jose, California. In 1986, he was a research visitor at the Laboratory for Circuits and Systems associated with the University of Grenoble, France. During 1996 to 1997, he was a visiting professor in the Computer Science Department at the University of California at Riverside and in the LIP6 Laboratory of the University of Paris VI.

His research interests are in multiprocessors and multicomputers, focusing on their interconnection networks.

Zvonko Vranesic received his B.A.Sc., M.A.Sc., and Ph.D. degrees, all in Electrical Engineering, from the University of Toronto. From 1963 to 1965 he worked as a design engineer with the Northern Electric Co. Ltd. in Bramalea, Ontario. In 1968 he joined the University of Toronto, where he is now a Professor Emeritus in the Department of Electrical & Computer Engineering. During the 1978–79 academic year, he was a Senior Visitor at the University of Cambridge, England, and during 1984-85 he was at the University of Paris, 6. From 1995 to 2000 he served as Chair of the Division of Engineering Science at the University of Toronto. He is also involved in research and development at the Altera Toronto Technology Center.

His current research interests include computer architecture and field-programmable VLSI technology.

He is a coauthor of four other books: *Fundamentals of Digital Logic with VHDL Design*, 3rd ed.; *Fundamentals of Digital Logic with Verilog Design*, 2nd ed.; *Microcomputer Structures*; and *Field-Programmable Gate Arrays*. In 1990, he received the Wighton Fellowship for "innovative and distinctive contributions to undergraduate laboratory instruction." In 2004, he received the Faculty Teaching Award from the Faculty of Applied Science and Engineering at the University of Toronto.

Safwat Zaky received his B.Sc. degree in Electrical Engineering and B.Sc. in Mathematics, both from Cairo University, Egypt, and his M.A.Sc. and Ph.D. degrees in Electrical Engineering from the University of Toronto. From 1969 to 1972 he was with Bell Northern Research, Bramalea, Ontario, where he worked on applications of electro-optics and

This page intentionally left blank

magnetics in mass storage and telephone switching. In 1973, he joined the University of Toronto, where he is now Professor Emeritus in the Department of Electrical and Computer Engineering. He served as Chair of the Department from 1993 to 2003 and as Vice-Provost from 2003 to 2009. During 1980 to 1981, he was a senior visitor at the Computer Laboratory, University of Cambridge, England.

He is a Fellow of the Canadian Academy of Engineering. His research interests are in the areas of computer architecture, digital-circuit design, and electromagnetic compatibility. He is a coauthor of the book *Microcomputer Structures* and is a recipient of the IEEE Third Millennium Medal and of the Vivek Goel Award for distinguished service to the University of Toronto.

Naraig Manjikian received his B.A.Sc. degree in Computer Engineering and M.A.Sc. degree in Electrical Engineering from the University of Waterloo, Canada, and his Ph.D. degree in Electrical Engineering from the University of Toronto. In 1997, he joined Queen's University, Kingston, Canada, where he is now an Associate Professor in the Department of Electrical and Computer Engineering. From 2004 to 2006, he served as Undergraduate Chair for Computer Engineering. From 2006 to 2007, he served as Acting Head of the Department of Electrical and Computer Engineering, and from 2007 until 2009, he served as Associate Head for Student and Alumni Affairs. During 2003 to 2004, he was a visiting professor at McGill University, Montreal, Canada, and the University of British Columbia. During 2010 to 2011, he was a visiting professor at McGill University.

His research interests are in the areas of computer architecture, multiprocessor systems, field-programmable VLSI technology, and applications of parallel processing.

PREFACE

This book is intended for use in a first-level course on computer organization and embedded systems in electrical engineering, computer engineering, and computer science curricula. The book is self-contained, assuming only that the reader has a basic knowledge of computer programming in a high-level language. Many students who study computer organization will have had an introductory course on digital logic circuits. Therefore, this subject is not covered in the main body of the book. However, we have provided an extensive appendix on logic circuits for those students who need it.

The book reflects our experience in teaching three distinct groups of students: electrical and computer engineering undergraduates, computer science undergraduates, and engineering science undergraduates. We have always approached the teaching of courses on computer organization from a practical point of view. Thus, a key consideration in shaping the contents of the book has been to carefully explain the main principles, supported by examples drawn from commercially available processors. Our main commercial examples are based on: Altera's Nios II, Freescale's ColdFire, ARM, and Intel's IA-32 architectures.

It is important to recognize that digital system design is not a straightforward process of applying optimal design algorithms. Many design decisions are based largely on heuristic judgment and experience. They involve cost/performance and hardware/software tradeoffs over a range of alternatives. It is our goal to convey these notions to the reader.

The book is aimed at a one-semester course in engineering or computer science programs. It is suitable for both hardware- and software-oriented students. Even though the emphasis is on hardware, we have addressed a number of relevant software issues.

McGraw-Hill maintains a Website with support material for the book at http://www.mhhe.com/hamacher.

SCOPE OF THE BOOK

The first three chapters introduce the basic structure of computers, the operations that they perform at the machine-instruction level, and input/output methods as seen by a programmer. The fourth chapter provides an overview of the system software needed to translate programs written in assembly and high-level languages into machine language and to manage their execution. The remaining eight chapters deal with the organization, interconnection, and performance of hardware units in modern computers, including a coverage of embedded systems.

Five substantial appendices are provided. The first appendix covers digital logic circuits. Then, four current commercial instruction set architectures—Altera's Nios II, Freescale's ColdFire, ARM, and Intel's IA-32—are described in separate appendices.

Chapter 1 provides an overview of computer hardware and informally introduces terms that are discussed in more depth in the remainder of the book. This chapter discusses the basic functional units and the ways they interact to form a complete computer system. Number and character representations are discussed, along with basic arithmetic operations. An introduction to performance issues and a brief treatment of the history of computer development are also provided.

Chapter 2 gives a methodical treatment of machine instructions, addressing techniques, and instruction sequencing. Program examples at the machine-instruction level, expressed in a generic assembly language, are used to discuss concepts that include loops, subroutines, and stacks. The concepts are introduced using a RISC-style instruction set architecture. A comparison with CISC-style instruction sets is also included.

Chapter 3 presents a programmer's view of basic input/output techniques. It explains how program-controlled I/O is performed using polling, as well as how interrupts are used in I/O transfers.

Chapter 4 considers system software. The tasks performed by compilers, assemblers, linkers, and loaders are explained. Utility programs that trace and display the results of executing a program are described. Operating system routines that manage the execution of user programs and their input/output operations, including the handling of interrupts, are also described.

Chapter 5 explores the design of a RISC-style processor. This chapter explains the sequence of processing steps needed to fetch and execute the different types of machine instructions. It then develops the hardware organization needed to implement these processing steps. The differing requirements of CISC-style processors are also considered.

Chapter 6 provides coverage of the use of pipelining and multiple execution units in the design of high-performance processors. A pipelined version of the RISC-style processor design from Chapter 5 is used to illustrate pipelining. The role of the compiler and the relationship between pipelined execution and instruction set design are explored. Superscalar processors are discussed.

Input/output hardware is considered in **Chapter 7**. Interconnection networks, including the bus structure, are discussed. Synchronous and asynchronous operation is explained. Interconnection standards, including USB and PCI Express, are also presented.

Semiconductor memories, including SDRAM, Rambus, and Flash memory implementations, are discussed in **Chapter 8**. Caches are explained as a way for increasing the memory bandwidth. They are discussed in some detail, including performance modeling. Virtual-memory systems, memory management, and rapid address-translation techniques are also presented. Magnetic and optical disks are discussed as components in the memory hierarchy.

Chapter 9 explores the implementation of the arithmetic unit of a computer. Logic design for fixed-point add, subtract, multiply, and divide hardware, operating on 2's-complement numbers, is described. Carry-lookahead adders and high-speed multipliers are explained, including descriptions of the Booth multiplier recoding and carry-save addition techniques. Floating-point number representation and operations, in the context of the IEEE Standard, are presented.

Today, far more processors are in use in embedded systems than in general-purpose computers. **Chapters 10 and 11** are dedicated to the subject of embedded systems. First, basic aspects of system integration, component interconnections, and real-time operation are presented in Chapter 10. The use of microcontrollers is discussed. Then, Chapter 11 concentrates on system-on-a-chip (SoC) implementations, in which a single chip integrates the processing, memory, I/O, and timer functionality needed to satisfy application-specific requirements. A substantial example shows how FPGAs and modern design tools can be used in this environment.

Chapter 12 focuses on parallel processing and performance. Hardware multithreading and vector processing are introduced as enhancements in a single processor. Shared-memory multiprocessors are then described, along with the issue of cache coherence. Interconnection networks for multiprocessors are presented.

Appendix A provides extensive coverage of logic circuits, intended for a reader who has not taken a course on the design of such circuits.

Appendices B, C, D, and E illustrate how the instruction set concepts introduced in Chapters 2 and 3 are implemented in four commercial processors: Nios II, ColdFire, ARM, and Intel IA-32. The Nios II and ARM processors illustrate the RISC design style. ColdFire has an easy-to-teach CISC design, while the IA-32 CISC architecture represents the most successful commercial design. The presentation for each processor includes assembly-language examples from Chapters 2 and 3, implemented in the context of that processor. The details given in these appendices are not essential for understanding the material in the main body of the book. It is sufficient to cover only one of these appendices to gain an appreciation for commercial processor instruction sets. The choice of a processor to use as an example is likely to be influenced by the equipment in an accompanying laboratory. Instructors may wish to use more that one processor to illustrate the different design approaches.

Changes in the Sixth Edition

Substantial changes in content and organization have been made in preparing the sixth edition of this book. They include the following:

- The basic concepts of instruction set architecture are now covered using the RISC-style approach. This is followed by a comparative examination of the CISC-style approach.
- The processor design discussion is focused on a RISC-style implementation, which leads naturally to pipelined operation.
- Two chapters on embedded systems are included: one dealing with the basic structure of such systems and the use of microcontrollers, and the other dealing with system-on-a-chip implementations.
- Appendices are used to give examples of four commercial processors. Each appendix includes the essential information about the instruction set architecture of the given processor.
- Solved problems have been included in a new section toward the end of chapters and appendices. They provide the student with solutions that can be expected for typical problems.

Difficulty Level of Problems

The problems at the end of chapters and appendices have been classified as easy (E), medium (M), or difficult (D). These classifications should be interpreted as follows:

- Easy—Solutions can be derived in a few minutes by direct application of specific information presented in one place in the relevant section of the book.
- Medium—Use of the book material in a way that does not directly follow any examples presented is usually needed. In some cases, solutions may follow the general pattern of an example, but will take longer to develop than those for easy problems.
- Difficult—Some additional insight is needed to solve these problems. If a solution requires a program to be written, its underlying algorithm or form may be quite different from that of any program example given in the book. If a hardware design is required, it may involve an arrangement and interconnection of basic logic circuit components that is quite different from any design shown in the book. If a performance analysis is needed, it may involve the derivation of an algebraic expression.

What Can Be Covered in a One-Semester Course

This book is suitable for use at the university or college level as a text for a one-semester course in computer organization. It is intended for the first course that students will take on computer organization.

There is more than enough material in the book for a one-semester course. The core material on computer organization and relevant software issues is given in Chapters 1 through 9. For students who have not had a course in logic circuits, the material in Appendix A should be studied near the beginning of a course and certainly prior to covering Chapter 5.

A course aimed at embedded systems should include Chapters 1, 2, 3, 4, 7, 8, 10 and 11.

Use of the material on commercial processor examples in Appendices B through E can be guided by instructor and student interest, as well as by relevance to any hardware laboratory associated with a course.

Acknowledgments

We wish to express our thanks to many people who have helped us during the preparation of this sixth edition of the book.

Our colleagues Daniel Etiemble of University of Paris South and Glenn Gulak of University of Toronto provided numerous comments and suggestions that helped significantly in shaping the material.

Blair Fort and Dan Vranesic provided valuable help with some of the programming examples.

Warren R. Carithers of Rochester Institute of Technology, Krishna M. Kavi of University of North Texas, and Nelson Luiz Passos of Midwestern State University provided reviews of material from both the fifth and sixth editions of the book.

The following people provided reviews of material from the fifth edition of the book: Goh Hock Ann of Multimedia University, Joseph E. Beaini of University of Colorado Denver, Kalyan Mohan Goli of Jawaharlal Nehru Technological University, Jaimon Jacob of Model Engineering College Ernakulam, M. Kumaresan of Anna University Coimbatore, Kenneth K. C. Lee of City University of Hong Kong, Manoj Kumar Mishra of Institute of Technical Education and Research, Junita Mohamad-Saleh of Universiti Sains Malaysia, Prashanta Kumar Patra of College of Engineering and Technology Bhubaneswar, Shanq-Jang Ruan of National Taiwan University of Science and Technology, S. D. Samantaray of G. B. Pant University of Agriculture and Technology, Shivakumar Sastry of University of Akron, Donatella Sciuto of Politecnico of Milano, M. P. Singh of National Institute of Technology Patna, Albert Starling of University of Arkansas, Shannon Tauro of University of California Irvine, R. Thangarajan of Kongu Engineering College, Ashok Kunar Turuk of National Institute of Technology Rourkela, and Philip A. Wilsey of University of Cincinnati.

Finally, we truly appreciate the support of Raghothaman Srinivasan, Peter E. Massar, Darlene M. Schueller, Lisa Bruflodt, Curt Reynolds, Brenda Rolwes, and Laura Fuller at McGraw-Hill.

Carl Hamacher
Zvonko Vranesic
Safwat Zaky
Naraig Manjikian

CONTENTS

chapter

1

Basic Structure of Computers

Chapter Objectives

In this chapter you will be introduced to:

- The different types of computers
- The basic structure of a computer and its operation
- Machine instructions and their execution
- Number and character representations
- Addition and subtraction of binary numbers
- Basic performance issues in computer systems
- A brief history of computer development

This book is about computer organization. It explains the function and design of the various units of digital computers that store and process information. It also deals with the input units of the computer which receive information from external sources and the output units which send computed results to external destinations. The input, storage, processing, and output operations are governed by a list of instructions that constitute a program.

Most of the material in the book is devoted to *computer hardware* and *computer architecture*. Computer hardware consists of electronic circuits, magnetic and optical storage devices, displays, electromechanical devices, and communication facilities. Computer architecture encompasses the specification of an instruction set and the functional behavior of the hardware units that implement the instructions.

Many aspects of programming and software components in computer systems are also discussed in the book. It is important to consider both hardware and software aspects of the design of the various computer components in order to gain a good understanding of computer systems.

1.1 COMPUTER TYPES

Since their introduction in the 1940s, digital computers have evolved into many different types that vary widely in size, cost, computational power, and intended use. Modern computers can be divided roughly into four general categories:

- *Embedded computers* are integrated into a larger device or system in order to automatically monitor and control a physical process or environment. They are used for a specific purpose rather than for general processing tasks. Typical applications include industrial and home automation, appliances, telecommunication products, and vehicles. Users may not even be aware of the role that computers play in such systems.
- *Personal computers* have achieved widespread use in homes, educational institutions, and business and engineering office settings, primarily for dedicated individual use. They support a variety of applications such as general computation, document preparation, computer-aided design, audiovisual entertainment, interpersonal communication, and Internet browsing. A number of classifications are used for personal computers. *Desktop computers* serve general needs and fit within a typical personal workspace. *Workstation computers* offer higher computational capacity and more powerful graphical display capabilities for engineering and scientific work. Finally, *Portable* and *Notebook computers* provide the basic features of a personal computer in a smaller lightweight package. They can operate on batteries to provide mobility.
- *Servers* and *Enterprise systems* are large computers that are meant to be shared by a potentially large number of users who access them from some form of personal computer over a public or private network. Such computers may host large databases and provide information processing for a government agency or a commercial organization.
- *Supercomputers* and *Grid computers* normally offer the highest performance. They are the most expensive and physically the largest category of computers. Supercomputers are used for the highly demanding computations needed in weather forecasting, engineering design and simulation, and scientific work. They have a high cost. Grid computers provide a more cost-effective alternative. They combine a large number of personal computers and

disk storage units in a physically distributed high-speed network, called a grid, which is managed as a coordinated computing resource. By evenly distributing the computational workload across the grid, it is possible to achieve high performance on large applications ranging from numerical computation to information searching.

There is an emerging trend in access to computing facilities, known as *cloud computing*. Personal computer users access widely distributed computing and storage server resources for individual, independent, computing needs. The Internet provides the necessary communication facility. Cloud hardware and software service providers operate as a utility, charging on a pay-as-you-use basis.

1.2 Functional Units

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1. The input unit accepts coded information from human operators using devices such as keyboards, or from other computers over digital communication lines. The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions. Later chapters will provide more details on individual units and their interconnections. We refer to the

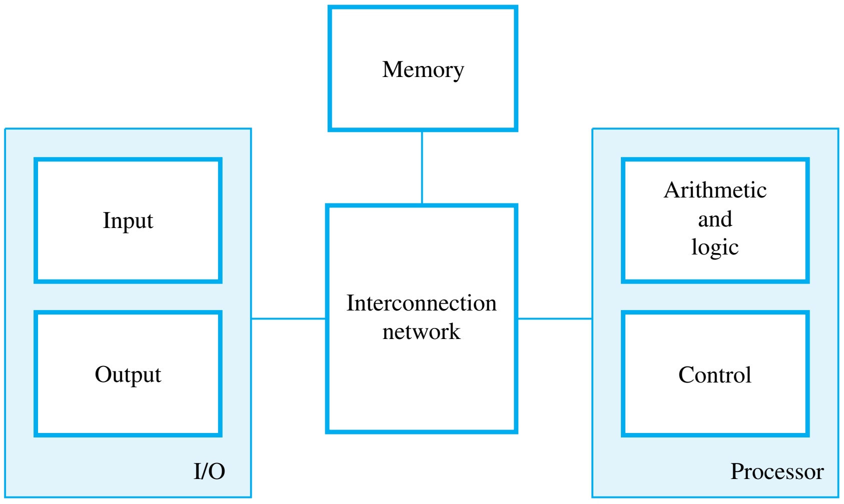


Figure 1.1 Basic functional units of a computer.

arithmetic and logic circuits, in conjunction with the main control circuits, as the *processor*. Input and output equipment is often collectively referred to as the *input-output* (I/O) unit.

We now take a closer look at the information handled by a computer. It is convenient to categorize this information as either instructions or data. *Instructions*, or *machine instructions*, are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices
- Specify the arithmetic and logic operations to be performed

A *program* is a list of instructions which performs a task. Programs are stored in the memory. The processor fetches the program instructions from the memory, one after another, and performs the desired operations. The computer is controlled by the stored program, except for possible external interruption by an operator or by I/O devices connected to it. *Data* are numbers and characters that are used as operands by the instructions. Data are also stored in the memory.

The instructions and data handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states. Each instruction, number, or character is encoded as a string of binary digits called *bits*, each having one of two possible values, 0 or 1, represented by the two stable states. Numbers are usually represented in positional binary notation, as discussed in Section 1.4. Alphanumeric characters are also expressed in terms of binary codes, as discussed in Section 1.5.

1.2.1 Input Unit

Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.

Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.

Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

1.2.2 Memory Unit

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

Primary memory, also called *main memory*, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually. Instead, they are handled in groups of fixed size called *words*. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the *word length* of the computer, typically 16, 32, or 64 bits.

To provide easy access to any word in the memory, a distinct *address* is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.

Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a *random-access memory* (RAM). The time required to access one word is called the *memory access time*. This time is independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for current RAM units.

Cache Memory

As an adjunct to the main memory, a smaller, faster RAM unit, called a *cache*, is used to hold sections of a program that are currently being executed, along with any associated data. The cache is tightly coupled with the processor and is usually contained on the same integrated-circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.

Now, suppose a number of instructions are executed repeatedly as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.

Secondary Storage

Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent *secondary storage* is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including *magnetic disks*, *optical disks* (DVD and CD), and *flash memory devices*.

1.2.3 Arithmetic and Logic Unit

Most computer operations are executed in the *arithmetic and logic unit* (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

When operands are brought into the processor, they are stored in high-speed storage elements called *registers*. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

1.2.4 Output Unit

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a *printer*. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.

Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touchscreen capability. The dual role of such units is the reason for using the single name *input/output* (I/O) unit in many cases.

1.2.5 Control Unit

The memory, arithmetic and logic, and I/O units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the responsibility of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by program instructions that identify the devices involved and the information to be transferred. Control circuits are responsible for generating the *timing signals* that govern the transfers and determine when a given action is to take place. Data transfers between the processor and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

1.3 Basic Operational Concepts

In Section 1.2, we stated that the activity in a computer is governed by instructions. To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.

A typical instruction might be

```
Load     R2, LOC
```

This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2. The original contents of location LOC are preserved, whereas those of register R2 are overwritten. Execution of this instruction requires several steps. First, the instruction is fetched from the memory into the processor. Next, the operation to be performed is determined by the control unit. The operand at LOC is then fetched from the memory into the processor. Finally, the operand is stored in register R2.

After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them. For example, the instruction

```
Add     R4, R2, R3
```

adds the contents of registers R2 and R3, then places their sum into register R4. The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

After completing the desired operations, the results are in processor registers. They can be transferred to the memory using instructions such as

```
Store     R4, LOC
```

This instruction copies the operand in register R4 to memory location LOC. The original contents of location LOC are overwritten, but those of R4 are preserved.

For Load and Store instructions, transfers between the memory and the processor are initiated by sending the address of the desired memory location to the memory unit and asserting the appropriate control signals. The data are then transferred to or from the memory.

Figure 1.2 shows how the memory and the processor can be connected. It also shows some components of the processor that have not been discussed yet. The interconnections between these components are not shown explicitly since we will only discuss their functional characteristics here. Chapter 5 describes the details of the interconnections as part of processor organization.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The *instruction register* (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction. The *program counter* (PC) is another specialized register. It

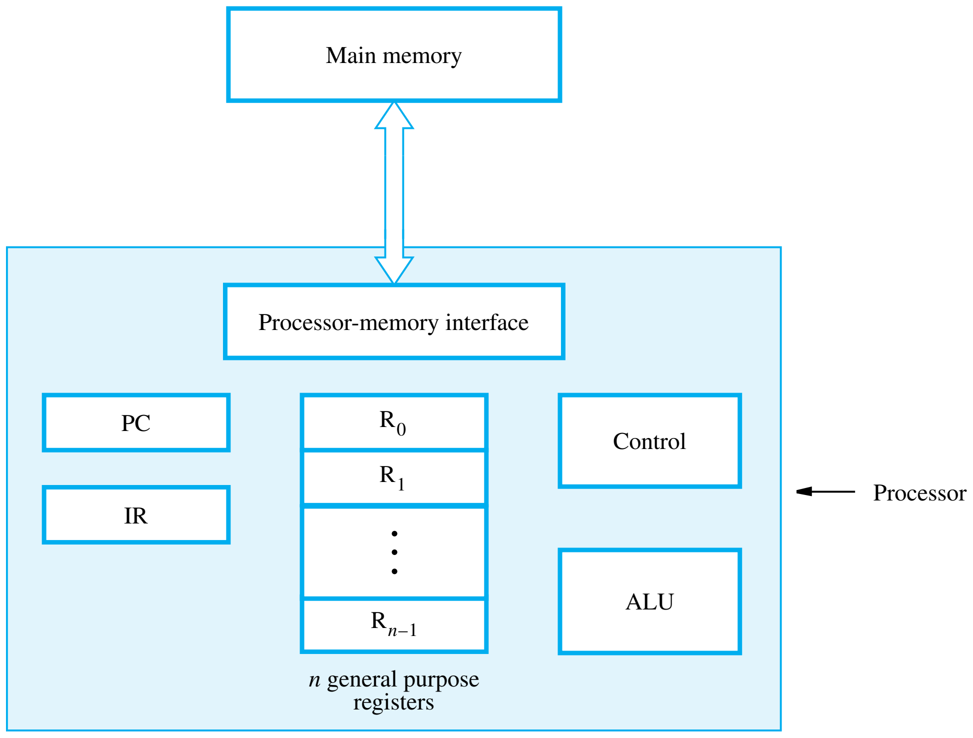


Figure 1.2 Connection between the processor and the main memory.

contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC *points* to the next instruction that is to be fetched from the memory. In addition to the IR and PC, Figure 1.2 shows *general-purpose registers* R_0 through R_{n-1}, often called processor registers. They serve a variety of functions, including holding operands that have been loaded from the memory for processing. The roles of the general-purpose registers are explained in detail in Chapter 2.

The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal. The interface waits for the word to be retrieved, then transfers it to the appropriate processor register. If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.

Let us now consider some typical operating steps. A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage through the input unit. Execution of the program begins when the PC is set to point to the first instruction of the program. The contents of the PC are transferred to the memory along with a Read control signal. When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR. At this point, the instruction is ready to be interpreted and executed.

Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation. When the operand has been fetched from the memory, it is transferred to a processor register. After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register. If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.

At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.

In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions are provided for the purpose of handling I/O transfers.

Normal execution of a program may be preempted if some device requires urgent service. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to respond immediately, execution of the current program must be suspended. To cause this, the device raises an *interrupt* signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an *interrupt-service routine*. Because such diversions may alter the internal state of the processor, its state must be saved in the memory before servicing the interrupt request. Normally, the information that is saved includes the contents of the PC, the contents of the general-purpose registers, and some control information. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

This section has provided an overview of the operation of a computer. Detailed discussion of these concepts is given in subsequent chapters, first from the point of view of the programmer in Chapters 2, 3, and 4, and then from the point of view of the hardware designer in later chapters.

1.4 Number Representation and Arithmetic Operations

The most natural way to represent a number in a computer system is by a string of bits, called a binary number. We will first describe binary number representations for integers as well as arithmetic operations on them. Then we will provide a brief introduction to the representation of floating-point numbers.

1.4.1 INTEGERS

Consider an *n*-bit vector

$$B = b_{n-1} \ldots b_1 b_0$$

where $b_i = 0$ or 1 for $0 \leq i \leq n - 1$. This vector can represent an unsigned integer value $V(B)$ in the range 0 to $2^n - 1$, where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

We need to represent both positive and negative numbers. Three systems are used for representing such numbers:

- Sign-and-magnitude
- 1's-complement
- 2's-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. Figure 1.3 illustrates all three representations using 4-bit numbers. Positive values have identical representations in all systems, but negative values have different representations. In the *sign-and-magnitude* system, negative values are represented by changing the most

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	− 0	− 7	− 8
1 0 0 1	− 1	− 6	− 7
1 0 1 0	− 2	− 5	− 6
1 0 1 1	− 3	− 4	− 5
1 1 0 0	− 4	− 3	− 4
1 1 0 1	− 5	− 2	− 3
1 1 1 0	− 6	− 1	− 2
1 1 1 1	− 7	− 0	− 1

Figure 1.3 Binary, signed-integer representations.

significant bit (b_3 in Figure 1.3) from 0 to 1 in the B vector of the corresponding positive value. For example, $+5$ is represented by 0101, and -5 is represented by 1101.

In *1's-complement* representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for -3 is obtained by complementing each bit in the vector 0011 to yield 1100. The same operation, bit complementing, is done to convert a negative number to the corresponding positive value. Converting either way is referred to as forming the 1's-complement of a given number. For n-bit numbers, this operation is equivalent to subtracting the number from $2^n - 1$. In the case of the 4-bit numbers in Figure 1.3, we subtract from $2^4 - 1 = 15$, or 1111 in binary.

Finally, in the *2's-complement* system, forming the 2's-complement of an n-bit number is done by subtracting the number from 2^n. Hence, the 2's-complement of a number is obtained by adding 1 to the 1's-complement of that number.

Note that there are distinct representations for $+0$ and -0 in both the sign-and-magnitude and 1's-complement systems, but the 2's-complement system has only one representation for 0. For 4-bit numbers, as shown in Figure 1.3, the value -8 is representable in the 2's-complement system but not in the other systems. The sign-and-magnitude system seems the most natural, because we deal with sign-and-magnitude decimal values in manual computations. The 1's-complement system is easily related to this system, but the 2's-complement system may appear somewhat unnatural. However, we will show that the 2's-complement system leads to the most efficient way to carry out addition and subtraction operations. It is the one most often used in modern computers.

Addition of Unsigned Integers

Addition of 1-bit numbers is illustrated in Figure 1.4. The sum of 1 and 1 is the 2-bit vector 10, which represents the value 2. We say that the *sum* is 0 and the *carry-out* is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end. The carry-out from a bit pair becomes the *carry-in* to the next bit pair to the left. The carry-in must be added to a bit pair in generating the sum and carry-out at that position. For example, if both bits of a pair are 1 and the carry-in is 1, then the sum is 1 and the carry-out is 1, which represents the value 3.

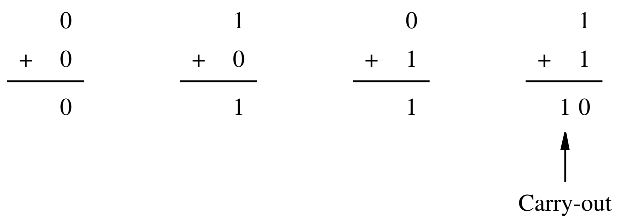


Figure 1.4 Addition of 1-bit numbers.

Addition and Subtraction of Signed Integers

We introduced three systems for representing positive and negative numbers, or, simply, *signed numbers*. These systems differ only in the way they represent negative values. Their relative merits from the standpoint of ease of performing arithmetic operations can be summarized as follows. The sign-and-magnitude system is the simplest representation, but it is also the most awkward for addition and subtraction operations. The 1's-complement method is somewhat better. The 2's-complement system is the most efficient method for performing addition and subtraction operations.

To understand 2's-complement arithmetic, consider addition modulo N (abbreviated as mod N). A helpful graphical device for the description of addition of unsigned integers mod N is a circle with the values 0 through $N - 1$ marked along its perimeter, as shown in Figure 1.5*a*. Consider the case $N = 16$, shown in part (*b*) of the figure. The decimal values 0 through 15 are represented by their 4-bit binary values 0000 through 1111 around the outside of the circle. In terms of decimal values, the operation $(7 + 5)$ mod 16 yields the value 12. To perform this operation graphically, locate 7 (0111) on the outside of the circle and then move 5 units in the clockwise direction to arrive at the answer 12 (1100). Similarly, $(9 + 14)$ mod $16 = 7$; this is modeled on the circle by locating 9 (1001) and moving 14 units in the clockwise direction past the zero position to arrive at the answer 7 (0111). This graphical technique works for the computation of $(a + b)$ mod 16 for any unsigned integers a and b; that is, to perform addition, locate a and move b units in the clockwise direction to arrive at $(a + b)$ mod 16.

Now consider a different interpretation of the mod 16 circle. We will reinterpret the binary vectors outside the circle to represent the signed integers from -8 through $+7$ in the 2's-complement representation as shown inside the circle.

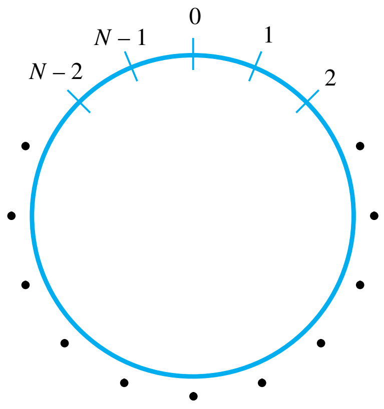
Let us apply the mod 16 addition technique to the example of adding $+7$ to -3. The 2's-complement representation for these numbers is 0111 and 1101, respectively. To add these numbers, locate 0111 on the circle in Figure 1.5*b*. Then move 1101 (13) steps in the clockwise direction to arrive at 0100, which yields the correct answer of $+4$. Note that the 2's-complement representation of -3 is interpreted as an unsigned value for the number of steps to move.

If we perform this addition by adding bit pairs from right to left, we obtain

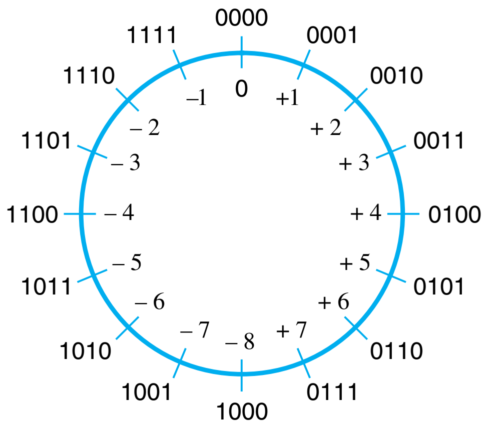
$$\begin{array}{ccccc} & 0 & 1 & 1 & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 \\ \uparrow & & & & \\ \text{Carry-out} & & & & \end{array}$$

If we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer. In fact, this is always the case. Ignoring this carry-out is a natural result of using mod N arithmetic. As we move around the circle in Figure 1.5*b*, the value next to 1111 would normally be 10000. Instead, we go back to the value 0000.

The rules governing addition and subtraction of n-bit signed numbers using the 2's-complement representation system may be stated as follows:



(a) Circle representation of integers mod N



(b) Mod 16 system for 2's-complement numbers

Figure 1.5 Modular number systems and the 2's-complement system.

- To *add* two numbers, add their n-bit representations, ignoring the carry-out bit from the most significant bit (MSB) position. The sum will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

- To *subtract* two numbers X and Y, that is, to perform $X - Y$, form the 2's-complement of Y, then add it to X using the *add* rule. Again, the result will be the algebraically correct value in 2's-complement representation if the actual result is in the range -2^{n-1} through $+2^{n-1} - 1$.

Figure 1.6 shows some examples of addition and subtraction in the 2's-complement system. In all of these 4-bit examples, the answers fall within the representable range of −8 through +7. When answers do not fall within the representable range, we say that *arithmetic overflow* has occurred. A later subsection discusses such situations. The four addition operations (*a*) through (*d*) in Figure 1.6 follow the add rule, and the six subtraction operations (*e*) through (*j*) follow the subtract rule. The subtraction operation requires forming the 2's-complement of the subtrahend (the bottom value). This operation

```
(a)     0 0 1 0    (+2)          (b)     0 1 0 0    (+4)
      + 0 0 1 1    (+3)                + 1 0 1 0    (−6)
      ---------    ----                ---------    ----
        0 1 0 1    (+5)                  1 1 1 0    (−2)

(c)     1 0 1 1    (−5)          (d)     0 1 1 1    (+7)
      + 1 1 1 0    (−2)                + 1 1 0 1    (−3)
      ---------    ----                ---------    ----
        1 0 0 1    (−7)                  0 1 0 0    (+4)

(e)     1 1 0 1    (−3)                  1 1 0 1
      − 1 0 0 1    (−7)       ⇒        + 0 1 1 1
      ---------    ----                ---------    ----
                                         0 1 0 0    (+4)

(f)     0 0 1 0    (+2)                  0 0 1 0
      − 0 1 0 0    (+4)       ⇒        + 1 1 0 0
      ---------    ----                ---------    ----
                                         1 1 1 0    (−2)

(g)     0 1 1 0    (+6)                  0 1 1 0
      − 0 0 1 1    (+3)       ⇒        + 1 1 0 1
      ---------    ----                ---------    ----
                                         0 0 1 1    (+3)

(h)     1 0 0 1    (−7)                  1 0 0 1
      − 1 0 1 1    (−5)       ⇒        + 0 1 0 1
      ---------    ----                ---------    ----
                                         1 1 1 0    (−2)

(i)     1 0 0 1    (−7)                  1 0 0 1
      − 0 0 0 1    (+1)       ⇒        + 1 1 1 1
      ---------    ----                ---------    ----
                                         1 0 0 0    (−8)

(j)     0 0 1 0    (+2)                  0 0 1 0
      − 1 1 0 1    (−3)       ⇒        + 0 0 1 1
      ---------    ----                ---------    ----
                                         0 1 0 1    (+5)
```

Figure 1.6 2's-complement Add and Subtract operations.

is done in exactly the same manner for both positive and negative numbers. To form the 2's-complement of a number, form the bit complement of the number and add 1.

The simplicity of adding and subtracting signed numbers in 2's-complement representation is the reason why this number representation is used in modern computers. It might seem that the 1's-complement representation would be just as good as the 2's-complement system. However, although complementation is easy, the result obtained after an addition operation is not always correct. The carry-out, c_n, cannot be ignored. If $c_n = 0$, the result obtained is correct. If $c_n = 1$, then a 1 must be added to the result to make it correct. The need for this correction operation means that addition and subtraction cannot be implemented as conveniently in the 1's-complement system as in the 2's-complement system.

Sign Extension

We often need to represent a value given in a certain number of bits by using a larger number of bits. For a positive number, this is achieved by adding 0s to the left. For a negative number in 2's-complement representation, the leftmost bit, which indicates the sign of the number, is a 1. A longer number with the same value is obtained by replicating the sign bit to the left as many times as needed. To see why this is correct, examine the mod 16 circle of Figure 1.5*b*. Compare it to larger circles for the mod 32 or mod 64 cases. The representations for the values -1, -2, etc., are exactly the same, with 1s added to the left. In summary, to represent a signed number in 2's-complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called *sign extension*.

Overflow in Integer Arithmetic

Using 2's-complement representation, n bits can represent values in the range -2^{n-1} to $+2^{n-1} - 1$. For example, the range of numbers that can be represented by 4 bits is -8 through $+7$, as shown in Figure 1.3. When the actual result of an arithmetic operation is outside the representable range, an *arithmetic overflow* has occurred.

When adding unsigned numbers, a carry-out of 1 from the most significant bit position indicates that an overflow has occurred. However, this is not always true when adding signed numbers. For example, using 2's-complement representation for 4-bit signed numbers, if we add $+7$ and $+4$, the sum vector is 1011, which is the representation for -5, an incorrect result. In this case, the carry-out bit from the MSB position is 0. If we add -4 and -6, we get $0110 = +6$, also an incorrect result. In this case, the carry-out bit is 1. Hence, the value of the carry-out bit from the sign-bit position is not an indicator of overflow. Clearly, overflow may occur only if both summands have the same sign. The addition of numbers with different signs cannot cause overflow because the result is always within the representable range.

These observations lead to the following way to detect overflow when adding two numbers in 2's-complement representation. Examine the signs of the two summands and the sign of the result. When both summands have the same sign, an overflow has occurred when the sign of the sum is not the same as the signs of the summands.

When subtracting two numbers, the testing method needed for detecting overflow has to be modified somewhat; but it is still quite straightforward. See Problem 1.10.

1.4.2 FLOATING-POINT NUMBERS

Until now we have only considered integers, which have an implied binary point at the right end of the number, just after bit b_0. If we use a full word in a 32-bit word length computer to represent a signed integer in 2's-complement representation, the range of values that can be represented is -2^{31} to $+2^{31} - 1$. In decimal terms, this range is somewhat smaller than -10^{10} to $+10^{10}$.

The same 32-bit patterns can also be interpreted as fractions in the range -1 to $+1 - 2^{-31}$ if we assume that the implied binary point is just to the right of the sign bit; that is, between bit b_{31} and bit b_{30} at the left end of the 32-bit representation. In this case, the magnitude of the smallest fraction representable is approximately 10^{-10}.

Neither of these two *fixed-point* number representations has a range that is sufficient for many scientific and engineering calculations. For convenience, we would like to have a binary number representation that can easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In this case, the binary point is said to *float*, and the numbers are called *floating-point numbers.*

Since the position of the binary point in a floating-point number varies, it must be indicated explicitly in the representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, 3.7291×10^{-27}, -1.0341×10^{2}, -7.3000×10^{-14}, and so on. We say that these numbers have been given to 5 *significant digits* of precision. The *scale factors* 10^{23}, 10^{-27}, 10^{2}, and 10^{-14} indicate the actual position of the decimal point with respect to the significant digits. The same approach can be used to represent binary floating-point numbers in a computer, except that it is more appropriate to use 2 as the base of the scale factor. Because the base is fixed, it does not need to be given in the representation. The exponent may be positive or negative.

We conclude that a binary floating-point number can be represented by:

- a sign for the number
- some significant bits
- a signed scale factor exponent for an implied base of 2

An established international IEEE (Institute of Electrical and Electronics Engineers) standard for 32-bit floating-point number representation uses a sign bit, 23 significant bits, and 8 bits for a signed exponent of the scale factor, which has an implied base of 2. In decimal terms, the range of numbers represented is roughly $\pm 10^{-38}$ to $\pm 10^{38}$, which is adequate for most scientific and engineering calculations. The same IEEE standard also defines a 64-bit representation to accommodate more significant bits and more bits for the signed exponent, resulting in much higher precision and a much larger range of values.

Floating-point number representation and arithmetic operations on floating-point numbers are considered in detail in Chapter 9. Some of the commercial processors described in Appendices B to E include operations on floating-point numbers in their instruction sets and have processor registers dedicated to holding floating-point numbers.

1.5 CHARACTER REPRESENTATION

The most common encoding scheme for characters is ASCII (American Standard Code for Information Interchange). Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes as shown in Table 1.1. It is convenient to use an 8-bit *byte* to represent and store a character. The code occupies the low-order seven bits. The high-order bit is usually set to 0. Note that the codes for the alphabetic and numeric characters are in increasing sequential order when interpreted as unsigned binary numbers. This facilitates sorting operations on alphabetic and numeric data.

The low-order four bits of the ASCII codes for the decimal digits 0 to 9 are the first ten values of the binary number system. This 4-bit encoding is referred to as the *binary-coded decimal* (BCD) code.

1.6 PERFORMANCE

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes programs is affected by the design of its instruction set, its hardware and its software, including the operating system, and the technology in which the hardware is implemented. Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language. We do not describe the details of compilers or operating systems in this book. However, Chapter 4 provides an overview of software, including a discussion of the role of compilers and operating systems. This book concentrates on the design of instruction sets, along with memory, processor, and I/O hardware, and the organization of both small and large computers. Section 1.2.2 describes how caches can improve memory performance. Some performance aspects of instruction sets are discussed in Chapter 2. In this section, we give an overview of how performance is affected by technology, as well as processor and system organization.

1.6.1 TECHNOLOGY

The technology of Very Large Scale Integration (VLSI) that is used to fabricate the electronic circuits for a processor on a single chip is a critical factor in the speed of execution of machine instructions. The speed of switching between the 0 and 1 states in logic circuits is largely determined by the size of the transistors that implement the circuits. Smaller transistors switch faster. Advances in fabrication technology over several decades have reduced transistor sizes dramatically. This has two advantages: instructions can be executed faster, and more transistors can be placed on a chip, leading to more logic functionality and more memory storage capacity.

Table 1.1 The 7-bit ASCII code.

Bit positions	**Bit positions 654**							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	´	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	1	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	—	○	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgment
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACK	Acknowledgment	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

Bit positions of code format = $\boxed{6}\boxed{5}\boxed{4}\boxed{3}\boxed{2}\boxed{1}\boxed{0}$

1.6.2 Parallelism

Performance can be increased by performing a number of operations in parallel. Parallelism can be implemented on many different levels.

Instruction-level Parallelism

The simplest way to execute a sequence of instructions in a processor is to complete all steps of the current instruction before starting the steps of the next instruction. If we overlap the execution of the steps of successive instructions, total execution time will be reduced. For example, the next instruction could be fetched from memory at the same time that an arithmetic operation is being performed on the register operands of the current instruction. This form of parallelism is called *pipelining*. It is discussed in detail in Chapter 6.

Multicore Processors

Multiple processing units can be fabricated on a single chip. In technical literature, the term *core* is used for each of these processors. The term processor is then used for the complete chip. Hence, we have the terminology *dual-core*, *quad-core*, and *octo-core* processors for chips that have two, four, and eight cores, respectively.

Multiprocessors

Computer systems may contain many processors, each possibly containing multiple cores. Such systems are called *multiprocessors*. These systems either execute a number of different application tasks in parallel, or they execute subtasks of a single large task in parallel. All processors usually have access to all of the memory in such systems, and the term *shared-memory multiprocessor* is often used to make this clear. The high performance of these systems comes with much higher complexity and cost, arising from the use of multiple processors and memory units, along with more complex interconnection networks.

In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. The computers normally have access only to their own memory units. When the tasks they are executing need to share data, they do so by exchanging *messages* over a communication network. This property distinguishes them from shared-memory multiprocessors, leading to the name *message-passing multicomputers*.

Multiprocessors and multicomputers are described in Chapter 12.

1.7 Historical Perspective

Electronic digital computers as we know them today have been developed since the 1940s. A long, slow evolution of mechanical calculating devices preceded the development of electronic computers. Here, we briefly sketch the history of computer development. A more extensive coverage can be found in Hayes [1].

In the 300 years before the mid-1900s, a series of increasingly complex mechanical devices, constructed from gear wheels, levers, and pulleys, were used to perform the basic operations of addition, subtraction, multiplication, and division. Holes on punched cards were mechanically sensed and used to control the automatic sequencing of a list of calculations, which essentially provided a programming capability. These devices enabled the computation of complete mathematical tables of logarithms and trigonometric functions as approximated by polynomials. Output results were punched on cards or printed on paper. Electromechanical relay devices, such as those used in early telephone switching systems, provided the means for performing logic functions in computers built in the late 1930s and early 1940s.

During World War II, the first electronic computer was designed and built at the University of Pennsylvania, using the vacuum tube technology developed for radios and military radar equipment. Vacuum tube circuits were used to perform logic operations and to store data. This technology initiated the modern era of electronic digital computers.

Development of the technologies used to fabricate processors, memories, and I/O units of computers has been divided into four generations: the first generation, 1945 to 1955; the second generation, 1955 to 1965; the third generation, 1965 to 1975; and the fourth generation, 1975 to the present.

1.7.1 The First Generation

The key concept of a stored program was introduced at the same time as the development of the first electronic digital computer. Programs and their data were located in the same memory, as they are today. This facilitates changing existing programs and data or preparing and loading new programs and data. Assembly language was used to prepare programs and was translated into machine language for execution.

Basic arithmetic operations were performed in a few milliseconds, using vacuum tube technology to implement logic functions. This provided a 100- to 1000-fold increase in speed relative to earlier mechanical and electromechanical technology. Mercury delay-line memory was used at first. I/O functions were performed by devices similar to typewriters. Magnetic core memories and magnetic tape storage devices were also developed.

1.7.2 The Second Generation

The transistor was invented at AT&T Bell Laboratories in the late 1940s and quickly replaced the vacuum tube in implementing logic functions. This fundamental technology shift marked the start of the second generation. Magnetic core memories and magnetic drum storage devices were widely used in the second generation. Magnetic disk storage devices were developed in this generation. The earliest high-level languages, such as Fortran, were developed, making the preparation of application programs much easier. Compilers were developed to translate these high-level language programs into assembly language, which was then translated into executable machine-language form. IBM became a major computer manufacturer during this time.

1.7.3 The Third Generation

Texas Instruments and Fairchild Semiconductor developed the ability to fabricate many transistors on a single silicon chip, called integrated-circuit technology. This enabled faster and less costly processors and memory elements to be built. Integrated-circuit memories began to replace magnetic core memories. This technological development marked the beginning of the third generation. Other developments included the introduction of microprogramming, parallelism, and pipelining. Operating system software allowed efficient sharing of a computer system by several user programs. Cache and virtual memories were developed. Cache memory makes the main memory appear faster than it really is, and virtual memory makes it appear larger. System 360 mainframe computers from IBM and the line of PDP minicomputers from Digital Equipment Corporation were dominant commercial products of the third generation.

1.7.4 The Fourth Generation

By the early 1970s, integrated-circuit fabrication techniques had evolved to the point where complete processors and large sections of the main memory of small computers could be implemented on single chips. This marked the start of the fourth generation. Tens of thousands of transistors could be placed on a single chip, and the name Very Large Scale Integration (VLSI) was coined to describe this technology. A complete processor fabricated on a single chip became known as a microprocessor. Companies such as Intel, National Semiconductor, Motorola, Texas Instruments, and Advanced Micro Devices have been the driving forces of this technology. Current VLSI technology enables the integration of multiple processors (cores) and cache memories on a single chip.

A particular form of VLSI technology, called Field Programmable Gate Arrays (FPGAs), has allowed system developers to design and implement processor, memory, and I/O circuits on a single chip to meet the requirements of specific applications, especially in embedded computer systems. Sophisticated computer-aided-design tools make it possible to develop FPGA-based products quickly. Companies such as Altera and Xilinx provide this technology, along with the required software development systems.

Embedded computer systems, portable notebook computers, and versatile mobile telephone handsets are now in widespread use. Desktop personal computers and workstations interconnected by wired or wireless local area networks and the Internet, with access to database servers and search engines, provide a variety of powerful computing platforms.

Organizational concepts such as parallelism and hierarchical memories have evolved to produce the high-performance computing systems of today as the fourth generation has matured. Supercomputers and Grid computers, at the upper end of high-performance computing, are used for weather forecasting, scientific and engineering computations, and simulations.

1.8 Concluding Remarks

This chapter has introduced basic concepts about the structure of computers and their operation. Machine instructions and programs have been described briefly. The addition and subtraction of binary numbers has been explained. Much of the terminology needed to deal with these subjects has been defined. Subsequent chapters provide detailed explanations of these terms and concepts, with an emphasis on architecture and hardware.

1.9 Solved Problems

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 1.1 **Problem:** List the steps needed to execute the machine instruction

Load R2, LOC

in terms of transfers between the components shown in Figure 1.2 and some simple control commands. An overview of the steps needed is given in Section 1.3. Assume that the address of the memory location containing this instruction is initially in register PC.

Solution: The required steps are:

- Send the address of the instruction word from register PC to the memory and issue a Read control command.
- Wait until the requested word has been retrieved from the memory, then load it into register IR, where it is interpreted (decoded) by the control circuitry to determine the operation to be performed.
- Increment the contents of register PC to point to the next instruction in memory.
- Send the address value LOC from the instruction in register IR to the memory and issue a Read control command.
- Wait until the requested word has been retrieved from the memory, then load it into register R2.

Example 1.2 **Problem:** Quantify the effect on performance that results from the use of a cache in the case of a program that has a total of 500 instructions, including a 100-instruction loop that is executed 25 times. Determine the ratio of execution time without the cache to execution time with the cache. This ratio is called the *speedup*.

Assume that main memory accesses require 10 units of time and cache accesses require 1 unit of time. We also make the following further assumptions so that we can simplify calculations in order to easily illustrate the advantage of using a cache:

- Program execution time is proportional to the total amount of time needed to fetch instructions from either the main memory or the cache, with operand data accesses being ignored.
- Initially, all instructions are stored in the main memory, and the cache is empty.
- The cache is large enough to contain all of the loop instructions.

Solution: Execution time without the cache is

$$T = 400 \times 10 + 100 \times 10 \times 25 = 29{,}000$$

Execution time with the cache is

$$T_{cache} = 500 \times 10 + 100 \times 1 \times 24 = 7{,}400$$

Therefore, the speedup is

$$T/T_{cache} = 3.92$$

Problem: Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then perform addition and subtraction on each pair. Indicate whether or not overflow occurs for each case. **Example 1.3**

(*a*) 7 and 13

(*b*) −12 and 9

Solution: The conversion and operations are:

(*a*) $7_{10} = 00111_2$ and $13_{10} = 01101_2$

Adding these two positive numbers, we obtain 10100, which is a negative number. Therefore, overflow has occurred.

To subtract them, we first form the 2's-complement of 01101, which is 10011. Then we perform addition with 00111 to obtain 11010, which is -6_{10}, the correct answer.

(*b*) $-12_{10} = 10100_2$ and $9_{10} = 01001_2$

Adding these two numbers, we obtain $11101 = -3_{10}$, the correct answer.

To subtract them, we first form the 2's-complement of 01001, which is 10111. Then we perform addition of the two negative numbers 10100 and 10111 to obtain 01011, which is a positive number. Therefore, overflow has occurred.

Problems

1.1 **[E]** Repeat Example 1.1 for the machine instruction

```
Add     R4, R2, R3
```

which is discussed in Section 1.3.

1.2 **[E]** Repeat Example 1.1 for the machine instruction

```
Store     R4, LOC
```

which is discussed in Section 1.3.

1.3 **[M]** (*a*) Give a short sequence of machine instructions for the task "Add the contents of memory location A to those of location B, and place the answer in location C". Instructions

```
Load    Ri, LOC
```

and

```
Store    Ri, LOC
```

are the only instructions available to transfer data between the memory and the general-purpose registers. Add instructions are described in Section 1.3. Do not change the contents of either location A or B.

(*b*) Suppose that Move and Add instructions are available with the formats

```
Move    Location1, Location2
```

and

```
Add     Location1, Location2
```

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers. Is it possible to use fewer instructions of these types to accomplish the task in part (*a*)? If yes, give the sequence.

1.4 **[M]** (*a*) A program consisting of a total of 300 instructions contains a 50-instruction loop that is executed 15 times. The processor contains a cache, as described in Section 1.2.2. Fetching and executing an instruction that is in the main memory requires 20 time units. If the instruction is found in the cache, fetching and executing it requires only 2 time units. Ignoring operand data accesses, calculate the ratio of program execution time without the cache to execution time with the cache. This ratio is called the *speedup* due to the use of the cache. Assume that the cache is initially empty, that it is large enough to hold the loop, and that the program starts with all instructions in the main memory.

(*b*) Generalize part (*a*) by replacing the constants 300, 50, 15, 20, and 2 with the variables w, x, y, m, and c. Develop an expression for speedup.

(*c*) For the values $w = 300$, $x = 50$, $m = 20$, and $c = 2$ what value of y results in a speedup of 5?

(*d*) Consider the form of the expression for speedup developed in part (*b*). What is the upper limit on speedup as the number of loop iterations, *y*, becomes larger and larger?

1.5 **[M]** (*a*) A processor cache is discussed in Section 1.2.2. Suppose that execution time for a program is proportional to instruction fetch time. Assume that fetching an instruction from the cache takes 1 time unit, but fetching it from the main memory takes 10 time units. Also, assume that a requested instruction is found in the cache with probability 0.96. Finally, assume that if an instruction is not found in the cache it must first be fetched from the main memory into the cache and then fetched from the cache to be executed. Compute the ratio of program execution time without the cache to program execution time with the cache. This ratio is called the *speedup* resulting from the presence of the cache.

(*b*) If the size of the cache is doubled, assume that the probability of not finding a requested instruction there is cut in half. Repeat part (*a*) for a doubled cache size.

1.6 **[E]** Extend Figure 1.4 to incorporate both possibilities for a carry-in (0 or 1) to each of the four cases shown in the figure. Specify both the sum and carry-out bits for each of the eight new cases.

1.7 **[M]** Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case.

(*a*) 4 and 11

(*b*) 6 and 14

(*c*) −13 and 12

(*d*) −4 and 8

(*e*) −2 and −9

(*f*) −9 and −14

1.8 **[M]** Repeat Problem 1.7 for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.

1.9 **[E]** A memory byte location contains the pattern 01010011. What decimal value does this pattern represent when interpreted as a binary number? What does it represent as an ASCII code?

1.10 **[E]** A way to detect overflow when adding two 2's-complement numbers is given at the end of Section 1.4.1. State how to detect overflow when subtracting two such numbers.

References

1. J. P. Hayes, *Computer Architecture and Organization*, 3rd Ed., McGraw-Hill, New York, 1998.