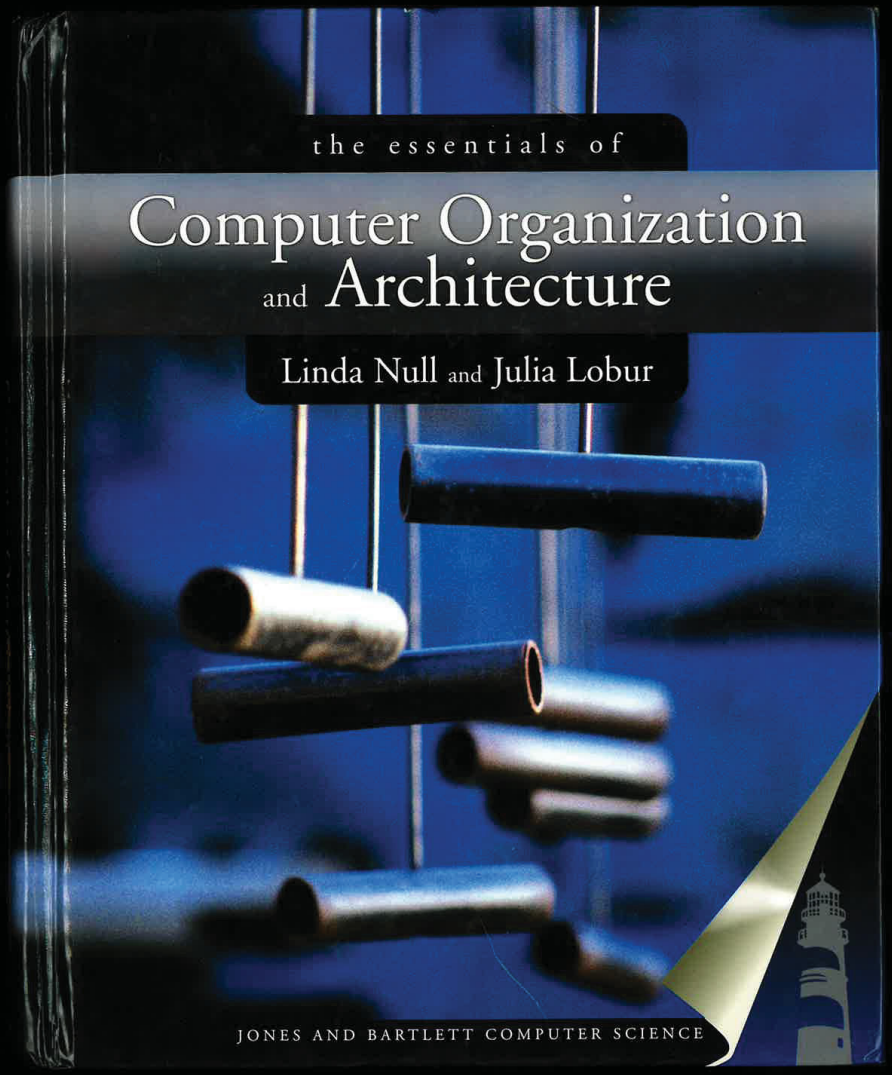

the essentials of
Computer Organization
and Architecture
Linda Null and Julia Lobur
JONES AND BARTLETT COMPUTER SCIENCE

USAA 1027


the essentials of
Computer Organization
and Architecture
Linda Null
Pennsylvania State University
Julia Lobur
Pennsylvania State University
JONES AND BARTLETT PUBLISHERS
Sudbury, Massachusetts
BOSTON TORONTO LONDON SINGAPORE

World Headquarters
Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers
Canada
2406 Nikanna Road
Mississauga, ON L5C 2W6
CANADA

Jones and Bartlett Publishers International
Barb House, Barb Mews
London W6 7PA
UK

Copyright © 2003 by Jones and Bartlett Publishers, Inc.

Cover image © David Buffington / Getty Images

Illustrations based upon and drawn from art provided by Julia Lobur

Library of Congress Cataloging-in-Publication Data

Null, Linda.
The essentials of computer organization and architecture / Linda Null, Julia Lobur.
p. cm.
ISBN 0-7637-0444-X
1. Computer organization. 2. Computer architecture. I. Lobur, Julia. II. Title.

QA76.9.C643 N85 2003
004.2'2—dc21

2002040576

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or any information storage or retrieval system, without written permission from the copyright owner.

Chief Executive Officer: Clayton Jones
Chief Operating Officer: Don W. Jones, Jr.
Executive V.P. and Publisher: Robert W. Holland, Jr.
V.P., Design and Production: Anne Spencer
V.P., Manufacturing and Inventory Control: Therese Bräuer
Director, Sales and Marketing: William Kane
Editor-in-Chief, College: J. Michael Stranz
Production Manager: Amy Rose
Senior Marketing Manager: Nathan Schultz
Associate Production Editor: Karen C. Ferreira
Associate Editor: Theresa DiDonato
Production Assistant: Jenny McIsaac
Cover Design: Kristin E. Ohlin
Composition: Northeast Compositors
Text Design: Anne Flanagan
Printing and Binding: Courier Westford
Cover Printing: Jaguar Advanced Graphics

This book was typeset in Quark 4.1 on a Macintosh G4. The font families used were Times, Mixage, and Prestige Elite. The first printing was printed on 45# Highland Plus.

Printed in the United States of America

07 06 05 04 03 10 9 8 7 6 5 4 3 2

In memory of my father, Merrill Cornell, a pilot and man of endless talent and courage, who taught me that when we step into the unknown, we either find solid ground, or we learn to fly.

—L. M. N.

To the loving memory of my mother, Anna J. Surowski, who made all things possible for her girls.

—J. M. L.

PREFACE

TO THE STUDENT

This is a book about computer organization and architecture. It focuses on the function and design of the various components necessary to process information digitally. We present computing systems as a series of layers, starting with low-level hardware and progressing to higher-level software, including assemblers and operating systems. These levels constitute a hierarchy of virtual machines. The study of computer organization focuses on this hierarchy and the issues involved with how we partition the levels and how each level is implemented. The study of computer architecture focuses on the interface between hardware and software, and emphasizes the structure and behavior of the system. The majority of information contained in this textbook is devoted to computer hardware, and computer organization and architecture, and their relationship to software performance.

Students invariably ask, "Why, if I am a computer science major, must I learn about computer hardware? Isn't that for computer engineers? Why do I care what the inside of a computer looks like?" As computer users, we probably do not have to worry about this any more than we need to know what our car looks like under the hood in order to drive it. We can certainly write high-level language programs without understanding how these programs execute; we can use various application packages without understanding how they really work. But what happens when the program we have written needs to be faster and more

efficient, or the application we are using doesn't do precisely what we want? As computer scientists, we need a basic understanding of the computer system itself in order to rectify these problems.

There is a fundamental relationship between the computer hardware and the many aspects of programming and software components in computer systems. In order to write good software, it is very important to understand the computer system as a whole. Understanding hardware can help you explain the mysterious errors that sometimes creep into your programs, such as the infamous segmentation fault or bus error. The level of knowledge about computer organization and computer architecture that a high-level programmer must have depends on the task the high-level programmer is attempting to complete.

For example, to write compilers, you must understand the particular hardware to which you are compiling. Some of the ideas used in hardware (such as pipelining) can be adapted to compilation techniques, thus making the compiler faster and more efficient. To model large, complex, real-world systems, you must understand how floating-point arithmetic should, and does, work (which are not necessarily the same thing). To write device drivers for video, disks, or other I/O devices, you need a good understanding of I/O interfacing and computer architecture in general. If you want to work on embedded systems, which are usually very resource-constrained, you must understand all of the time, space, and price tradeoffs. To do research on, and make recommendations for, hardware systems, networks, or specific algorithms, you must acquire an understanding of benchmarking and then learn how to present performance results adequately. Before buying hardware, you need to understand benchmarking and all of the ways in which others can *manipulate* the performance results to "prove" that one system is better than another. Regardless of our particular area of expertise, as computer scientists, it is imperative that we understand how hardware interacts with software.

You may also be wondering why a book with the word *essentials* in its title is so large. The reason is twofold. First, the subject of computer organization is expansive and it grows by the day. Second, there is little agreement as to which topics from within this burgeoning sea of information are truly essential and which are just helpful to know. In writing this book, one goal was to provide a concise text compliant with the computer architecture curriculum guidelines jointly published by the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic Engineers (IEEE). These guidelines encompass the subject matter that experts agree constitutes the "essential" core body of knowledge relevant to the subject of computer organization and architecture.

We have augmented the ACM/IEEE recommendations with subject matter that we feel is useful—if not essential—to your continuing computer science studies and to your professional advancement. The topics we feel will help you in your continuing computer science studies include operating systems, compilers, database management, and data communications. Other subjects are included because they will help you understand how actual systems work in real life.

We hope that you find reading this book an enjoyable experience, and that you take time to delve deeper into some of the material that we have presented. It is our intention that this book will serve as a useful reference long after your formal course is complete. Although we give you a substantial amount of information, it is only a foundation upon which you can build throughout the remainder of your studies and your career. Successful computer professionals continually add to their knowledge about how computers work. Welcome to the start of your journey.

TO THE INSTRUCTOR

About the Book

This book is the outgrowth of two computer science organization and architecture classes taught at The Pennsylvania State University Harrisburg campus. As the computer science curriculum evolved, we found it necessary not only to modify the material taught in the courses but also to condense the courses from a two-semester sequence into a three credit, one-semester course. Many other schools have also recognized the need to compress material in order to make room for emerging topics. This new course, as well as this textbook, is primarily for computer science majors, and is intended to address the topics in computer organization and architecture with which computer science majors must be familiar. This book not only integrates the underlying principles in these areas, but it also introduces and motivates the topics, providing the breadth necessary for majors, while providing the depth necessary for continuing studies in computer science.

Our primary objective in writing this book is to change the way computer organization and architecture are typically taught. A computer science major should leave a computer organization and architecture class with not only an understanding of the important general concepts on which the digital computer is founded, but also with a comprehension of how those concepts apply to the real world. These concepts should transcend vendor-specific terminology and design; in fact, students should be able to take concepts given in the specific and translate to the generic and vice versa. In addition, students must develop a firm foundation for further study in the major.

The title of our book, *The Essentials of Computer Organization and Architecture,* is intended to convey that the topics presented in the text are those for which every computer science major should have exposure, familiarity, or mastery. We do not expect students using our textbook to have complete mastery of all topics presented. It is our firm belief, however, that there are certain topics that must be mastered; there are those topics for which students must have a definite familiarity; and there are certain topics for which a brief introduction and exposure are adequate.

We do not feel that concepts presented in sufficient depth can be learned by studying general principles in isolation. We therefore present the topics as an inte-

grated set of solutions, not simply a collection of individual pieces of information. We feel our explanations, examples, exercises, tutorials, and simulators all combine to provide the student with a total learning experience that exposes the inner workings of a modern digital computer at the appropriate level.

We have written this textbook in an informal style, omitting unnecessary jargon, writing clearly and concisely, and avoiding unnecessary abstraction, in hopes of increasing student enthusiasm. We have also broadened the range of topics typically found in a first-level architecture book to include system software, a brief tour of operating systems, performance issues, alternative architectures, and a concise introduction to networking, as these topics are intimately related to computer hardware. Like most books, we have chosen an architectural model, but it is one that we have designed with simplicity in mind.

Relationship to Computing Curricula 2001

In December of 2001, the ACM/IEEE Joint Task Force unveiled the 2001 Computing Curricula (CC-2001). These new guidelines represent the first major revision since the very popular Computing Curricula 1991. CC-2001 represents several major changes from CC-1991, but we are mainly concerned with those that address computer organization and computer architecture. CC-1991 suggested approximately 59 lecture hours for architecture (defined as both organization and architecture and labeled AR), including the following topics: digital logic, digital systems, machine-level representation of data, assembly-level machine organization, memory system organization and architecture, interfacing and communication, and alternative architectures. The latest release of CC-2001 (available at www.computer.org/education/cc2001/) reduces architecture coverage to 36 core hours, including digital logic and digital systems (3 hours), machine-level representation of data (3 hours), assembly-level machine organization (9 hours), memory system organization and architecture (5 hours), interfacing and communication (3 hours), functional organization (7 hours), and multiprocessing and alternative architectures (3 hours). In addition, CC-2001 suggests including performance enhancements and architectures for networks and distributed systems as part of the architecture and organization module for CC-2001. We are pleased, after completely revising our course and writing this textbook, that our new material is in direct correlation with the ACM/IEEE 2001 Curriculum guidelines for computer organization and architecture as follows:

AR1. Digital logic and digital systems (core): Chapters 1 and 3
AR2. Machine-level representation of data (core): Chapter 2
AR3. Assembly-level machine organization (core): Chapters 4, 5 and 6
AR4. Memory system organization and architecture (core): Chapter 6
AR5. Interfacing and communication (core): Chapter 7
AR6. Functional organization (core): Chapters 4 and 5
AR7. Multiprocessing and alternative architectures (core): Chapter 9

AR8. Performance enhancements (elective): Chapters 9 and 10
AR9. Architecture for networks and distributed systems (elective): Chapter 11

Why another text?

No one can deny there is a plethora of textbooks for teaching computer organization and architecture already on the market. In our 25-plus years of teaching these courses, we have used many very good textbooks. However, each time we have taught the course, the content has evolved, and, eventually, we discovered we were writing significantly more course notes to bridge the gap between the material in the textbook and the material we deemed necessary to present in our classes. We found that our course material was migrating from a computer engineering approach to organization and architecture toward a computer science approach to these topics. When the decision was made to fold the organization class and the architecture class into one course, we simply could not find a textbook that covered the material we felt was necessary for our majors, written from a computer science point of view, written without machine-specific terminology, and designed to motivate the topics before covering them.

In this textbook, we hope to convey the spirit of design used in the development of modern computing systems and what impact this has on computer science students. Students, however, must have a strong understanding of the basic concepts before they can understand and appreciate the non-tangible aspects of design. Most organization and architecture textbooks present a similar subset of technical information regarding these basics. We, however, pay particular attention to the level at which the information should be covered, and to presenting that information in the context that has relevance for computer science students. For example, throughout this book, when concrete examples are necessary, we offer examples for personal computers, enterprise systems, and mainframes, as these are the types of systems most likely to be encountered. We avoid the "PC bias" prevalent in similar books in the hope that students will gain an appreciation for the differences, similarities, and the roles various platforms play within today's automated infrastructures. Too often, textbooks forget that motivation is, perhaps, the single most important key in learning. To that end, we include many real-world examples, while attempting to maintain a balance between theory and application.

Features

We have included many features in this textbook to emphasize the various concepts in computer organization and architecture, and to make the material more accessible to students. Some of the features are listed below:

- *Sidebars.* These sidebars include interesting tidbits of information that go a step beyond the main focus of the chapter, thus allowing readers to delve further into the material.

- *Real-World Examples.* We have integrated the textbook with examples from real life to give students a better understanding of how technology and techniques are combined for practical purposes.
- *Chapter Summaries.* These sections provide brief yet concise summaries of the main points in each chapter.
- *Further Reading.* These sections list additional sources for those readers who wish to investigate any of the topics in more detail, and contain references to definitive papers and books related to the chapter topics.
- *Review Questions.* Each chapter contains a set of review questions designed to ensure that the reader has a firm grasp on the material.
- *Chapter Exercises.* Each chapter has a broad selection of exercises to reinforce the ideas presented. More challenging exercises are marked with an asterisk.
- *Answers to Selected Exercises.* To ensure students are on the right track, we provide answers to representative questions from each chapter. Questions with answers in the back of the text are marked with a blue diamond.
- *Special "Focus On" Sections.* These sections provide additional information for instructors who may wish to cover certain concepts, such as Kmaps and input/output, in more detail. Additional exercises are provided for these sections as well.
- *Appendix.* The appendix provides a brief introduction or review of data structures, including topics such as stacks, linked lists, and trees.
- *Glossary.* An extensive glossary includes brief definitions of all key terms from the chapters.
- *Index.* An exhaustive index is provided with this book, with multiple cross-references, to make finding terms and concepts easier for the reader.

About the Authors

We bring to this textbook not only 25-plus years of combined teaching experience, but also 20 years of industry experience. Our combined efforts therefore stress the underlying principles of computer organization and architecture, and how these topics relate in practice. We include real-life examples to help students appreciate how these fundamental concepts are applied in the world of computing.

Linda Null received a Ph.D. in Computer Science from Iowa State University in 1991, an M.S. in Computer Science from Iowa State University in 1989, an M.S. in Computer Science Education from Northwest Missouri State University in 1983, an M.S. in Mathematics Education from Northwest Missouri State University in 1980, and a B.S. in Mathematics and English from Northwest Missouri State University in 1977. She has been teaching mathematics and computer science for over 25 years and is currently the Computer Science graduate program coordinator at The Pennsylvania State University Harrisburg campus, where she has been a member of the faculty since 1995. Her areas of interest include computer organization and architecture, operating systems, and computer security.

Julia Lobur has been a practitioner in the computer industry for over 20 years. She has held positions as a systems consultant, a staff programmer/analyst, a systems and network designer, and a software development manager, in addition to part-time teaching duties.

Prerequisites

The typical background necessary for a student using this textbook includes a year of programming experience using a high-level procedural language. Students are also expected to have taken a year of college-level mathematics (calculus or discrete mathematics), as this textbook assumes and incorporates these mathematical concepts. This book assumes no prior knowledge of computer hardware.

A computer organization and architecture class is customarily a prerequisite for an undergraduate operating systems class (students must know about the memory hierarchy, concurrency, exceptions, and interrupts), compilers (students must know about instruction sets, memory addressing, and linking), networking (students must understand the hardware of a system before attempting to understand the network that ties these components together), and of course, any advanced architecture class. This text covers the topics necessary for these courses.

General Organization and Coverage

Our presentation of concepts in this textbook is an attempt at a concise, yet thorough, coverage of the topics we feel are essential for the computer science major. We do not feel the best way to do this is by "compartmentalizing" the various topics; therefore, we have chosen a structured, yet integrated approach where each topic is covered in the context of the entire computer system.

As with many popular texts, we have taken a bottom-up approach, starting with the digital logic level and building to the application level that students should be familiar with before starting the class. The text is carefully structured so that the reader understands one level before moving on to the next. By the time the reader reaches the application level, all of the necessary concepts in computer organization and architecture have been presented. Our goal is to allow the students to tie the hardware knowledge covered in this book to the concepts learned in their introductory programming classes, resulting in a complete and thorough picture of how hardware and software fit together. Ultimately, the extent of hardware understanding has a significant influence on software design and performance. If students can build a firm foundation in hardware fundamentals, this will go a long way toward helping them to become better computer scientists.

The concepts in computer organization and architecture are integral to many of the everyday tasks that computer professionals perform. To address the numerous areas in which a computer professional should be educated, we have taken a high-level look at computer architecture, providing low-level coverage only when deemed necessary for an understanding of a specific concept. For example, when discussing ISAs, many hardware-dependent issues are introduced in the context

of different case studies to both differentiate and reinforce the issues associated with ISA design.

The text is divided into eleven chapters and an appendix as follows:

- **Chapter 1** provides a historical overview of computing in general, pointing out the many milestones in the development of computing systems, and allowing the reader to visualize how we arrived at the current state of computing. This chapter introduces the necessary terminology, the basic components in a computer system, the various logical levels of a computer system, and the von Neumann computer model. It provides a high-level view of the computer system, as well as the motivation and necessary concepts for further study.
- **Chapter 2** provides thorough coverage of the various means computers use to represent both numerical and character information. Addition, subtraction, multiplication and division are covered once the reader has been exposed to number bases and the typical numeric representation techniques, including one's complement, two's complement, and BCD. In addition, EBCDIC, ASCII, and Unicode character representations are addressed. Fixed- and floating-point representation are also introduced. Codes for data recording and error detection and correction are covered briefly.
- **Chapter 3** is a classic presentation of digital logic and how it relates to Boolean algebra. This chapter covers both combinational and sequential logic in sufficient detail to allow the reader to understand the logical makeup of more complicated MSI (medium scale integration) circuits (such as decoders). More complex circuits, such as buses and memory, are also included. We have included optimization and Kmaps in a special "Focus On" section.
- **Chapter 4** illustrates basic computer organization and introduces many fundamental concepts, including the fetch-decode-execute cycle, the data path, clocks and buses, register transfer notation, and of course, the CPU. A very simple architecture, MARIE, and its ISA are presented to allow the reader to gain a full understanding of the basic architectural organization involved in program execution. MARIE exhibits the classical von Neumann design, and includes a program counter, an accumulator, an instruction register, 4096 bytes of memory, and two addressing modes. Assembly language programming is introduced to reinforce the concepts of instruction format, instruction mode, data format, and control that are presented earlier. This is not an assembly language textbook and was not designed to provide a practical course in assembly language programming. The primary objective in introducing assembly is to further the understanding of computer architecture in general. However, a simulator for MARIE is provided so assembly language programs can be written, assembled, and run on the MARIE architecture. The two methods of control, hardwiring and microprogramming, are introduced and compared in this chapter. Finally, Intel and MIPS architectures are compared to reinforce the concepts in the chapter.
- **Chapter 5** provides a closer look at instruction set architectures, including instruction formats, instruction types, and addressing modes. Instruction-level pipelining is introduced as well. Real-world ISAs (including Intel, MIPS, and Java) are presented to reinforce the concepts presented in the chapter.
- **Chapter 6** covers basic memory concepts, such as RAM and the various memory devices, and also addresses the more advanced concepts of the memory hierarchy, including cache memory and virtual memory. This chapter gives a thorough presentation of direct mapping, associative mapping, and set-associative mapping techniques for cache. It also provides a detailed look at overlays, paging and segmentation, TLBs, and the various algorithms and devices associated with each. A tutorial and simulator for this chapter is available on the book's website.
- **Chapter 7** provides a detailed overview of I/O fundamentals, bus communication and protocols, and typical external storage devices, such as magnetic and optical disks, as well as the various formats available for each. DMA, programmed I/O, and interrupts are covered as well. In addition, various techniques for exchanging information between devices are introduced. RAID architectures are covered in detail, and various data compression formats are introduced.
- **Chapter 8** discusses the various programming tools available (such as compilers and assemblers) and their relationship to the architecture of the machine on which they are run. The goal of this chapter is to tie the programmer's view of a computer system with the actual hardware and architecture of the underlying machine. In addition, operating systems are introduced, but only covered in as much detail as applies to the architecture and organization of a system (such as resource use and protection, traps and interrupts, and various other services).
- **Chapter 9** provides an overview of alternative architectures that have emerged in recent years. RISC, Flynn's Taxonomy, parallel processors, instruction-level parallelism, multiprocessors, interconnection networks, shared memory systems, cache coherence, memory models, superscalar machines, neural networks, systolic architectures, dataflow computers, and distributed architectures are covered. Our main objective in this chapter is to help the reader realize we are not limited to the von Neumann architecture, and to force the reader to consider performance issues, setting the stage for the next chapter.
- **Chapter 10** addresses various performance analysis and management issues. The necessary mathematical preliminaries are introduced, followed by a discussion of MIPS, FLOPS, benchmarking, and various optimization issues with which a computer scientist should be familiar, including branch prediction, speculative execution, and loop optimization.
- **Chapter 11** focuses on network organization and architecture, including network components and protocols. The OSI model and TCP/IP suite are introduced in the context of the Internet. This chapter is by no means intended to be comprehensive. The main objective is to put computer architecture in the correct context relative to network architecture.

An appendix on data structures is provided for those situations where students may need a brief introduction or review of such topics as stacks, queues, and linked lists.

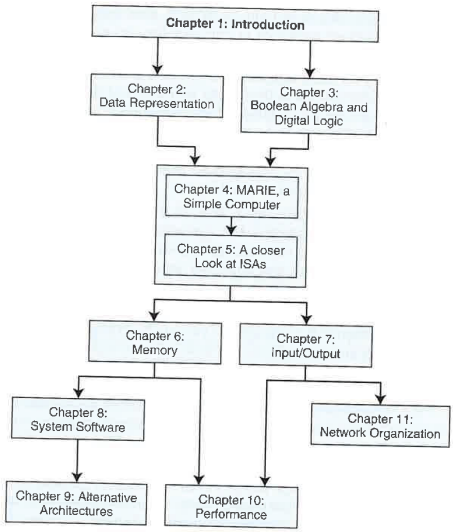


FIGURE P.1 Prerequisite Relationship Among Chapters

The sequencing of the chapters is such that they can be taught in the given numerical order. However, an instructor can modify the order to better fit a given curriculum if necessary. Figure P.1 shows the prerequisite relationships that exist between various chapters.

Intended Audience

This book was originally written for an undergraduate class in computer organization and architecture for computer science majors. Although specifically directed toward computer science majors, the book does not preclude its use by IS and IT majors.

This book contains more than sufficient material for a typical one-semester (14 week, 42 lecture hours) course; however, all of the material in the book cannot be mastered by the average student in a one-semester class. If the instructor

plans to cover all topics in detail, a two-semester sequence would be optimal. The organization is such that an instructor can cover the major topic areas at different levels of depth, depending on the experience and needs of the students. Table P.1 gives the instructor an idea of the length of time required to cover the topics, and also lists the corresponding levels of accomplishment for each chapter.

It is our intention that this book will serve as a useful reference long after the formal course is complete.

Support Materials

A textbook is a fundamental tool in learning, but its effectiveness is greatly enhanced by supplemental materials and exercises, which emphasize the major concepts, provide immediate feedback to the reader, and motivate understanding through repetition. We have, therefore, created the following ancillary materials for *The Essentials of Computer Organization and Architecture*:

- *Instructor's Manual*. This manual contains answers to exercises and sample exam questions. In addition, it provides hints on teaching various concepts and trouble areas often encountered by students.
- *Lecture Slides*. These slides contain lecture material appropriate for a one-semester course in computer organization and architecture.
- *Figures and Tables*. For those who wish to prepare their own lecture materials, we provide the figures and tables in downloadable form.

	One Semester (42 Hours)		Two Semesters (84 Hours)	
Chapter	Lecture Hours	Expected Level	Lecture Hours	Expected Level
1	3	Mastery	3	Mastery
2	6	Mastery	6	Mastery
3	6	Mastery	6	Mastery
4	6	Familiarity	10	Mastery
5	3	Familiarity	8	Mastery
6	5	Familiarity	9	Mastery
7	2	Familiarity	6	Mastery
8	2	Exposure	7	Mastery
9	3	Familiarity	9	Mastery
10	3	Exposure	9	Mastery
11	3	Exposure	11	Mastery

TABLE P.1 Suggested Lecture Hours

- *Memory Tutorial and Simulator*. This package allows students to apply the concepts on cache and virtual memory.
- *MARIE Simulator*. This package allows students to assemble and run MARIE programs.
- *Tutorial Software*. Other tutorial software is provided for various concepts in the book.
- *The Companion website*. All software, slides, and related materials can be downloaded from the book's website:

  `http://computerscience.jbpub.com/ECOA`

The exercises, sample exam problems, and solutions have been tested in numerous classes. The Instructor's Manual, which includes suggestions for teaching the various chapters in addition to answers for the book's exercises, suggested programming assignments, and sample example questions, is available to instructors who adopt the book. (Please contact your Jones and Bartlett Publishers Representative at 1-800-832-0034 for access to this area of the web site.)

The Instructional Model: MARIE

In a computer organization and architecture book, the choice of architectural model affects the instructor as well as the students. If the model is too complicated, both the instructor and the students tend to get bogged down in details that really have no bearing on the concepts being presented in class. Real architectures, although interesting, often have far too many peculiarities to make them usable in an introductory class. To make things even more complicated, real architectures change from day to day. In addition, it is difficult to find a book incorporating a model that matches the local computing platform in a given department, noting that the platform, too, may change from year to year.

To alleviate these problems, we have designed our own simple architecture, MARIE, specifically for pedagogical use. MARIE (Machine Architecture that is Really Intuitive and Easy) allows students to learn the essential concepts of computer organization and architecture, including assembly language, without getting caught up in the unnecessary and confusing details that exist in real architectures. Despite its simplicity, it simulates a functional system. The MARIE machine simulator, MarieSim, has a user-friendly GUI that allows students to: (1) create and edit source code; (2) assemble source code into machine object code; (3) run machine code; and, (4) debug programs.

Specifically, MarieSim has the following features:

- Support for the MARIE assembly language introduced in Chapter 4
- An integrated text editor for program creation and modification
- Hexadecimal machine language object code
- An integrated debugger with single step mode, break points, pause, resume, and register and memory tracing

- A graphical memory monitor displaying the 4096 addresses in MARIE's memory
- A graphical display of MARIE's registers
- Highlighted instructions during program execution
- User-controlled execution speed
- Status messages
- User-viewable symbol tables
- An interactive assembler that lets the user correct any errors and reassemble automatically, without changing environments
- Online help
- Optional core dumps, allowing the user to specify the memory range
- Frames with sizes that can be modified by the user
- A small learning curve, allowing students to learn the system quickly

MarieSim was written in the Java™ language so that the system would be portable to any platform for which a Java™ Virtual Machine (JVM) is available. Students of Java may wish to look at the simulator's source code, and perhaps even offer improvements or enhancements to its simple functions.

Figure P.2, the MarieSim Graphical Environment, shows the graphical environment of the MARIE machine simulator. The screen consists of four parts: the menu bar, the central monitor area, the memory monitor, and the message area.

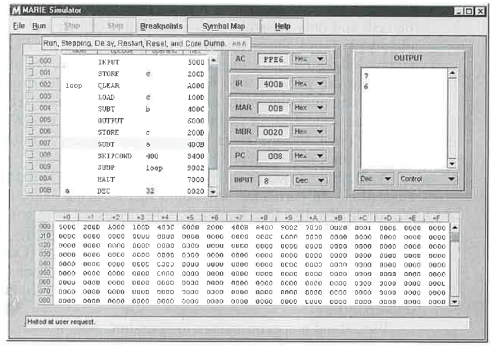


FIGURE P.2 The MarieSim Graphical Environment

Menu options allow the user to control the actions and behavior of the MARIE Machine Simulator system. These options include loading, starting, stopping, setting breakpoints, and pausing programs that have been written in MARIE assembly language.

The MARIE Simulator illustrates the process of assembly, loading, and execution, all in one simple environment. Users can see assembly language statements directly from their programs, along with the corresponding machine code (hexadecimal) equivalents. The addresses of these instructions are indicated as well, and users can view any portion of memory at any time. Highlighting is used to indicate the initial loading address of a program in addition to the currently executing instruction while a program runs. The graphical display of the registers and memory allows the student to see how the instructions cause the values within the registers and memory to change.

If You Find an Error

We have attempted to make this book as technically accurate as possible, but even though the manuscript has been through numerous proof readings, errors have a way of escaping detection. We would greatly appreciate hearing from readers who find any errors that need correcting. Your comments and suggestions are always welcome by sending email to ECOA@jbpub.com.

Credits and Acknowledgments

Few books are entirely the result of one or two people's unaided efforts, and this one is no exception. We now realize that writing a textbook is a formidable task and only possible with a combined effort, and we find it impossible to adequately thank those who have made this book possible. If, in the following acknowledgements, we inadvertently omit anyone, we humbly apologize.

All of the students who have taken our computer organization and architecture classes over the years have provided invaluable feedback regarding what works and what doesn't when covering the various topics in the classes. We particularly thank those classes that used preliminary versions of the textbook for their tolerance and diligence in finding errors.

A number of people have read the manuscript in detail and provided useful suggestions. In particular, we would like to thank Mary Creel and Hans Royer. We would also like to acknowledge the reviewers who gave their time and effort, in addition to many good suggestions, to ensure a quality text, including: Victor Clincy (Kennesaw State University); Robert Franks (Central College); Karam Mossaad (The University of Texas at Austin); Michael Schulte (University of Missouri, St. Louis); Peter Smith (CSU Northridge); Xiaobo Zhou (Wayne State University).

We extend a special thanks to Karishma Rao for her time and effort in producing a quality memory software module.

The publishing team at Jones and Bartlett has been wonderful to work with, and each member deserves a special thanks, including Amy Rose, Theresa DiDonato, Nathan Schultz, and J. Michael Stranz.

I, Linda Null, would personally like to thank my husband, Tim Wahls, for his patience while living life as a "book widower," for listening and commenting with frankness about the book's contents, for doing such an extraordinary job with all of the cooking, and for putting up with the almost daily compromises necessitated by my writing this book. I consider myself amazingly lucky to be married to such a wonderful man. I extend my heart-felt thanks to my mentor, Merry McDonald, who taught me the value and joys of learning and teaching, and doing both with integrity. Lastly, I would like to express my deepest gratitude to Julia Lobur, as without her, this book and its accompanying software would not be a reality.

I, Julia Lobur, am deeply indebted to my partner, Marla Cattermole, for making possible my work on this book through her forbearance and fidelity. She has nurtured my body through her culinary delights and my spirit through her wisdom. She has taken up my slack in many ways while working hard at her own career and her own advanced degree. I would also like to convey my profound gratitude to Linda Null: Foremost for her unsurpassed devotion to the field of computer science education and dedication to her students, and consequently, for giving me the opportunity to share with her the ineffable experience of textbook authorship.

Contents

CHAPTER 1 **Introduction** 1

1.1 **Overview** 1

1.2 **The Main Components of a Computer** 3

1.3 **An Example System: Wading through the Jargon** 4

1.4 **Standards Organizations** 10

1.5 **Historical Development** 12

1.5.1 Generation Zero: Mechanical Calculating Machines (1642–1945) 12

1.5.2 The First Generation: Vacuum Tube Computers (1945–1953) 14

1.5.3 The Second Generation: Transistorized Computers (1954–1965) 19

1.5.4 The Third Generation: Integrated Circuit Computers (1965–1980) 21

1.5.5 The Fourth Generation: VLSI Computers (1980–????) 22

1.5.6 Moore's Law 24

1.6 **The Computer Level Hierarchy** 25

1.7 **The von Neumann Model** 27

1.8 **Non-von Neumann Models** 29

Chapter Summary 31

Further Reading 31

References 32

Review of Essential Terms and Concepts 33

Exercises 34

CHAPTER 2 Data Representation in Computer Systems 37

2.1 Introduction 37
2.2 Positional Numbering Systems 38
2.3 Decimal to Binary Conversions 38
2.3.1 Converting Unsigned Whole Numbers 39
2.3.2 Converting Fractions 41
2.3.3 Converting between Power-of-Two Radices 44
2.4 Signed Integer Representation 44
2.4.1 Signed Magnitude 44
2.4.2 Complement Systems 49
2.5 Floating-Point Representation 55
2.5.1 A Simple Model 56
2.5.2 Floating-Point Arithmetic 58
2.5.3 Floating-Point Errors 59
2.5.4 The IEEE-754 Floating-Point Standard 61
2.6 Character Codes 62
2.6.1 Binary-Coded Decimal 62
2.6.2 EBCDIC 63
2.6.3 ASCII 63
2.6.4 Unicode 65
2.7 Codes for Data Recording and Transmission 67
2.7.1 Non-Return-to-Zero Code 68
2.7.2 Non-Return-to-Zero-Invert Encoding 69
2.7.3 Phase Modulation (Manchester Coding) 70
2.7.4 Frequency Modulation 70
2.7.5 Run-Length-Limited Code 71
2.8 Error Detection and Correction 73
2.8.1 Cyclic Redundancy Check 73
2.8.2 Hamming Codes 77
2.8.3 Reed-Solomon 82
Chapter Summary 83
Further Reading 84
References 85
Review of Essential Terms and Concepts 85
Exercises 86

CHAPTER 3 Boolean Algebra and Digital Logic 93

3.1 Introduction 93
3.2 Boolean Algebra 94
3.2.1 Boolean Expressions 94
3.2.2 Boolean Identities 96
3.2.3 Simplification of Boolean Expressions 98
3.2.4 Complements 99
3.2.5 Representing Boolean Functions 100
3.3 Logic Gates 102
3.3.1 Symbols for Logic Gates 102
3.3.2 Universal Gates 103
3.3.3 Multiple Input Gates 104
3.4 Digital Components 105
3.4.1 Digital Circuits and Their Relationship to Boolean Algebra 105
3.4.2 Integrated Circuits 106
3.5 Combinational Circuits 106
3.5.1 Basic Concepts 107
3.5.2 Examples of Typical Combinational Circuits 107
3.6 Sequential Circuits 113
3.6.1 Basic Concepts 114
3.6.2 Clocks 114
3.6.3 Flip-Flops 115
3.6.4 Examples of Sequential Circuits 117
3.7 Designing Circuits 120
Chapter Summary 121
Further Reading 122
References 123
Review of Essential Terms and Concepts 123
Exercises 124

Focus on Karnaugh Maps 130
3A.1 Introduction 130
3A.2 Description of Kmaps and Terminology 131
3A.3 Kmap Simplification for Two Variables 133
3A.4 Kmap Simplification for Three Variables 134
3A.5 Kmap Simplification for Four Variables 137
3A.6 Don't Care Conditions 140
3A.7 Summary 141
Exercises 141

CHAPTER 4 MARIE: An Introduction to a Simple Computer 145

4.1 Introduction 145
4.1.1 CPU Basics and Organization 145
4.1.2 The Bus 147
4.1.3 Clocks 151
4.1.4 The Input/Output Subsystem 153
4.1.5 Memory Organization and Addressing 153
4.1.6 Interrupts 156
4.2 MARIE 157
4.2.1 The Architecture 157
4.2.2 Registers and Buses 159
4.2.3 The Instruction Set Architecture 160
4.2.4 Register Transfer Notation 163
4.3 Instruction Processing 166
4.3.1 The Fetch-Decode-Execute Cycle 166
4.3.2 Interrupts and I/O 166
4.4 A Simple Program 169
4.5 A Discussion on Assemblers 170
4.5.1 What Do Assemblers Do? 170
4.5.2 Why Use Assembly Language? 173
4.6 Extending Our Instruction Set 174
4.7 A Discussion on Decoding: Hardwired vs. Microprogrammed Control 179
4.8 Real-World Examples of Computer Architectures 182
4.8.1 Intel Architectures 183
4.8.2 MIPS Architectures 187
Chapter Summary 189
Further Reading 190
References 191
Review of Essential Terms and Concepts 192
Exercises 193

CHAPTER 5 A Closer Look at Instruction Set Architectures 199

5.1 Introduction 199
5.2 Instruction Formats 199
5.2.1 Design Decisions for Instruction Sets 200

5.2.2 Little versus Big Endian 201
5.2.3 Internal Storage in the CPU: Stacks versus Registers 203
5.2.4 Number of Operands and Instruction Length 204
5.2.5 Expanding Opcodes 208
5.3 Instruction Types 210
5.4 Addressing 211
5.4.1 Data Types 211
5.4.2 Address Modes 212
5.5 Instruction-Level Pipelining 214
5.6 Real-World Examples of ISAs 219
5.6.1 Intel 220
5.6.2 MIPS 220
5.6.3 Java Virtual Machine 221
Chapter Summary 225
Further Reading 226
References 227
Review of Essential Terms and Concepts 228
Exercises 229

CHAPTER 6 Memory 233

6.1 Memory 233
6.2 Types of Memory 233
6.3 The Memory Hierarchy 235
6.3.1 Locality of Reference 237
6.4 Cache Memory 237
6.4.1 Cache Mapping Schemes 239
6.4.2 Replacement Policies 247
6.4.3 Effective Access Time and Hit Ratio 248
6.4.4 When Does Caching Break Down? 249
6.4.5 Cache Write Policies 249
6.5 Virtual Memory 250
6.5.1 Paging 251
6.5.2 Effective Access Time Using Paging 258
6.5.3 Putting It All Together: Using Cache, TLBs, and Paging 259
6.5.4 Advantages and Disadvantages of Paging and Virtual Memory 259
6.5.5 Segmentation 262
6.5.6 Paging Combined with Segmentation 263

6.6 A Real-World Example of Memory Management 263
Chapter Summary 264
Further Reading 265
References 266
Review of Essential Terms and Concepts 266
Exercises 267

CHAPTER 7 Input/Output and Storage Systems 273

7.1 Introduction 273
7.2 Amdahl's Law 274
7.3 I/O Architectures 275
7.3.1 I/O Control Methods 276
7.3.2 I/O Bus Operation 280
7.3.3 Another Look at Interrupt-Driven I/O 283
7.4 Magnetic Disk Technology 286
7.4.1 Rigid Disk Drives 288
7.4.2 Flexible (Floppy) Disks 292
7.5 Optical Disks 293
7.5.1 CD-ROM 294
7.5.2 DVD 297
7.5.3 Optical Disk Recording Methods 298
7.6 Magnetic Tape 299
7.7 RAID 301
7.7.1 RAID Level 0 302
7.7.2 RAID Level 1 303
7.7.3 RAID Level 2 303
7.7.4 RAID Level 3 304
7.7.5 RAID Level 4 305
7.7.6 RAID Level 5 306
7.7.7 RAID Level 6 307
7.7.8 Hybrid RAID Systems 308
7.8 Data Compression 309
7.8.1 Statistical Coding 311
7.8.2 Ziv-Lempel (LZ) Dictionary Systems 318
7.8.3 GIF and PNG Compression 322
7.8.4 JPEG Compression 323

Chapter Summary 328
Further Reading 328
References 329
Review of Essential Terms and Concepts 330
Exercises 332
Focus on Selected Disk Storage Implementations 335
7A.1 Introduction 335
7A.2 Data Transmission Modes 335
7A.3 SCSI 338
7A.4 Storage Area Networks 350
7A.5 Other I/O Connections 352
7A.6 Summary 354
Exercises 354

CHAPTER 8 System Software 357

8.1 Introduction 357
8.2 Operating Systems 358
8.2.1 Operating Systems History 359
8.2.2 Operating System Design 364
8.2.3 Operating System Services 366
8.3 Protected Environments 370
8.3.1 Virtual Machines 371
8.3.2 Subsystems and Partitions 374
8.3.3 Protected Environments and the Evolution of Systems Architectures 376
8.4 Programming Tools 378
8.4.1 Assemblers and Assembly 378
8.4.2 Link Editors 381
8.4.3 Dynamic Link Libraries 382
8.4.4 Compilers 384
8.4.5 Interpreters 388
8.5 Java: All of the Above 389
8.6 Database Software 395
8.7 Transaction Managers 401
Chapter Summary 403
Further Reading 404

References 405
Review of Essential Terms and Concepts 406
Exercises 407

CHAPTER 9 Alternative Architectures 411

9.1 Introduction 411
9.2 RISC Machines 412
9.3 Flynn's Taxonomy 417
9.4 Parallel and Multiprocessor Architectures 421
9.4.1 Superscalar and VLIW 422
9.4.2 Vector Processors 424
9.4.3 Interconnection Networks 425
9.4.4 Shared Memory Multiprocessors 430
9.4.5 Distributed Computing 434
9.5 Alternative Parallel Processing Approaches 435
9.5.1 Dataflow Computing 435
9.5.2 Neural Networks 438
9.5.3 Systolic Arrays 441
Chapter Summary 442
Further Reading 443
References 443
Review of Essential Terms and Concepts 445
Exercises 446

CHAPTER 10 Performance Measurement and Analysis 451

10.1 Introduction 451
10.2 The Basic Computer Performance Equation 452
10.3 Mathematical Preliminaries 453
10.3.1 What the Means Mean 454
10.3.2 The Statistics and Semantics 459
10.4 Benchmarking 461
10.4.1 Clock Rate, MIPS, and FLOPS 462
10.4.2 Synthetic Benchmarks: Whetstone, Linpack, and Dhrystone 464
10.4.3 Standard Performance Evaluation Cooperation Benchmarks 465
10.4.4 Transaction Performance Council Benchmarks 469
10.4.5 System Simulation 476

10.5 CPU Performance Optimization 477
10.5.1 Branch Optimization 477
10.5.2 Use of Good Algorithms and Simple Code 480
10.6 Disk Performance 484
10.6.1 Understanding the Problem 484
10.6.2 Physical Considerations 485
10.6.3 Logical Considerations 486
Chapter Summary 492
Further Reading 493
References 494
Review of Essential Terms and Concepts 495
Exercises 495

CHAPTER 11 Network Organization and Architecture 501

11.1 Introduction 501
11.2 Early Business Computer Networks 501
11.3 Early Academic and Scientific Networks: The Roots and Architecture of the Internet 502
11.4 Network Protocols I: ISO/OSI Protocol Unification 506
11.4.1 A Parable 507
11.4.2 The OSI Reference Model 508
11.5 Network Protocols II: TCP/IP Network Architecture 512
11.5.1 The IP Layer for Version 4 512
11.5.2 The Trouble with IP Version 4 516
11.5.3 Transmission Control Protocol 520
11.5.4 The TCP Protocol at Work 521
11.5.5 IP Version 6 525
11.6 Network Organization 530
11.6.1 Physical Transmission Media 530
11.6.2 Interface Cards 535
11.6.3 Repeaters 536
11.6.4 Hubs 537
11.6.5 Switches 537
11.6.6 Bridges and Gateways 538
11.6.7 Routers and Routing 539
11.7 High-Capacity Digital Links 548
11.7.1 The Digital Hierarchy 549

11.7.2 ISDN 553
11.7.3 Asynchronous Transfer Mode 556
11.8 A Look at the Internet 557
11.8.1 Ramping on to the Internet 558
11.8.2 Ramping up the Internet 565
Chapter Summary 566
Further Reading 566
References 568
Review of Essential Terms and Concepts 568
Exercises 570

APPENDIX A **Data Structures and the Computer** 575
A.1 Introduction 575
A.2 Fundamental Structures 575
A.2.1 Arrays 575
A.2.2 Queues and Linked Lists 577
A.2.3 Stacks 578
A.3 Trees 581
A.4 Network Graphs 587
Summary 590
Further Reading 590
References 590
Exercises 591

Glossary 595

Answers and Hints for Selected Exercises 633

Index 647

"Computing is not about computers anymore. It is about living. . . . We have seen computers move out of giant air-conditioned rooms into closets, then onto desktops, and now into our laps and pockets. But this is not the end. . . . Like a force of nature, the digital age cannot be denied or stopped. . . . The information superhighway may be mostly hype today, but it is an understatement about tomorrow. It will exist beyond people's wildest predictions. . . . We are not waiting on any invention. It is here. It is now. It is almost genetic in its nature, in that each generation will become more digital than the preceding one."

—Nicholas Negroponte, professor of media technology at MIT

CHAPTER 1 Introduction

1.1 OVERVIEW

Dr. Negroponte is among many who see the computer revolution as if it were a force of nature. This force has the potential to carry humanity to its digital destiny, allowing us to conquer problems that have eluded us for centuries, as well as all of the problems that emerge as we solve the original problems. Computers have freed us from the tedium of routine tasks, liberating our collective creative potential so that we can, of course, build bigger and better computers.

As we observe the profound scientific and social changes that computers have brought us, it is easy to start feeling overwhelmed by the complexity of it all. This complexity, however, emanates from concepts that are fundamentally very simple. These simple ideas are the ones that have brought us where we are today, and are the foundation for the computers of the future. To what extent they will survive in the future is anybody's guess. But today, they are the foundation for all of computer science as we know it.

Computer scientists are usually more concerned with writing complex program algorithms than with designing computer hardware. Of course, if we want our algorithms to be useful, a computer eventually has to run them. Some algorithms are so complicated that they would take too long to run on today's systems. These kinds of algorithms are considered *computationally infeasible*. Certainly, at the current rate of innovation, some things that are infeasible today could be feasible tomorrow, but it seems that no matter how big or fast computers become, someone will think up a problem that will exceed the reasonable limits of the machine.

To understand why an algorithm is infeasible, or to understand why the implementation of a feasible algorithm is running too slowly, you must be able to see the program from the computer's point of view. You must understand what makes a computer system tick before you can attempt to optimize the programs that it runs. Attempting to optimize a computer system without first understanding it is like attempting to tune your car by pouring an elixir into the gas tank: You'll be lucky if it runs at all when you're finished.

Program optimization and system tuning are perhaps the most important motivations for learning how computers work. There are, however, many other reasons. For example, if you want to write compilers, you must understand the hardware environment within which the compiler will function. The best compilers leverage particular hardware features (such as pipelining) for greater speed and efficiency.

If you ever need to model large, complex, real-world systems, you will need to know how floating-point arithmetic should work as well as how it really works in practice. If you wish to design peripheral equipment or the software that drives peripheral equipment, you must know every detail of how a particular computer deals with its input/output (I/O). If your work involves embedded systems, you need to know that these systems are usually resource-constrained. Your understanding of time, space, and price tradeoffs, as well as I/O architectures, will be essential to your career.

All computer professionals should be familiar with the concepts of benchmarking and be able to interpret and present the results of benchmarking systems. People who perform research involving hardware systems, networks, or algorithms find benchmarking techniques crucial to their day-to-day work. Technical managers in charge of buying hardware also use benchmarks to help them buy the best system for a given amount of money, keeping in mind the ways in which performance benchmarks can be manipulated to imply results favorable to particular systems.

The preceding examples illustrate the idea that a fundamental relationship exists between computer hardware and many aspects of programming and software components in computer systems. Therefore, regardless of our area of expertise, as computer scientists, it is imperative that we understand how hardware interacts with software. We must become familiar with how various circuits and components fit together to create working computer systems. We do this through the study of *computer organization*. Computer organization addresses issues such as control signals (how the computer is controlled), signaling methods, and memory types. It encompasses all physical aspects of computer systems. It helps us to answer the question: How does a computer work?

The study of *computer architecture*, on the other hand, focuses on the structure and behavior of the computer system and refers to the logical aspects of system implementation as seen by the programmer. Computer architecture includes many elements such as instruction sets and formats, operation codes, data types, the number and types of registers, addressing modes, main memory access methods, and various I/O mechanisms. The architecture of a system directly affects the logical execution of programs. Studying computer architecture helps us to answer the question: How do I design a computer?

The computer architecture for a given machine is the combination of its hardware components plus its *instruction set architecture* (*ISA*). The ISA is the agreed-upon interface between all the software that runs on the machine and the hardware that executes it. The ISA allows you to talk to the machine.

The distinction between computer organization and computer architecture is not clear-cut. People in the fields of computer science and computer engineering hold differing opinions as to exactly which concepts pertain to computer organization and which pertain to computer architecture. In fact, neither computer organization nor computer architecture can stand alone. They are interrelated and interdependent. We can truly understand each of them only after we comprehend both of them. Our comprehension of computer organization and architecture ultimately leads to a deeper understanding of computers and computation—the heart and soul of computer science.

1.2 THE MAIN COMPONENTS OF A COMPUTER

Although it is difficult to distinguish between the ideas belonging to computer organization and those ideas belonging to computer architecture, it is impossible to say where hardware issues end and software issues begin. Computer scientists design algorithms that usually are implemented as programs written in some computer language, such as Java or C. But what makes the algorithm run? Another algorithm, of course! And another algorithm runs that algorithm, and so on until you get down to the machine level, which can be thought of as an algorithm implemented as an electronic device. Thus, modern computers are actually implementations of algorithms that execute other algorithms. This chain of nested algorithms leads us to the following principle:

> **Principle of Equivalence of Hardware and Software:** *Anything that can be done with software can also be done with hardware, and anything that can be done with hardware can also be done with software.*[1]

A special-purpose computer can be designed to perform any task, such as word processing, budget analysis, or playing a friendly game of Tetris. Accordingly, programs can be written to carry out the functions of special-purpose computers, such as the embedded systems situated in your car or microwave. There are times when a simple embedded system gives us much better performance than a complicated computer program, and there are times when a program is the preferred approach. The Principle of Equivalence of Hardware and Software tells us that we have a choice. Our knowledge of computer organization and architecture will help us to make the best choice.

[1]What this principle does not address is the speed with which the equivalent tasks are carried out. Hardware implementations are almost always faster.

We begin our discussion of computer hardware by looking at the components necessary to build a computing system. At the most basic level, a computer is a device consisting of three pieces:

1. A processor to interpret and execute programs
2. A memory to store both data and programs
3. A mechanism for transferring data to and from the outside world

We discuss these three components in detail as they relate to computer hardware in the following chapters.

Once you understand computers in terms of their component parts, you should be able to understand what a system is doing at all times and how you could change its behavior if so desired. You might even feel like you have a few things in common with it. This idea is not as far-fetched as it appears. Consider how a student sitting in class exhibits the three components of a computer: the student's brain is the processor, the notes being taken represent the memory, and the pencil or pen used to take notes is the I/O mechanism. But keep in mind that your abilities far surpass those of any computer in the world today, or any that can be built in the foreseeable future.

1.3 AN EXAMPLE SYSTEM: WADING THROUGH THE JARGON

This book will introduce you to some of the vocabulary that is specific to computers. This jargon can be confusing, imprecise, and intimidating. We believe that with a little explanation, we can clear the fog.

For the sake of discussion, we have provided a facsimile computer advertisement (see Figure 1.1). The ad is typical of many in that it bombards the reader with phrases such as "64MB SDRAM," "64-bit PCI sound card" and "32KB L1 cache." Without having a handle on such terminology, you would be hard-pressed to know whether the stated system is a wise buy, or even whether the system is able to serve your needs. As we progress through this book, you will learn the concepts behind these terms.

Before we explain the ad, however, we need to discuss something even more basic: the measurement terminology you will encounter throughout your study of computers.

It seems that every field has its own way of measuring things. The computer field is no exception. So that computer people can tell each other how big something is, or how fast something is, they must use the same units of measure. When we want to talk about how big some computer thing is, we speak of it in terms of thousands, millions, billions, or trillions of characters. The prefixes for terms are given in the left side of Figure 1.2. In computing systems, as you shall see, powers of 2 are often more important than powers of 10, but it is easier for people to understand powers of 10. Therefore, these prefixes are given in both powers of 10 and powers of 2. Because 1,000 is close in value to 2^{10} (1,024), we can approximate powers of 10 by powers of 2. Prefixes used in system metrics are often applied where the underlying base system is base 2, not base 10. For example, a

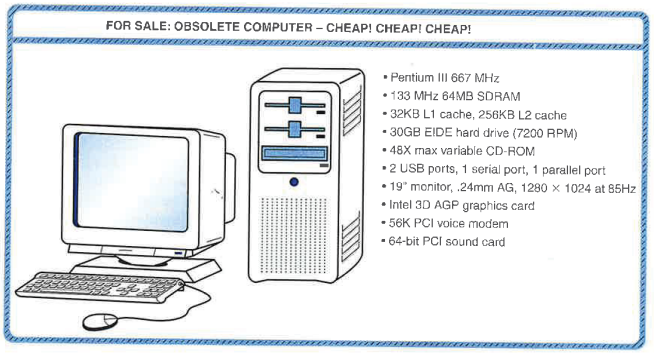


FIGURE 1.1 A Typical Computer Advertisement

Kilo- (k)	(1 thousand = $10^3 \approx 2^{10}$)	Milli- (m)	(1 thousandth = $10^{-3} \approx 2^{-10}$)
Mega- (M)	(1 million = $10^6 \approx 2^{20}$)	Micro- (μ)	(1 millionth = $10^{-6} \approx 2^{-20}$)
Giga- (G)	(1 billion = $10^9 \approx 2^{30}$)	Nano- (n)	(1 billionth = $10^{-9} \approx 2^{-30}$)
Tera- (T)	(1 trillion = $10^{12} \approx 2^{40}$)	Pico- (p)	(1 trillionth = $10^{-12} \approx 2^{-40}$)
Peta- (P)	(1 quadrillion = $10^{15} \approx 2^{50}$)	Femto- (f)	(1 quadrillionth = $10^{-15} \approx 2^{-50}$)

FIGURE 1.2 Common Prefixes Associated with Computer Organization and Architecture

kilobyte (1KB) of memory is *typically* 1,024 bytes of memory rather than 1,000 bytes of memory. However, a 1GB disk drive might actually be 1 billion bytes instead of 2^{30} (approximately 1.07 billion). You should always read the manufacturer's fine print just to make sure you know exactly what 1K, 1KB, or 1G represents.

When we want to talk about how fast something is, we speak in terms of fractions of a second—usually thousandths, millionths, billionths, or trillionths. Prefixes for these metrics are given in the right-hand side of Figure 1.2. Notice that the fractional prefixes have exponents that are the reciprocal of the prefixes on the left side of the figure. Therefore, if someone says to you that an operation requires a microsecond to complete, you should also understand that a million of those operations could take place in one second. When you need to talk about how many of these things happen in a second, you would use the prefix *mega-*. When you need to talk about how fast the operations are performed, you would use the prefix *micro-*.

Now to explain the ad: The microprocessor is the part of a computer that actually executes program instructions; it is the brain of the system. The microprocessor in the ad is a Pentium III, operating at 667MHz. Every computer system contains a clock that keeps the system synchronized. The clock sends electrical pulses simultaneously to all main components, ensuring that data and instructions will be where they're supposed to be, when they're supposed to be there. The number of pulsations emitted each second by the clock is its frequency. Clock frequencies are measured in cycles per second, or *hertz*. Because computer system clocks generate millions of pulses per second, we say that they operate in the *megahertz (MHz)* range. Many computers today operate in the *gigahertz* range, generating billions of pulses per second. And because nothing much gets done in a computer system without microprocessor involvement, the frequency rating of the microprocessor is crucial to overall system speed. The microprocessor of the system in our advertisement operates at 667 million cycles per second, so the seller says that it runs at 667MHz.

The fact that this microprocessor runs at 667MHz, however, doesn't necessarily mean that it can execute 667 million instructions every second, or, equivalently, that every instruction requires 1.5 nanoseconds to execute. Later in this book, you will see that each computer instruction requires a fixed number of cycles to execute. Some instructions require one clock cycle; however, most instructions require more than one. The number of instructions per second that a microprocessor can actually execute is *proportionate* to its clock speed. The number of clock cycles required to carry out a particular machine instruction is a function of both the machine's organization and its architecture.

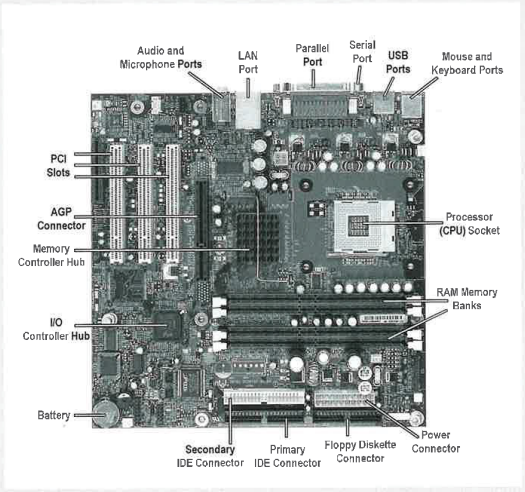
The next thing that we see in the ad is "133MHz 64MB SDRAM." The 133MHz refers to the speed of the system *bus*, which is a group of wires that moves data and instructions to various places within the computer. Like the microprocessor, the speed of the bus is also measured in MHz. Many computers have a special local bus for data that supports very fast transfer speeds (such as those required by video). This local bus is a high-speed pathway that connects memory directly to the processor. Bus speed ultimately sets the upper limit on the system's information-carrying capability.

The system in our advertisement also boasts a memory capacity of 64 megabytes (MB), or about 64 million characters. Memory capacity not only determines the size of the programs that you can run, but also how many programs you can run at the same time without bogging down the system. Your application or operating system manufacturer will usually recommend how much memory you'll need to run their products. (Sometimes these recommendations can be hilariously conservative, so be careful whom you believe!)

In addition to memory size, our advertised system provides us with a memory type, *SDRAM*, short for *synchronous dynamic random access memory*. SDRAM is much faster than conventional (nonsynchronous) memory because it can synchronize itself with a microprocessor's bus. At this writing, SDRAM bus synchronization is possible only with buses running at 200MHz and below. Newer memory technologies such as RDRAM (Rambus DRAM) and SLDRAM (SyncLink DRAM) are required for systems running faster buses.

A Look Inside a Computer

Have you even wondered what the inside of a computer really looks like? The example computer described in this section gives a good overview of the components of a modern PC. However, opening a computer and attempting to find and identify the various pieces can be frustrating, even if you are familiar with the components and their functions.



Courtesy of Intel Corporation

If you remove the cover on your computer, you will no doubt first notice a big metal box with a fan attached. This is the power supply. You will also see various drives, including a hard drive, and perhaps a floppy drive and CD-ROM or DVD drive. There are many integrated circuits — small, black rectangular boxes with legs attached. You will also notice electrical pathways, or buses, in the system. There are printed circuit boards (expansion cards) that plug into sockets on the motherboard, the large board at the bottom of a standard desktop PC or on the side of a PC configured as a tower or mini-tower. The motherboard is the printed circuit board that connects all of the components in the

computer, including the CPU, and RAM and ROM memory, as well as an assortment of other essential components. The components on the motherboard tend to be the most difficult to identify. Above you see an Intel D850 motherboard with the more important components labeled.

The I/O ports at the top of the board allow the computer to communicate with the outside world. The I/O controller hub allows all connected devices to function without conflict. The PCI slots allow for expansion boards belonging to various PCI devices. The AGP connector is for plugging in the AGP graphics card. There are two RAM memory banks and a memory controller hub. There is no processor plugged into this motherboard, but we see the socket where the CPU is to be placed. All computers have an internal battery, as seen at the lower left-hand corner. This motherboard has two IDE connector slots, and one floppy disk controller. The power supply plugs into the power connector.

A note of caution regarding looking inside the box: There are many safety considerations involved with removing the cover for both you and your computer. There are many things you can do to minimize the risks. First and foremost, make sure the computer is turned off. Leaving it plugged in is often preferred, as this offers a path for static electricity. Before opening your computer and touching anything inside, you should make sure you are properly grounded so static electricity will not damage any components. Many of the edges, both on the cover and on the circuit boards, can be sharp, so take care when handling the various pieces. Trying to jam misaligned cards into sockets can damage both the card and the motherboard, so be careful if you decide to add a new card or remove and reinstall an existing one.

The next line in the ad, "32KB L1 cache, 256KB L2 cache" also describes a type of memory. In Chapter 6, you will learn that no matter how fast a bus is, it still takes "a while" to get data from memory to the processor. To provide even faster access to data, many systems contain a special memory called *cache*. The system in our advertisement has two kinds of cache. Level 1 cache (L1) is a small, fast memory cache that is built into the microprocessor chip and helps speed up access to frequently used data. Level 2 cache (L2) is a collection of fast, built-in memory chips situated between the microprocessor and main memory. Notice that the cache in our system has a capacity of kilobytes (KB), which is much smaller than main memory. In Chapter 6 you will learn how cache works, and that a bigger cache isn't always better.

On the other hand, everyone agrees that the more fixed disk capacity you have, the better off you are. The advertised system has 30GB, which is fairly impressive. The storage capacity of a fixed (or hard) disk is not the only thing to consider, however. A large disk isn't very helpful if it is too slow for its host system. The computer in our ad has a hard drive that rotates at 7200 RPM (revolutions per minute). To the knowledgeable reader, this indicates (but does not state

outright) that this is a fairly fast drive. Usually disk speeds are stated in terms of the number of milliseconds required (on average) to access data on the disk, in addition to how fast the disk rotates.

Rotational speed is only one of the determining factors in the overall performance of a disk. The manner in which it connects to—or *interfaces* with—the rest of the system is also important. The advertised system uses a disk interface called *EIDE*, or *enhanced integrated drive electronics*. EIDE is a cost-effective hardware interface for mass storage devices. EIDE contains special circuits that allow it to enhance a computer's connectivity, speed, and memory capability. Most EIDE systems share the main system bus with the processor and memory, so the movement of data to and from the disk is also dependent on the speed of the system bus.

Whereas the system bus is responsible for all data movement internal to the computer, *ports* allow movement of data to and from devices external to the computer. Our ad speaks of three different ports with the line, "2 USB ports, 1 serial port, 1 parallel port." Most desktop computers come with two kinds of data ports: serial ports and parallel ports. Serial ports transfer data by sending a series of electrical pulses across one or two data lines. Parallel ports use at least eight data lines, which are energized simultaneously to transmit data. Our advertised system also comes equipped with a special serial connection called a *USB* (*universal serial bus*) port. USB is a popular external bus that supports *Plug-and-Play* (the ability to configure devices automatically) as well as *hot plugging* (the ability to add and remove devices while the computer is running).

Some systems augment their main bus with dedicated I/O buses. *Peripheral Component Interconnect* (*PCI*) is one such I/O bus that supports the connection of multiple peripheral devices. PCI, developed by the Intel Corporation, operates at high speeds and also supports Plug-and-Play. There are two PCI devices mentioned in the ad. The PCI modem allows the computer to connect to the Internet. (We discuss modems in detail in Chapter 11.) The other PCI device is a sound card, which contains components needed by the system's stereo speakers. You will learn more about different kinds of I/O, I/O buses, and disk storage in Chapter 7.

After telling us about the ports in the advertised system, the ad supplies us with some specifications for the monitor by saying, "19" monitor, .24mm AG, 1280 × 1024 at 85Hz." Monitors have little to do with the speed or efficiency of a computer system, but they have great bearing on the comfort of the user. The monitor in the ad supports a *refresh rate* of 85Hz. This means that the image displayed on the monitor is repainted 85 times a second. If the refresh rate is too slow, the screen may exhibit an annoying jiggle or wavelike behavior. The eyestrain caused by a wavy display makes people tire easily; some people may even experience headaches after periods of prolonged use. Another source of eyestrain is poor resolution. A higher-resolution monitor makes for better viewing and finer graphics. Resolution is determined by the *dot pitch* of the monitor, which is the distance between a dot (or pixel) and the closest dot of the same color. The smaller the dot, the sharper the image. In this case, we have a 0.24 millimeter

(mm) dot pitch supported by an *AG* (*aperture grill*) display. Aperture grills direct the electron beam that paints the monitor picture on the phosphor coating inside the glass of the monitor. AG monitors produce crisper images than the older shadow mask technology. This monitor is further supported by an *AGP* (*accelerated graphics port*) graphics card. This is a graphics interface designed by Intel specifically for 3D graphics.

In light of the preceding discussion, you may be wondering why monitor dot pitch can't be made arbitrarily small to give picture perfect resolution. The reason is that the refresh rate is dependent on the dot pitch. Refreshing 100 dots, for example, requires more time than refreshing 50 dots. A smaller dot pitch requires more dots to cover the screen. The more dots to refresh, the longer it takes for each refresh cycle. Experts recommend a refresh rate of at least 75Hz. The 85Hz refresh rate of the advertised monitor is better than the minimum recommendation by 10Hz (about 13%).

Although we cannot delve into all of the brand-specific components available, after completing this book, you should understand the concept of how most computer systems operate. This understanding is important for casual users as well as experienced programmers. As a user, you need to be aware of the strengths and limitations of your computer system so you can make informed decisions about applications and thus use your system more effectively. As a programmer, you need to understand exactly how your system hardware functions so you can write effective and efficient programs. For example, something as simple as the algorithm your hardware uses to map main memory to cache and the method used for memory interleaving can have a tremendous impact on your decision to access array elements in row versus column-major order.

Throughout this book, we investigate both large and small computers. Large computers include mainframes (enterprise-class servers) and supercomputers. Small computers include personal systems, workstations and handheld devices. We will show that regardless of whether they carry out routine chores or perform sophisticated scientific tasks, the components of these systems are very similar. We also visit some architectures that lie outside what is now the mainstream of computing. We hope that the knowledge that you gain from this book will ultimately serve as a springboard for your continuing studies within the vast and exciting fields of computer organization and architecture.

1.4 STANDARDS ORGANIZATIONS

Suppose you decide that you'd like to have one of those nifty new .28mm dot pitch AG monitors. You figure that you can shop around a bit to find the best price. You make a few phone calls, surf the Web, and drive around town until you find the one that gives you the most for your money. From your experience, you know that you can buy your monitor anywhere and it will probably work fine on your system. You can make this assumption because computer equipment manufacturers have agreed to comply with connectivity and operational specifications established by a number of government and industry organizations.

Some of these standards-setting organizations are ad-hoc trade associations or consortia made up of industry leaders. Manufacturers know that by establishing common guidelines for a particular type of equipment, they can market their products to a wider audience than if they came up with separate—and perhaps incompatible—specifications.

Some standards organizations have formal charters and are recognized internationally as the definitive authority in certain areas of electronics and computers. As you continue your studies in computer organization and architecture, you will encounter specifications formulated by these groups, so you should know something about them.

The *Institute of Electrical and Electronic Engineers* (*IEEE*) is an organization dedicated to the advancement of the professions of electronic and computer engineering. The IEEE actively promotes the interests of the worldwide engineering community by publishing an array of technical literature. The IEEE also sets standards for various computer components, signaling protocols, and data representation, to name only a few areas of its involvement. The IEEE has a democratic, albeit convoluted, procedure established for the creation of new standards. Its final documents are well respected and usually endure for several years before requiring revision.

The *International Telecommunications Union* (*ITU*) is based in Geneva, Switzerland. The ITU was formerly known as the *Comité Consultatif International Télégraphique et Téléphonique*, or the International Consultative Committee on Telephony and Telegraphy. As its name implies, the ITU concerns itself with the interoperability of telecommunications systems, including telephone, telegraph, and data communication systems. The telecommunications arm of the ITU, the ITU-T, has established a number of standards that you will encounter in the literature. You will see these standards prefixed by ITU-T or the group's former initials, CCITT.

Many countries, including the European Community, have commissioned umbrella organizations to represent their interests within various international groups. The group representing the United States is the *American National Standards Institute* (*ANSI*). Great Britain has its *British Standards Institution* (*BSI*) in addition to having a voice on *CEN* (*Comite Europeen de Normalisation*), the European committee for standardization.

The *International Organization for Standardization* (*ISO*) is the entity that coordinates worldwide standards development, including the activities of ANSI with BSI among others. ISO is not an acronym, but derives from the Greek word, *isos*, meaning "equal." The ISO consists of over 2,800 technical committees, each of which is charged with some global standardization issue. Its interests range from the behavior of photographic film to the pitch of screw threads to the complex world of computer engineering. The proliferation of global trade has been facilitated by the ISO. Today, the ISO touches virtually every aspect of our lives.

Throughout this book, we mention official standards designations where appropriate. Definitive information concerning many of these standards can be

found in excruciating detail on the Web site of the organization responsible for establishing the standard cited. As an added bonus, many standards contain "normative" and informative references, which provide background information in areas related to the standard.

1.5 HISTORICAL DEVELOPMENT

During their 50-year life span, computers have become the perfect example of modern convenience. Living memory is strained to recall the days of steno pools, carbon paper, and mimeograph machines. It sometimes seems that these magical computing machines were developed instantaneously in the form that we now know them. But the developmental path of computers is paved with accidental discovery, commercial coercion, and whimsical fancy. And occasionally computers have even improved through the application of solid engineering practices! Despite all of the twists, turns, and technological dead ends, computers have evolved at a pace that defies comprehension. We can fully appreciate where we are today only when we have seen where we've come from.

In the sections that follow, we divide the evolution of computers into generations, each generation being defined by the technology used to build the machine. We have provided approximate dates for each generation for reference purposes only. You will find little agreement among experts as to the exact starting and ending times of each technological epoch.

Every invention reflects the time in which it was made, so one might wonder whether it would have been called a computer if it had been invented in the late 1990s. How much computation do we actually see pouring from the mysterious boxes perched on or beside our desks? Until recently, computers served us only by performing mind-bending mathematical manipulations. No longer limited to white-jacketed scientists, today's computers help us to write documents, keep in touch with loved ones across the globe, and do our shopping chores. Modern business computers spend only a minuscule part of their time performing accounting calculations. Their main purpose is to provide users with a bounty of strategic information for competitive advantage. Has the word *computer* now become a misnomer? An anachronism? What, then, should we call them, if not computers?

We cannot present the complete history of computing in a few pages. Entire books have been written on this subject and even they leave their readers wanting for more detail. If we have piqued your interest, we refer you to look at some of the books cited in the list of references at the end of this chapter.

1.5.1 Generation Zero: Mechanical Calculating Machines (1642–1945)

Prior to the 1500s, a typical European businessperson used an abacus for calculations and recorded the result of his ciphering in Roman numerals. After the decimal numbering system finally replaced Roman numerals, a number of people invented devices to make decimal calculations even faster and more accurate. Wilhelm Schickard (1592–1635) has been credited with the invention of the first mechanical calculator, the Calculating Clock (exact date unknown). This device was able to add and subtract numbers containing as many as six digits. In 1642, Blaise Pascal (1623–1662) developed a mechanical calculator called the Pascaline to help his father with his tax work. The Pascaline could do addition with carry and subtraction. It was probably the first mechanical adding device actually used for a practical purpose. In fact, the Pascaline was so well conceived that its basic design was still being used at the beginning of the twentieth century, as evidenced by the Lightning Portable Adder in 1908, and the Addometer in 1920. Gottfried Wilhelm von Leibniz (1646–1716), a noted mathematician, invented a calculator known as the Stepped Reckoner that could add, subtract, multiply, and divide. None of these devices could be programmed or had memory. They required manual intervention throughout each step of their calculations.

Although machines like the Pascaline were used into the twentieth century, new calculator designs began to emerge in the nineteenth century. One of the most ambitious of these new designs was the Difference Engine by Charles Babbage (1791–1871). Some people refer to Babbage as "the father of computing." By all accounts, he was an eccentric genius who brought us, among other things, the skeleton key and the "cow catcher," a device intended to push cows and other movable obstructions out of the way of locomotives.

Babbage built his Difference Engine in 1822. The Difference Engine got its name because it used a calculating technique called the *method of differences*. The machine was designed to mechanize the solution of polynomial functions and was actually a calculator, not a computer. Babbage also designed a general-purpose machine in 1833 called the Analytical Engine. Although Babbage died before he could build it, the Analytical Engine was designed to be more versatile than his earlier Difference Engine. The Analytical Engine would have been capable of performing any mathematical operation. The Analytical Engine included many of the components associated with modern computers: an arithmetic processing unit to perform calculations (Babbage referred to this as the *mill*), a memory (the *store*), and input and output devices. Babbage also included a conditional branching operation where the next instruction to be performed was determined by the result of the previous operation. Ada, Countess of Lovelace and daughter of poet Lord Byron, suggested that Babbage write a plan for how the machine would calculate numbers. This is regarded as the first computer program, and Ada is considered to be the first computer programmer. It is also rumored that she suggested the use of the binary number system rather than the decimal number system to store data.

A perennial problem facing machine designers has been how to get data into the machine. Babbage designed the Analytical Engine to use a type of punched card for input and programming. Using cards to control the behavior of a machine did not originate with Babbage, but with one of his friends, Joseph-Marie Jacquard (1752–1834). In 1801, Jacquard invented a programmable weaving loom that could produce intricate patterns in cloth. Jacquard gave Babbage a tapestry that had been woven on this loom using more than 10,000 punched cards. To Babbage, it seemed only natural that if a loom could be controlled by cards,

then his Analytical Engine could be as well. Ada expressed her delight with this idea, writing, "[T]he Analytical Engine weaves algebraical patterns just as the Jacquard loom weaves flowers and leaves."

The punched card proved to be the most enduring means of providing input to a computer system. Keyed data input had to wait until fundamental changes were made in how calculating machines were constructed. In the latter half of the nineteenth century, most machines used wheeled mechanisms, which were difficult to integrate with early keyboards because they were levered devices. But levered devices could easily punch cards and wheeled devices could easily read them. So a number of devices were invented to encode and then "tabulate" card-punched data. The most important of the late-nineteenth-century tabulating machines was the one invented by Herman Hollerith (1860–1929). Hollerith's machine was used for encoding and compiling 1890 census data. This census was completed in record time, thus boosting Hollerith's finances and the reputation of his invention. Hollerith later founded the company that would become IBM. His 80-column punched card, the *Hollerith card*, was a staple of automated data processing for over 50 years.

1.5.2 The First Generation: Vacuum Tube Computers (1945–1953)

Although Babbage is often called the "father of computing," his machines were mechanical, not electrical or electronic. In the 1930s, Konrad Zuse (1910–1995) picked up where Babbage left off, adding electrical technology and other improvements to Babbage's design. Zuse's computer, the Z1, used electromechanical relays instead of Babbage's hand-cranked gears. The Z1 was programmable and had a memory, an arithmetic unit, and a control unit. Because money and resources were scarce in wartime Germany, Zuse used discarded movie film instead of punched cards for input. Although his machine was designed to use vacuum tubes, Zuse, who was building his machine on his own, could not afford the tubes. Thus, the Z1 correctly belongs in the first generation, although it had no tubes.

Zuse built the Z1 in his parents' Berlin living room while Germany was at war with most of Europe. Fortunately, he couldn't convince the Nazis to buy his machine. They did not realize the tactical advantage such a device would give them. Allied bombs destroyed all three of Zuse's first systems, the Z1, Z2, and Z3. Zuse's impressive machines could not be refined until after the war and ended up being another "evolutionary dead end" in the history of computers.

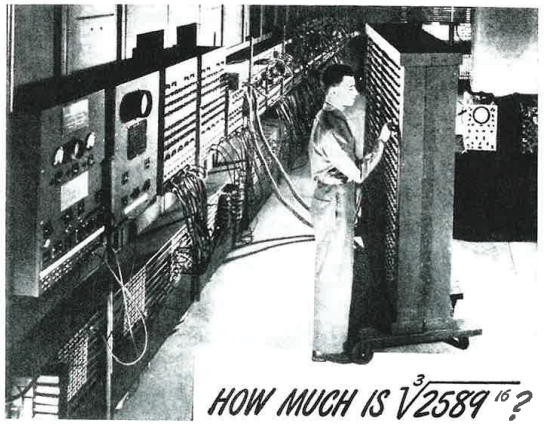
Digital computers, as we know them today, are the outcome of work done by a number of people in the 1930s and 1940s. Pascal's basic mechanical calculator was designed and modified simultaneously by many people; the same can be said of the modern electronic computer. Notwithstanding the continual arguments about who was first with what, three people clearly stand out as the inventors of modern computers: John Atanasoff, John Mauchly, and J. Presper Eckert.

John Atanasoff (1904–1995) has been credited with the construction of the first completely electronic computer. The Atanasoff Berry Computer (ABC) was a binary machine built from vacuum tubes. Because this system was built specifically to solve systems of linear equations, we cannot call it a general-purpose computer. There were, however, some features that the ABC had in common with the general-purpose ENIAC (Electronic Numerical Integrator and Computer), which was invented a few years later. These common features caused considerable controversy as to who should be given the credit (and patent rights) for the invention of the electronic digital computer. (The interested reader can find more details on a rather lengthy lawsuit involving Atanasoff and the ABC in Mollenhoff [1988].)

John Mauchly (1907–1980) and J. Presper Eckert (1929–1995) were the two principle inventors of the ENIAC, introduced to the public in 1946. The ENIAC is recognized as the first all-electronic, general-purpose digital computer. This machine used 17,468 vacuum tubes, occupied 1,800 square feet of floor space, weighed 30 tons, and consumed 174 kilowatts of power. The ENIAC had a memory capacity of about 1,000 information bits (about 20 10-digit decimal numbers) and used punched cards to store data.

John Mauchly's vision for an electronic calculating machine was born from his lifelong interest in predicting the weather mathematically. While a professor of physics at Ursinus College near Philadelphia, Mauchly engaged dozens of adding machines and student operators to crunch mounds of data that he believed would reveal mathematical relationships behind weather patterns. He felt that if he could have only a little more computational power, he could reach the goal that seemed just beyond his grasp. Pursuant to the Allied war effort, and with ulterior motives to learn about electronic computation, Mauchly volunteered for a crash course in electrical engineering at the University of Pennsylvania's Moore School of Engineering. Upon completion of this program, Mauchly accepted a teaching position at the Moore School, where he taught a brilliant young student, J. Presper Eckert. Mauchly and Eckert found a mutual interest in building an electronic calculating device. In order to secure the funding they needed to build their machine, they wrote a formal proposal for review by the school. They portrayed their machine as conservatively as they could, billing it as an "automatic calculator." Although they probably knew that computers would be able to function most efficiently using the binary numbering system, Mauchly and Eckert designed their system to use base 10 numbers, in keeping with the appearance of building a huge electronic adding machine. The university rejected Mauchly and Eckert's proposal. Fortunately, the United States Army was more interested.

During World War II, the army had an insatiable need for calculating the trajectories of its new ballistic armaments. Thousands of human "computers" were engaged around the clock cranking through the arithmetic required for these firing tables. Realizing that an electronic device could shorten ballistic table calculation from days to minutes, the army funded the ENIAC. And the ENIAC did indeed shorten the time to calculate a table from 20 hours to 30 seconds. Unfortunately, the machine wasn't ready before the end of the war. But the ENIAC had shown that vacuum tube computers were fast and feasible. During the next decade, vacuum tube systems continued to improve and were commercially successful.



HOW MUCH IS $\sqrt[3]{2589^{16}}$?

The Army's ENIAC can give you the answer in a fraction of a second!

Think that's a stumper? You should see *some* of the ENIAC's problems! Brain twisters that if put to paper would run off this page and feet beyond . . . addition, subtraction, multiplication, division—square root, cube root, any root. Solved by an incredibly complex system of circuits operating 18,000 electronic tubes and tipping the scales at 30 tons!

The ENIAC is symbolic of many amazing Army devices with a brilliant future for you! The new Regular Army needs men with aptitude for scientific work, and as one of the first trained in the post-war era, you stand to get in on the ground floor of important jobs which have never before existed. You'll find that an Army career pays off.

The most attractive fields are filling quickly. Get into the swim while the getting's good! 1½, 2 and 3 year enlistments are open in the Regular Army to ambitious young men 18 to 34 (17 with parents' consent) who are otherwise qualified. If you enlist for 3 years, you may choose your own branch of the service, of those still open. Get full details at your nearest Army Recruiting Station.

YOUR REGULAR ARMY SERVES THE NATION AND MANKIND IN WAR AND PEACE

A GOOD JOB FOR YOU

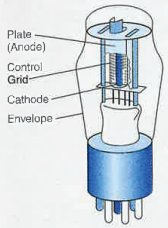
U. S. Army

CHOOSE THIS FINE PROFESSION NOW!

U.S. Army. 1946

What Is a Vacuum Tube?

The wired world that we know today was born from the invention of a single electronic device called a *vacuum tube* by Americans and—more accurately—a *valve* by the British. Vacuum tubes should be called valves because they control the flow of electrons in electrical systems in much the same way as valves control the flow of water in a plumbing system. In fact, some mid-twentieth-century breeds of these electron tubes contain no vacuum at all, but are filled with conductive gasses, such as mercury vapor, which can provide desirable electrical behavior.



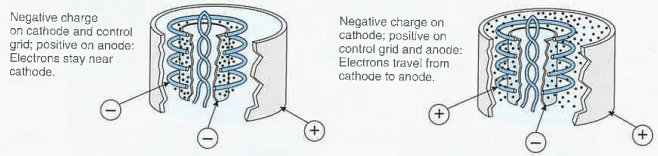
The electrical phenomenon that makes tubes work was discovered by Thomas A. Edison in 1883 while he was trying to find ways to keep the filaments of his light bulbs from burning away (or oxidizing) a few minutes after electrical current was applied. Edison reasoned correctly that one way to prevent filament oxidation would be to place the filament in a vacuum. Edison didn't immediately understand that air not only supports combustion, but also is a good insulator. When he energized the electrodes holding a new tungsten filament, the filament soon became hot and burned out as the others had before it. This time, however, Edison noticed that electricity continued to flow from the warmed negative terminal to the cool positive terminal within the light bulb. In 1911, Owen Willans Richardson analyzed this behavior. He concluded that when a negatively charged filament was heated, electrons "boiled off" as water molecules can be boiled to create steam. He aptly named this phenomenon *thermionic emission*.

Thermionic emission, as Edison had documented it, was thought by many to be only an electrical curiosity. But in 1905 a British former assistant to Edison, John A. Fleming, saw Edison's discovery as much more than a novelty. He knew that thermionic emission supported the flow of electrons in only one direction: from the negatively charged *cathode* to the positively charged *anode*, also called a *plate*. He realized that this behavior could *rectify* alternating current. That is, it could change alternating current into the direct current that was essential for the proper operation of telegraph equipment. Fleming used his ideas to invent an electronic valve later called a *diode tube* or *rectifier*.

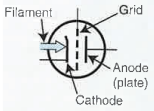


The diode was well suited for changing alternating current into direct current, but the greatest power of the electron tube was yet to be discovered. In

1907, an American named Lee DeForest added a third element, called a *control grid*. The control grid, when carrying a negative charge, can reduce or prevent electron flow from the cathode to the anode of a diode.

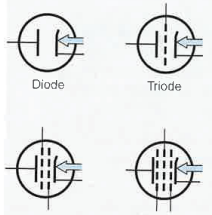


When DeForest patented his device, he called it an *audion tube*. It was later known as a *triode*. The schematic symbol for the triode is shown at the left.



A triode can act either as a switch or as an amplifier. Small changes in the charge of the control grid can cause much larger changes in the flow of electrons between the cathode and the anode. Therefore, a weak signal applied to the grid results in a much stronger signal at the plate output. A sufficiently large negative charge applied to the grid stops all electrons from leaving the cathode.

Additional control grids were eventually added to the triode to allow more exact control of the electron flow. Tubes with two grids (four elements) are called *tetrodes*; tubes with three grids are called *pentodes*. Triodes and pentodes were the tubes most commonly used in communications and computer applications. Often, two or three triodes or pentodes would be combined within one envelope so that they could share a single heater, thereby reducing the power consumption of a particular device. These latter-day devices were called "miniature" tubes because many were about 2 inches (5cm) high and one-half inch (1.5cm) in diameter. Equivalent full-sized diodes, triodes, and pentodes were just a little smaller than a household light bulb.



Vacuum tubes were not well suited for building computers. Even the simplest vacuum tube computer system required thousands of tubes. Enormous amounts of electrical power were required to heat the cathodes of these devices. To prevent a meltdown, this heat had to be removed from the system as quickly as possible. Power consumption and heat dissipation could be reduced by running the cathode heaters at lower voltages, but this reduced the already slow switching speed of the tube. Despite their limitations and power consumption, vacuum tube computer

systems, both analog and digital, served their purpose for many years and are the architectural foundation for all modern computer systems.

Although decades have passed since the last vacuum tube computer was manufactured, vacuum tubes are still used in audio amplifiers. These "high-end" amplifiers are favored by musicians who believe that tubes provide a resonant and pleasing sound unattainable by solid-state devices.

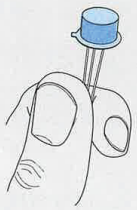
1.5.3 The Second Generation: Transistorized Computers (1954–1965)

The vacuum tube technology of the first generation was not very dependable. In fact, some ENIAC detractors believed that the system would never run because the tubes would burn out faster than they could be replaced. Although system reliability wasn't as bad as the doomsayers predicted, vacuum tube systems often experienced more downtime than uptime.

In 1948, three researchers with Bell Laboratories—John Bardeen, Walter Brattain, and William Shockley—invented the transistor. This new technology not only revolutionized devices such as televisions and radios, but also pushed the computer industry into a new generation. Because transistors consume less power than vacuum tubes, are smaller, and work more reliably, the circuitry in computers consequently became smaller and more reliable. Despite using transistors, computers of this generation were still bulky and quite costly. Typically only universities, governments, and large businesses could justify the expense. Nevertheless, a plethora of computer makers emerged in this generation; IBM, Digital Equipment Corporation (DEC), and Univac (now Unisys) dominated the industry. IBM marketed the 7094 for scientific applications and the 1401 for business applications. DEC was busy manufacturing the PDP-1. A company founded (but soon sold) by Mauchly and Eckert built the Univac systems. The most successful Unisys systems of this generation belonged to its 1100 series. Another company, Control Data Corporation (CDC), under the supervision of Seymour Cray, built the CDC 6600, the world's first supercomputer. The $10 million CDC 6600 could perform 10 million instructions per second, used 60-bit words, and had an astounding 128 kilowords of main memory.

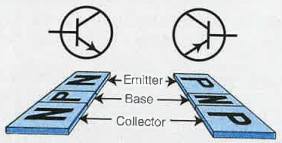
What Is a Transistor?

The transistor, short for *transfer resistor*, is the solid-state version of the triode. There is no such thing as a solid-state version of the tetrode or pentode. Because electrons are better behaved in a solid medium than in the open void of a vacuum tube, they need no extra controlling grids. Either germanium or silicon can be the basic "solid" used in these solid state devices. In their pure form, neither of these elements is a good conductor of electricity. But when they are combined with



trace amounts of elements that are their neighbors in the Periodic Chart of the Elements, they conduct electricity in an effective and easily controlled manner.

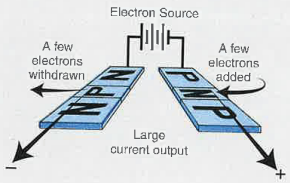
Boron, aluminum, and gallium can be found to the left of silicon and germanium on the Periodic Chart. Because they lie to the left of silicon and germanium, they have one less electron in their outer electron shell, or *valence*. So if you add a small amount of aluminum to silicon, the silicon ends up with a slight imbalance in its outer electron shell, and therefore attracts electrons from any pole that has a negative potential (an excess of electrons). When modified (or *doped*) in this way, silicon or germanium becomes a *P-type* material.



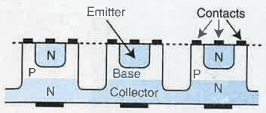
Similarly, if we add a little boron, arsenic, or gallium to silicon, we'll have extra electrons in valences of the silicon crystals. This gives us an *N-type* material. A small amount of current will flow through the N-type material if we provide the loosely bound electrons in the N-type material with a place to go. In other words, if we apply a positive potential to N-type material, electrons will flow from the negative pole to the positive pole. If the poles are reversed, that is, if we apply a negative potential to the N-type material and a positive potential to the P-type material, no current will flow. This means that we can make a solid-state diode from a simple junction of N- and P-type materials.

The solid-state triode, the transistor, consists of three layers of semiconductor material. Either a slice of P-type material is sandwiched between two N-type materials, or a slice of N-type material is sandwiched between two P-type materials. The former is called an NPN transistor, the latter a PNP transistor. The inner layer of the transistor is called the base; the other two layers are called the collector and emitter.

The figure to the left shows how current flows through NPN and PNP transistors. The base in a transistor works just like the control grid in a triode tube: Small changes in the current at the base of a transistor result in a large electron flow from the emitter to the collector.



A *discrete-component* transistor is shown in "TO-50" packaging in the figure at the beginning of this sidebar. There are only three wires (leads) that connect the base, emitter, and collector of the transistor to the rest of the circuit. Transistors are not only smaller than vacuum tubes, but they also run cooler and are much more reliable. Vacuum tube filaments, like light bulb filaments, run hot and eventually burn out. Computers using transistorized components will naturally be smaller and run cooler than their vacuum tube predecessors. The ultimate miniaturization, however, is not realized by replacing individual triodes with discrete transistors, but in shrinking entire circuits onto one piece of silicon.

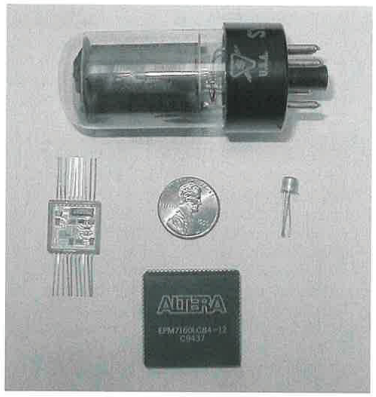


Integrated circuits, or *chips*, contain hundreds to millions of microscopic transistors. Several different techniques are used to manufacture integrated circuits. One of the simplest methods involves creating a circuit using computer-aided design software that can print large maps of each of the several silicon layers forming the chip. Each map is used like a photographic negative where light-induced changes in a photoresistive substance on the chip's surface produce the delicate patterns of the circuit when the silicon chip is immersed in a chemical that washes away the exposed areas of the silicon. This technique is called *photomicrolithography*. After the etching is completed, a layer of N-type or P-type material is deposited on the bumpy surface of the chip. This layer is then treated with a photoresistive substance, exposed to light, and etched as was the layer before it. This process continues until all of the layers have been etched. The resulting peaks and valleys of P- and N-material form microscopic electronic components, including transistors, that behave just like larger versions fashioned from discrete components, except that they run a lot faster and consume a small fraction of the power.

1.5.4 The Third Generation: Integrated Circuit Computers (1965–1980)

The real explosion in computer use came with the integrated circuit generation. Jack Kilby invented the integrated circuit (IC) or *microchip*, made of germanium. Six months later, Robert Noyce (who had also been working on integrated circuit design) created a similar device using silicon instead of germanium. This is the silicon chip upon which the computer industry was built. Early ICs allowed dozens of transistors to exist on a single silicon chip that was smaller than a single "discrete component" transistor. Computers became faster, smaller, and cheaper, bringing huge gains in processing power. The IBM System/360 family of computers was among the first commercially available systems to be built entirely of solid-state components. The 360 product line was also IBM's first offering where all of the machines in the family were compatible, meaning they all used the same assembly language. Users of smaller machines could upgrade to larger systems without rewriting all of their software. This was a revolutionary new concept at the time.

The IC generation also saw the introduction of time-sharing and multiprogramming (the ability for more than one person to use the computer at a time). Multiprogramming, in turn, necessitated the introduction of new operating systems for these computers. Time-sharing minicomputers such as DEC's PDP-8 and PDP-11 made computing affordable to smaller businesses and more universities.



Comparison of Computer Components
Clockwise, starting from the top:
1) Vacuum Tube
2) Transistor
3) Chip containing 3200 2-input NAND gates
4) Integrated circuit package (the small silver square in the lower left-hand corner is an integrated circuit)

Courtesy of Linda Null

IC technology also allowed for the development of more powerful supercomputers. Seymour Cray took what he had learned while building the CDC 6600 and started his own company, the Cray Research Corporation. This company produced a number of supercomputers, starting with the $8.8 million Cray-1, in 1976. The Cray-1, in stark contrast to the CDC 6600, could execute over 160 million instructions per second and could support 8 megabytes of memory.

1.5.5 The Fourth Generation: VLSI Computers (1980–????)

In the third generation of electronic evolution, multiple transistors were integrated onto one chip. As manufacturing techniques and chip technologies advanced, increasing numbers of transistors were packed onto one chip. There are now various levels of integration: SSI (small scale integration), in which there are

10 to 100 components per chip; MSI (medium scale integration), in which there are 100 to 1,000 components per chip; LSI (large scale integration), in which there are 1,000 to 10,000 components per chip; and finally, VLSI (very large scale integration), in which there are more than 10,000 components per chip. This last level, VLSI, marks the beginning of the fourth generation of computers.

To give some perspective to these numbers, consider the ENIAC-on-a-chip project. In 1997, to commemorate the fiftieth anniversary of its first public demonstration, a group of students at the University of Pennsylvania constructed a single-chip equivalent of the ENIAC. The 1,800 square-foot, 30-ton beast that devoured 174 kilowatts of power the minute it was turned on had been reproduced on a chip the size of a thumbnail. This chip contained approximately 174,569 transistors—an order of magnitude fewer than the number of components typically placed on the same amount of silicon in the late 1990s.

VLSI allowed Intel, in 1971, to create the world's first microprocessor, the 4004, which was a fully functional, 4-bit system that ran at 108KHz. Intel also introduced the random access memory (RAM) chip, accommodating four kilobits of memory on a single chip. This allowed computers of the fourth generation to become smaller and faster than their solid-state predecessors.

VLSI technology, and its incredible shrinking circuits, spawned the development of microcomputers. These systems were small enough and inexpensive enough to make computers available and affordable to the general public. The premiere microcomputer was the Altair 8800, released in 1975 by the Micro Instrumentation and Telemetry (MITS) corporation. The Altair 8800 was soon followed by the Apple I and Apple II, and Commodore's PET and Vic 20. Finally, in 1981, IBM introduced its PC (Personal Computer).

The Personal Computer was IBM's third attempt at producing an "entry-level" computer system. Its Datamaster as well as its 5100 Series desktop computers flopped miserably in the marketplace. Despite these early failures, IBM's John Opel convinced his management to try again. He suggested forming a fairly autonomous "independent business unit" in Boca Raton, Florida, far from IBM's headquarters in Armonk, New York. Opel picked Don Estridge, an energetic and capable engineer, to champion the development of the new system, code-named the Acorn. In light of IBM's past failures in the small-systems area, corporate management held tight rein on the Acorn's timeline and finances. Opel could get his project off of the ground only after promising to deliver it within a year, a seemingly impossible feat.

Estridge knew that the only way that he could deliver the PC within the wildly optimistic 12-month schedule would be to break with IBM convention and use as many "off-the-shelf" parts as possible. Thus, from the outset, the IBM PC was conceived with an "open" architecture. Although some people at IBM may have later regretted the decision to keep the architecture of the PC as nonproprietary as possible, it was this very openness that allowed IBM to set the standard for the industry. While IBM's competitors were busy suing companies for copying their system designs, PC clones proliferated. Before long, the price of "IBM-compatible" microcomputers came within reach for just about every small

business. Also, thanks to the clone makers, large numbers of these systems soon began finding true "personal use" in people's homes.

IBM eventually lost its microcomputer market dominance, but the genie was out of the bottle. For better or worse, the IBM architecture continues to be the de facto standard for microcomputing, with each year heralding bigger and faster systems. Today, the average desktop computer has many times the computational power of the mainframes of the 1960s.

Since the 1960s, mainframe computers have seen stunning improvements in price-performance ratios owing to VLSI technology. Although the IBM System/360 was an entirely solid-state system, it was still a water-cooled, power-gobbling behemoth. It could perform only about 50,000 instructions per second and supported only 16 megabytes of memory (while usually having *kilobytes* of physical memory installed). These systems were so costly that only the largest businesses and universities could afford to own or lease one. Today's mainframes—now called "enterprise servers"—are still priced in the millions of dollars, but their processing capabilities have grown several thousand times over, passing the billion-instructions-per-second mark in the late 1990s. These systems, often used as Web servers, routinely support hundreds of thousands of transactions *per minute!*

The processing power brought by VLSI to supercomputers defies comprehension. The first supercomputer, the CDC 6600, could perform 10 million instructions per second, and had 128 kilobytes of main memory. By contrast, supercomputers of today contain thousands of processors, can address terabytes of memory, and will soon be able to perform a *quadrillion* instructions per second.

What technology will mark the beginning of the fifth generation? Some say that the fifth generation will mark the acceptance of parallel processing and the use of networks and single-user workstations. Many people believe we have already crossed into this generation. Some people characterize the fifth generation as being the generation of neural network, DNA, or optical computing systems. It's possible that we won't be able to define the fifth generation until we have advanced into the sixth or seventh generation, and whatever those eras will bring.

1.5.6 Moore's Law

So where does it end? How small can we make transistors? How densely can we pack chips? No one can say for sure. Every year, scientists continue to thwart prognosticators' attempts to define the limits of integration. In fact, more than one skeptic raised an eyebrow when, in 1965, Intel founder Gordon Moore stated, "The density of transistors in an integrated circuit will double every year." The current version of this prediction is usually conveyed as "the density of silicon chips doubles every 18 months." This assertion has become known as *Moore's Law*. Moore intended this postulate to hold for only 10 years. However, advances in chip manufacturing processes have allowed this law to hold for almost 40 years (and many believe it will continue to hold well into the 2010s).

Yet, using current technology, Moore's Law cannot hold forever. There are physical and financial limitations that must ultimately come into play. At the current rate of miniaturization, it would take about 500 years to put the entire solar system on a chip! Clearly, the limit lies somewhere between here and there. Cost may be the ultimate constraint. Rock's Law, proposed by early Intel capitalist Arthur Rock, is a corollary to Moore's law: "The cost of capital equipment to build semiconductors will double every four years." Rock's Law arises from the observations of a financier who has seen the price tag of new chip facilities escalate from about $12,000 in 1968 to $12 million in the late 1990s. At this rate, by the year 2035, not only will the size of a memory element be smaller than an atom, but it would also require the entire wealth of the world to build a single chip! So even if we continue to make chips smaller and faster, the ultimate question may be whether we can afford to build them.

Certainly, if Moore's Law is to hold, Rock's Law must fall. It is evident that for these two things to happen, computers must shift to a radically different technology. Research into new computing paradigms has been proceeding in earnest during the last half decade. Laboratory prototypes fashioned around organic computing, superconducting, molecular physics, and quantum computing have been demonstrated. Quantum computers, which leverage the vagaries of quantum mechanics to solve computational problems, are particularly exciting. Not only would quantum systems compute exponentially faster than any previously used method, they would also revolutionize the way in which we define computational problems. Problems that today are considered ludicrously infeasible could be well within the grasp of the next generation's schoolchildren. These schoolchildren may, in fact, chuckle at our "primitive" systems in the same way that we are tempted to chuckle at the ENIAC.

1.6 THE COMPUTER LEVEL HIERARCHY

If a machine is to be capable of solving a wide range of problems, it must be able to execute programs written in different languages, from FORTRAN and C to Lisp and Prolog. As we shall see in Chapter 3, the only physical components we have to work with are wires and gates. A formidable open space—*a semantic gap*—exists between these physical components and a high-level language such as C++. For a system to be practical, the semantic gap must be invisible to most of the users of the system.

Programming experience teaches us that when a problem is large, we should break it down and use a "divide and conquer" approach. In programming, we divide a problem into modules and then design each module separately. Each module performs a specific task and modules need only know how to interface with other modules to make use of them.

Computer system organization can be approached in a similar manner. Through the principle of abstraction, we can imagine the machine to be built from a hierarchy of levels, in which each level has a specific function and exists as a distinct hypothetical machine. We call the hypothetical computer at each level a *virtual machine*. Each level's virtual machine executes its own particular set of instructions, calling upon machines at lower levels to carry out the tasks when necessary. By studying computer organization, you will see the rationale behind the hierarchy's partitioning, as

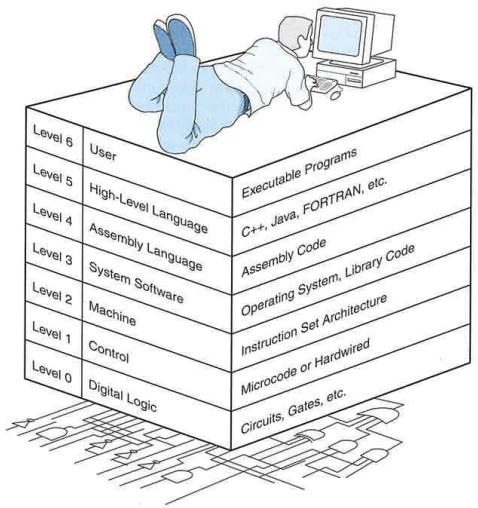


FIGURE 1.3 The Abstract Levels of Modern Computing Systems

well as how these layers are implemented and interface with each other. Figure 1.3 shows the commonly accepted layers representing the abstract virtual machines.

Level 6, the User Level, is composed of applications and is the level with which everyone is most familiar. At this level, we run programs such as word processors, graphics packages, or games. The lower levels are nearly invisible from the User Level.

Level 5, the High-Level Language Level, consists of languages such as C, C++, FORTRAN, Lisp, Pascal, and Prolog. These languages must be translated (using either a compiler or an interpreter) to a language the machine can understand. Compiled languages are translated into assembly language and then assembled into machine code. (They are translated to the next lower level.) The user at this level sees very little of the lower levels. Even though a programmer must know about data types and the instructions available for those types, she need not know about how those types are actually implemented.

Level 4, the Assembly Language Level, encompasses some type of assembly language. As previously mentioned, compiled higher-level languages are first translated to assembly, which is then directly translated to machine language. This is a one-to-one translation, meaning that one assembly language instruction is translated to exactly one machine language instruction. By having separate levels, we reduce the semantic gap between a high-level language, such as C++, and the actual machine language (which consists of 0s and 1s).

Level 3, the System Software Level, deals with operating system instructions. This level is responsible for multiprogramming, protecting memory, synchronizing processes, and various other important functions. Often, instructions translated from assembly language to machine language are passed through this level unmodified.

Level 2, the Instruction Set Architecture (ISA), or Machine Level, consists of the machine language recognized by the particular architecture of the computer system. Programs written in a computer's true machine language on a hardwired computer (see below) can be executed directly by the electronic circuits without any interpreters, translators, or compilers. We will study instruction set architectures in depth in Chapters 4 and 5.

Level 1, the Control Level, is where a *control unit* makes sure that instructions are decoded and executed properly and that data is moved where and when it should be. The control unit interprets the machine instructions passed to it, one at a time, from the level above, causing the required actions to take place.

Control units can be designed in one of two ways: They can be *hardwired* or they can be *microprogrammed*. In hardwired control units, control signals emanate from blocks of digital logic components. These signals direct all of the data and instruction traffic to appropriate parts of the system. Hardwired control units are typically very fast because they are actually physical components. However, once implemented, they are very difficult to modify for the same reason.

The other option for control is to implement instructions using a microprogram. A microprogram is a program written in a low-level language that is implemented directly by the hardware. Machine instructions produced in Level 2 are fed into this microprogram, which then interprets the instructions by activating hardware suited to execute the original instruction. One machine-level instruction is often translated into several microcode instructions. This is not the one-to-one correlation that exists between assembly language and machine language. Microprograms are popular because they can be modified relatively easily. The disadvantage of microprogramming is, of course, that the additional layer of translation typically results in slower instruction execution.

Level 0, the Digital Logic Level, is where we find the physical components of the computer system: the gates and wires. These are the fundamental building blocks, the implementations of the mathematical logic, that are common to all computer systems. Chapter 3 presents the Digital Logic Level in detail.

1.7 THE VON NEUMANN MODEL

In the earliest electronic computing machines, programming was synonymous with connecting wires to plugs. No layered architecture existed, so programming

a computer was as much of a feat of electrical engineering as it was an exercise in algorithm design. Before their work on the ENIAC was complete, John W. Mauchly and J. Presper Eckert conceived of an easier way to change the behavior of their calculating machine. They reckoned that memory devices, in the form of mercury delay lines, could provide a way to store program instructions. This would forever end the tedium of rewiring the system each time it had a new problem to solve, or an old one to debug. Mauchly and Eckert documented their idea, proposing it as the foundation for their next computer, the EDVAC. Unfortunately, while they were involved in the top secret ENIAC project during World War II, Mauchly and Eckert could not immediately publish their insight.

No such proscriptions, however, applied to a number of people working at the periphery of the ENIAC project. One of these people was a famous Hungarian mathematician named John von Neumann (pronounced *von noy-man*). After reading Mauchly and Eckert's proposal for the EDVAC, von Neumann published and publicized the idea. So effective was he in the delivery of this concept that history has credited him with its invention. All stored-program computers have come to be known as *von Neumann systems* using the *von Neumann architecture*. Although we are compelled by tradition to say that stored-program computers use the von Neumann architecture, we shall not do so without paying proper tribute to its true inventors: John W. Mauchly and J. Presper Eckert.

Today's version of the stored-program machine architecture satisfies at least the following characteristics:

- Consists of three hardware systems: A *central processing unit* (*CPU*) with a control unit, an *arithmetic logic unit* (*ALU*), *registers* (small storage areas), and a program counter; a *main-memory system*, which holds programs that control the computer's operation; and an *I/O system*.
- Capacity to carry out sequential instruction processing
- Contains a single path, either physically or logically, between the main memory system and the control unit of the CPU, forcing alternation of instruction and execution cycles. This single path is often referred to as the *von Neumann bottleneck*.

Figure 1.4 shows how these features work together in modern computer systems. Notice that the system shown in the figure passes all of its I/O through the arithmetic logic unit (actually, it passes through the accumulator, which is part of the ALU). This architecture runs programs in what is known as the *von Neumann execution cycle* (also called the *fetch-decode-execute cycle*), which describes how the machine works. One iteration of the cycle is as follows:

1. The control unit fetches the next program instruction from the memory, using the program counter to determine where the instruction is located.
2. The instruction is decoded into a language the ALU can understand.
3. Any data operands required to execute the instruction are fetched from memory and placed into registers within the CPU.
4. The ALU executes the instruction and places the results in registers or memory.

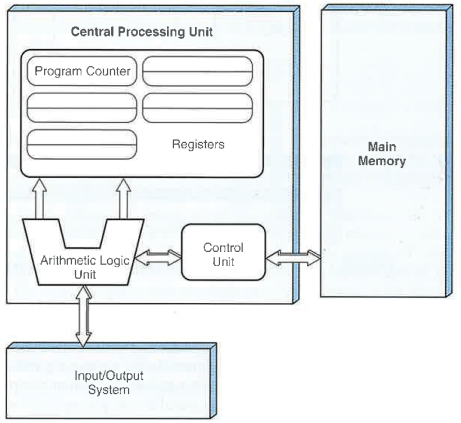


FIGURE 1.4 The von Neumann Architecture

The ideas present in the von Neumann architecture have been extended so that programs and data stored in a slow-to-access storage medium, such as a hard disk, can be copied to a fast-access, volatile storage medium such as RAM prior to execution. This architecture has also been streamlined into what is currently called the *system bus model*, which is shown in Figure 1.5. The data bus moves data from main memory to the CPU registers (and vice versa). The address bus holds the address of the data that the data bus is currently accessing. The control bus carries the necessary control signals that specify how the information transfer is to take place.

Other enhancements to the von Neumann architecture include using index registers for addressing, adding floating point data, using interrupts and asynchronous I/O, adding virtual memory, and adding general registers. You will learn a great deal about these enhancements in the chapters that follow.

1.8 NON–VON NEUMANN MODELS

Until recently, almost all general-purpose computers followed the von Neumann design. However, the von Neumann bottleneck continues to baffle engineers looking for ways to build fast systems that are inexpensive and compatible with the vast body of commercially available software. Engineers who are not constrained

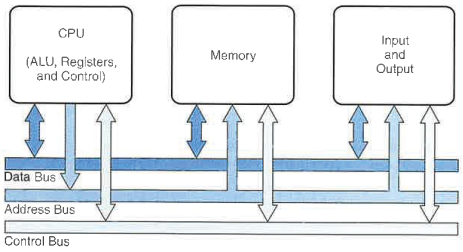


FIGURE 1.5 The Modified von Neumann Architecture, Adding a System Bus

by the need to maintain compatibility with von Neumann systems are free to use many different models of computing. A number of different subfields fall into the non-von Neumann category, including neural networks (using ideas from models of the brain as a computing paradigm), genetic algorithms (exploiting ideas from biology and DNA evolution), quantum computation (previously discussed), and parallel computers. Of these, parallel computing is currently the most popular.

Today, parallel processing solves some of our biggest problems in much the same way as settlers of the Old West solved their biggest problems using parallel oxen. If they were using an ox to move a tree and the ox was not big enough or strong enough, they certainly didn't try to grow a bigger ox—they used two oxen. If a computer isn't fast enough or powerful enough, instead of trying to develop a faster, more powerful computer, why not simply use multiple computers? This is precisely what parallel computing does. The first parallel-processing systems were built in the late 1960s and had only two processors. The 1970s saw the introduction of supercomputers with as many as 32 processors, and the 1980s brought the first systems with over 1,000 processors. Finally, in 1999, IBM announced the construction of a supercomputer called the Blue Gene. This massively parallel computer contains over 1 million processors, each with its own dedicated memory. Its first task is to analyze the behavior of protein molecules.

Even parallel computing has its limits, however. As the number of processors increases, so does the overhead of managing how tasks are distributed to those processors. Some parallel-processing systems require extra processors just to manage the rest of the processors and the resources assigned to them. No matter how many processors are placed in a system, or how many resources are assigned to them, somehow, somewhere, a bottleneck is bound to develop. The best that we can do to remedy this is to make sure that the slowest parts of the system are the ones that are used the least. This is the idea behind *Amdahl's Law*. This law states that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. The underlying premise is that every algorithm has a sequential part that ultimately limits the speedup that can be achieved by multiprocessor implementation.

CHAPTER SUMMARY

In this chapter we have presented a brief overview of computer organization and computer architecture and shown how they differ. We also have introduced some terminology in the context of a fictitious computer advertisement. Much of this terminology will be expanded on in later chapters.

Historically, **computers** were simply calculating machines. As computers became more sophisticated, they became general-purpose machines, which necessitated viewing each system as a hierarchy of levels instead of one gigantic machine. Each layer in this hierarchy serves a specific purpose, and all levels help minimize the semantic gap between a high-level programming language or application and the gates and wires that make up the physical hardware. Perhaps the single most important development in computing that affects us as programmers is the introduction of the stored-program concept of the von Neumann machine. Although there are other architectural models, the von Neumann architecture is predominant in today's general-purpose computers.

FURTHER READING

We encourage you to build on our brief presentation of the history of computers. We think that you will find this subject intriguing because it is as much about people as it is about machines. You can read about the "forgotten father of the computer," John Atanasoff, in Mollenhoff (1988). This book documents the odd relationship between Atanasoff and John Mauchly, and recounts the open court battle of two computer giants, Honeywell and Sperry Rand. This trial ultimately gave Atanasoff his proper recognition.

For a lighter look at computer history, try the book by Rochester and Gantz (1983). Augarten's (1985) illustrated history of computers is a delight to read and contains hundreds of hard-to-find pictures of early computers and computing devices. For a complete discussion of the historical development of computers, you can check out the three-volume dictionary by Cortada (1987). A particularly thoughtful account of the history of computing is presented in Ceruzzi (1998). If you are interested in an excellent set of case studies about historical computers, see Blaauw & Brooks (1997).

You will also be richly rewarded by reading McCartney's (1999) book about the ENIAC, Chopsky and Leonsis' (1988) chronicle of the development of the IBM PC, and Toole's (1998) biography of Ada, Countess of Lovelace. Polachek's (1997) article conveys a vivid picture of the complexity of calculating ballistic firing tables. After reading this article, you will understand why the army would gladly pay for anything that promised to make the process faster or more accurate. The Maxfield and Brown book (1997) contains a fascinating look at the origins and history of computing as well as in-depth explanations of how a computer works.

For more information on Moore's Law, we refer the reader to Schaller (1997). For detailed descriptions of early computers as well as profiles and reminiscences of industry pioneers, you may wish to consult the *IEEE Annals of the History of Computing*, which is published quarterly. The Computer Museum History Center can be found online at www.computerhistory.org. It contains various exhibits, research, timelines, and collections. Many cities now have computer museums and allow visitors to use some of the older computers.

A wealth of information can be found at the Web sites of the standards-making bodies discussed in this chapter (as well as the sites not discussed in this chapter). The IEEE can be found at: www.ieee.org; ANSI at www.ansi.org; the ISO at www.iso.ch; the BSI at www.bsi-global.com; and the ITU-T at www.itu.int. The ISO site offers a vast amount of information and standards reference materials.

The WWW Computer Architecture Home Page at www.cs.wisc.edu/~arch/www/ contains a comprehensive index to computer architecture–related information. Many USENET newsgroups are devoted to these topics as well, including comp.arch and comp.arch.storage.

The entire May/June 2000 issue of MIT's *Technology Review* magazine is devoted to architectures that may be the basis of tomorrow's computers. Reading this issue will be time well spent. In fact, we could say the same of every issue.

REFERENCES

Augarten, Stan. *Bit by Bit: An Illustrated History of Computers*. London: Unwin Paperbacks, 1985.

Blaauw, G., & Brooks, F. *Computer Architecture: Concepts and Evolution*. Reading, MA: Addison-Wesley, 1997.

Ceruzzi, Paul E. *A History of Modern Computing*. Cambridge, MA: MIT Press, 1998.

Chopsky, James, & Leonsis, Ted. *Blue Magic: The People, Power and Politics Behind the IBM Personal Computer*. New York: Facts on File Publications, 1988.

Cortada, J. W. *Historical Dictionary of Data Processing*, Volume 1: *Biographies*; Volume 2: *Organization*, Volume 3: *Technology*. Westport, CT: Greenwood Press, 1987.

Maguire, Yael, Boyden III, Edward S., and Gershenfeld, Neil. "Toward a Table-Top Quantum Computer." *IBM Systems Journal 39*: 3/4 (June 2000), pp. 823–839.

Maxfield, Clive, & Brown, A. *Bebop BYTES Back (An Unconventional Guide to Computers)*. Madison, AL: Doone Publications, 1997.

McCartney, Scott. *ENIAC: The Triumphs and Tragedies of the World's First Computer*. New York: Walker and Company, 1999.

Mollenhoff, Clark R. *Atanasoff: The Forgotten Father of the Computer*. Ames, IA: Iowa State University Press, 1988.

Polachek, Harry. "Before the ENIAC." *IEEE Annals of the History of Computing 19*: 2 (June 1997), pp. 25–30.

Rochester, J. B., & Gantz, J. *The Naked Computer: A Layperson's Almanac of Computer Lore, Wizardry, Personalities, Memorabilia, World Records, Mindblowers, and Tomfoolery*. New York: William A. Morrow, 1983.

Schaller, R. "Moore's Law: Past, Present, and Future." *IEEE Spectrum*, June 1997, pp. 52–59.

Tanenbaum, A. *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.

Toole, Betty A. *Ada, the Enchantress of Numbers: Prophet of the Computer Age*. Mill Valley, CA: Strawberry Press, 1998.

Waldrop, M. Mitchell. "Quantum Computing." *MIT Technology Review 103*: 3 (May/June 2000), pp. 60–66.

REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. What is the difference between computer organization and computer architecture?
2. What is an ISA?
3. What is the importance of the Principle of Equivalence of Hardware and Software?
4. Name the three basic components of every computer.
5. To what power of 10 does the prefix giga- refer? What is the (approximate) equivalent power of 2?
6. To what power of 10 does the prefix micro- refer? What is the (approximate) equivalent power of 2?
7. What unit is typically used to measure the speed of a computer clock?
8. Name two types of computer memory.
9. What is the mission of the IEEE?
10. What is the full name of the organization that uses the initials ISO? Is ISO an acronym?
11. ANSI is the acronym used by which organization?
12. What is the name of the Swiss organization that devotes itself to matters concerning telephony, telecommunications, and data communications?
13. Who is known as the father of computing and why?
14. What was the significance of the punch card?
15. Name two driving factors in the development of computers.
16. What is it about the transistor that made it such a great improvement over the vacuum tube?
17. How does an integrated circuit differ from a transistor?
18. Explain the differences between SSI, MSI, LSI, and VLSI.
19. What technology spawned the development of microcomputers? Why?
20. What is meant by an "open architecture"?
21. State Moore's Law. Can it hold indefinitely?
22. How is Rock's Law related to Moore's Law?
23. Name and explain the seven commonly accepted layers of the Computer Level Hierarchy. How does this arrangement help us to understand computer systems?
24. What was it about the von Neumann architecture that distinguished it from its predecessors?
25. Name the characteristics present in a von Neumann architecture.

26. How does the fetch-decode-execute cycle work?
27. What is meant by parallel computing?
28. What is the underlying premise of Amdahl's Law?

EXERCISES

◆1. In what ways are hardware and software different? In what ways are they the same?

2. a) How many milliseconds (ms) are in 1 second?
   b) How many microseconds (μs) are in 1 second?
   c) How many nanoseconds (ns) are in 1 millisecond?
   d) How many microseconds are in 1 millisecond?
   e) How many nanoseconds are in 1 microsecond?
   f) How many kilobytes (KB) are in 1 gigabyte (GB)?
   g) How many kilobytes are in 1 megabyte (MB)?
   h) How many megabytes are in 1 gigabyte (GB)?
   i) How many bytes are in 20 megabytes?
   j) How many kilobytes are in 2 gigabytes?

◆3. By what order of magnitude is something that runs in nanoseconds faster than something that runs in milliseconds?

4. Pretend you are ready to buy a new computer for personal use. First, take a look at ads from various magazines and newspapers and list terms you don't quite understand. Look these terms up and give a brief written explanation. Decide what factors are important in your decision as to which computer to buy and list them. After you select the system you would like to buy, identify which terms refer to hardware and which refer to software.

5. Pick your favorite computer language and write a small program. After compiling the program, see if you can determine the ratio of source code instructions to the machine language instructions generated by the compiler. If you add one line of source code, how does that affect the machine language program? Try adding different source code instructions, such as an add and then a multiply. How does the size of the machine code file change with the different instructions? Comment on the result.

6. Respond to the comment mentioned in Section 1.5: If invented today, what name do you think would be given to the computer? Give at least one good reason for your answer.

◆7. Suppose a transistor on an integrated circuit chip were 2 microns in size. According to Moore's Law, how large would that transistor be in 2 years? How is Moore's law relevant to programmers?

8. What circumstances helped the IBM PC become so successful?

9. List five applications of personal computers. Is there a limit to the applications of computers? Do you envision any radically different and exciting applications in the near future? If so, what?

10. Under the von Neumann architecture, a program and its data are both stored in memory. It is therefore possible for a program, thinking a memory location holds a piece of data when it actually holds a program instruction, to accidentally (or on purpose) modify itself. What implications does this present to you as a programmer?

11. Read a popular local newspaper and search through the job openings. (You can also check some of the more popular online career sites.) Which jobs require specific hardware knowledge? Which jobs imply knowledge of computer hardware? Is there any correlation between the required hardware knowledge and the company or its location?

"What would life be without arithmetic, but a scene of horrors?"

—Sydney Smith (1835)

CHAPTER 2 Data Representation in Computer Systems

2.1 INTRODUCTION

The organization of any computer depends considerably on how it represents numbers, characters, and control information. The converse is also true: Standards and conventions established over the years have determined certain aspects of computer organization. This chapter describes the various ways in which computers can store and manipulate numbers and characters. The ideas presented in the following sections form the basis for understanding the organization and function of all types of digital systems.

The most basic unit of information in a digital computer is called a *bit*, which is a contraction of *binary digit*. In the concrete sense, a bit is nothing more than a state of "on" or "off" (or "high" and "low") within a computer circuit. In 1964, the designers of the IBM System/360 mainframe computer established a convention of using groups of 8 bits as the basic unit of addressable computer storage. They called this collection of 8 bits a *byte*.

Computer *words* consist of two or more adjacent bytes that are sometimes addressed and almost always are manipulated collectively. The ***word size*** represents the data size that is handled most efficiently by a particular architecture. Words can be 16 bits, 32 bits, 64 bits, or any other size that makes sense within the context of a computer's organization (including sizes that are not multiples of eight). Eight-bit bytes can be divided into two 4-bit halves called *nibbles* (or *nybbles*). Because each bit of a byte has a value within a positional numbering system, the nibble containing the least-valued binary digit is called the low-order nibble, and the other half the high-order nibble.

109 LOAD ADD

2.2 POSITIONAL NUMBERING SYSTEMS

At some point during the middle of the sixteenth century, Europe embraced the decimal (or base 10) numbering system that the Arabs and Hindus had been using for nearly a millennium. Today, we take for granted that the number 243 means two hundreds, plus four tens, plus three units. Notwithstanding the fact that zero means "nothing," virtually everyone knows that there is a substantial difference between having 1 of something and having 10 of something.

The general idea behind positional numbering systems is that a numeric value is represented through increasing powers of a *radix* (or base). This is often referred to as a *weighted numbering system* because each position is weighted by a power of the radix.

The set of valid numerals for a positional numbering system is equal in size to the radix of that system. For example, there are 10 digits in the decimal system, 0 through 9, and 3 digits for the ternary (base 3) system, 0, 1, and 2. The largest valid number in a radix system is one smaller than the radix, so 8 is not a valid numeral in any radix system smaller than 9. To distinguish among numbers in different radices, we use the radix as a subscript, such as in 33_{10} to represent the decimal number 33. (In this book, numbers written without a subscript should be assumed to be decimal.) Any decimal integer can be expressed exactly in any other integral base system (see Example 2.1).

EXAMPLE 2.1 Three numbers represented as powers of a radix.

$$243.51_{10} = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2}$$
$$212_3 = 2 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 23_{10}$$
$$10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{10}$$

The two most important radices in computer science are binary (base two), and hexadecimal (base 16). Another radix of interest is octal (base 8). The binary system uses only the digits 0 and 1; the octal system, 0 through 7. The hexadecimal system allows the digits 0 through 9 with A, B, C, D, E, and F being used to represent the numbers 10 through 15. Figure 2.1 shows some of the radices.

2.3 DECIMAL TO BINARY CONVERSIONS

Gottfried Leibniz (1646–1716) was the first to generalize the idea of the (positional) decimal system to other bases. Being a deeply spiritual person, Leibniz attributed divine qualities to the binary system. He correlated the fact that any integer could be represented by a series of ones and zeros with the idea that God (1) created the universe out of nothing (0). Until the first binary digital computers were built in the late 1940s, this system remained nothing more than a mathematical curiosity. Today, it lies at the heart of virtually every electronic device that relies on digital controls.

Powers of 2
$2^{-2} = \frac{1}{4} = 0.25$
$2^{-1} = \frac{1}{2} = 0.5$
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1,024$
$2^{15} = 32,768$
$2^{16} = 65,536$

Decimal	4-Bit Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

FIGURE 2.1 Some Numbers to Remember

Because of its simplicity, the binary numbering system translates easily into electronic circuitry. It is also easy for humans to understand. Experienced computer professionals can recognize smaller binary numbers (such as those shown in Figure 2.1) at a glance. Converting larger values and fractions, however, usually requires a calculator or pencil and paper. Fortunately, the conversion techniques are easy to master with a little practice. We show a few of the simpler techniques in the sections that follow.

2.3.1 Converting Unsigned Whole Numbers

We begin with the base conversion of unsigned numbers. Conversion of signed numbers (numbers that can be positive or negative) is more complex, and it is important that you first understand the basic technique for conversion before continuing with signed numbers.

Conversion between base systems can be done by using either repeated subtraction or a division-remainder method. The subtraction method is cumbersome and requires a familiarity with the powers of the radix being used. Being the more intuitive of the two methods, however, we will explain it first.

As an example, let's say that we want to convert 104_{10} to base 3. We know that $3^4 = 81$ is the highest power of 3 that is less than 104, so our base 3 number will be 5 digits wide (one for each power of the radix: 0 through 4). We make note that 81 goes once into 104 and subtract, leaving a difference of 23. We know that the next power of 3, $3^3 = 27$, is too large to subtract, so we note the zero "placeholder" and look for how many times $3^2 = 9$ divides 23. We see that it goes twice and subtract 18. We are left with 5 from which we subtract $3^1 = 3$, leaving 2, which is 2×3^0. These steps are shown in Example 2.2.

EXAMPLE 2.2 Convert 104_{10} to base 3 using subtraction.

$$
\begin{array}{l}
104 \\
\underline{-81} = 3^4 \times 1 \\
23 \\
\underline{-0} = 3^3 \times 0 \\
23 \\
\underline{-18} = 3^2 \times 2 \\
5 \\
\underline{-3} = 3^1 \times 1 \\
2 \\
\underline{-2} = 3^0 \times 2 \\
0
\end{array}
$$

$104_{10} = 10212_3$

The division-remainder method is faster and easier than the repeated subtraction method. It employs the idea that successive divisions by the base are in fact successive subtractions by powers of the base. The remainders that we get when we sequentially divide by the base end up being the digits of the result, which are read from bottom to top. This method is illustrated in Example 2.3.

EXAMPLE 2.3 Convert 104_{10} to base 3 using the division-remainder method.

```
3 |104  2   3 divides 104 34 times with a remainder of 2
3 |34   1   3 divides 34 11 times with a remainder of 1
3 |11   2   3 divides 11 3 times with a remainder of 2
3 |3    0   3 divides 3 1 time with a remainder of 0
3 |1    1   3 divides 1 0 times with a remainder of 1
  0
```

Reading the remainders from bottom to top, we have: $104_{10} = 10212_3$.

This method works with any base, and because of the simplicity of the calculations, it is particularly useful in converting from decimal to binary. Example 2.4 shows such a conversion.

EXAMPLE 2.4 Convert 147_{10} to binary.

```
2 |147  1   2 divides 147 73 times with a remainder of 1
2 |73   1   2 divides 73 36 times with a remainder of 1
2 |36   0   2 divides 36 18 times with a remainder of 0
2 |18   0   2 divides 18 9 times with a remainder of 0
2 |9    1   2 divides 9 4 times with a remainder of 1
2 |4    0   2 divides 4 2 times with a remainder of 0
2 |2    0   2 divides 2 1 time with a remainder of 0
2 |1    1   2 divides 1 0 times with a remainder of 1
  0
```

Reading the remainders from bottom to top, we have: $147_{10} = 10010011_2$.

A binary number with N bits can represent unsigned integers from 0 to $2^N - 1$. For example, 4 bits can represent the decimal values 0 through 15, while 8 bits can represent the values 0 through 255. The range of values that can be represented by a given number of bits is extremely important when doing arithmetic operations on binary numbers. Consider a situation in which binary numbers are 4 bits in length, and we wish to add 1111_2 (15_{10}) to 1111_2. We know that 15 plus 15 is 30, but 30 cannot be represented using only 4 bits. This is an example of a condition known as *overflow*, which occurs in unsigned binary representation when the result of an arithmetic operation is outside the range of allowable precision for the given number of bits. We address overflow in more detail when discussing signed numbers in Section 2.4.

2.3.2 Converting Fractions

Fractions in any base system can be approximated in any other base system using negative powers of a radix. *Radix points* separate the integer part of a number from its fractional part. In the decimal system, the radix point is called a decimal point. Binary fractions have a binary point.

Fractions that contain repeating strings of digits to the right of the radix point in one base may not necessarily have a repeating sequence of digits in another base. For instance, ⅔ is a repeating decimal fraction, but in the ternary system it terminates as 0.2_3 ($2 \times 3^{-1} = 2 \times \frac{1}{3}$).

We can convert fractions between different bases using methods analogous to the repeated subtraction and division-remainder methods for converting integers. Example 2.5 shows how we can use repeated subtraction to convert a number from decimal to base 5.

EXAMPLE 2.5 Convert 0.4304_{10} to base 5.

$$
\begin{array}{l}
0.4304 \\
\underline{-\ 0.4000} = 5^{-1} \times 2 \\
0.0304 \\
\underline{-\ 0.0000} = 5^{-2} \times 0 \qquad \text{(A placeholder)} \\
0.0304 \\
\underline{-\ 0.0240} = 5^{-3} \times 3 \\
0.0064 \\
\underline{-\ 0.0064} = 5^{-4} \times 4 \\
0.0000
\end{array}
$$

Reading from top to bottom, we find $0.4304_{10} = 0.2034_5$.

Because the remainder method works with positive powers of the radix for conversion of integers, it stands to reason that we would use multiplication to convert fractions, because they are expressed in negative powers of the radix. However, instead of looking for remainders, as we did above, we use only the integer part of the product after multiplication by the radix. The answer is read from top to bottom instead of bottom to top. Example 2.6 illustrates the process.

EXAMPLE 2.6 Convert 0.4304_{10} to base 5.

$$
\begin{array}{ll}
.4304 & \\
\underline{\times\ \ 5} & \\
2.1520 & \text{The integer part is 2, omit from subsequent multiplication.} \\
.1520 & \\
\underline{\times\ \ 5} & \\
0.7600 & \text{The integer part is 0, we'll need it as a placeholder.} \\
.7600 & \\
\underline{\times\ \ 5} & \\
3.8000 & \text{The integer part is 3, omit from subsequent multiplication.} \\
.8000 & \\
\underline{\times\ \ 5} & \\
4.0000 & \text{The fractional part is now zero, so we are done.}
\end{array}
$$

Reading from top to bottom, we have $0.4304_{10} = 0.2034_5$.

This example was contrived so that the process would stop after a few steps. Often things don't work out quite so evenly, and we end up with repeating fractions. Most computer systems implement specialized rounding algorithms to provide a predictable degree of accuracy. For the sake of clarity, however, we will simply discard (or truncate) our answer when the desired accuracy has been achieved, as shown in Example 2.7.

EXAMPLE 2.7 Convert 0.34375_{10} to binary with 4 bits to the right of the binary point.

$$
\begin{array}{ll}
.34375 & \\
\underline{\times\ \ 2} & \\
0.68750 & \text{(Another placeholder.)} \\
.68750 & \\
\underline{\times\ \ 2} & \\
1.37500 & \\
.37500 & \\
\underline{\times\ \ 2} & \\
0.75000 & \\
.75000 & \\
\underline{\times\ \ 2} & \\
1.50000 & \text{(This is our fourth bit. We will stop here.)}
\end{array}
$$

Reading from top to bottom, $0.34375_{10} = 0.0101_2$ to four binary places.

The methods just described can be used to directly convert any number in any base to any other base, say from base 4 to base 3 (as in Example 2.8). However, in most cases, it is faster and more accurate to first convert to base 10 and then to the desired base. One exception to this rule is when you are working between bases that are powers of two, as you'll see in the next section.

EXAMPLE 2.8 Convert 3121_4 to base 3.

First, convert to decimal:

$$
\begin{aligned}
3121_4 &= 3 \times 4^3 + 1 \times 4^2 + 2 \times 4^1 + 1 \times 4^0 \\
&= 3 \times 64 + 1 \times 16 + 2 \times 4 + 4 = 217_{10}
\end{aligned}
$$

Then convert to base 3:

$$
\begin{array}{rl}
3\ |217 & 1 \\
3\ |72 & 0 \\
3\ |24 & 0 \\
3\ |8 & 2 \\
3\ |2 & 2 \\
0 & \text{We have } 3121_4 = 22001_3.
\end{array}
$$

2.3.3 Converting between Power-of-Two Radices

Binary numbers are often expressed in hexadecimal—and sometimes octal—to improve their readability. Because $16 = 2^4$, a group of 4 bits (called a *hextet*) is easily recognized as a hexadecimal digit. Similarly, with $8 = 2^3$, a group of 3 bits (called an *octet*) is expressible as one octal digit. Using these relationships, we can therefore convert a number from binary to octal or hexadecimal by doing little more than looking at it.

EXAMPLE 2.9 Convert 1100100111101_2 to octal and hexadecimal.

110	010	011	101	Separate into groups of three for the octal conversion.
6	2	3	5	

$1100100111101_2 = 6235_8$

1100	1001	1101	Separate into groups of 4 for the hexadecimal conversion.
C	9	D	

$1100100111101_2 = C9D_{16}$

If there are too few bits, leading zeros can be added.

2.4 SIGNED INTEGER REPRESENTATION

We have seen how to convert an unsigned integer from one base to another. Signed numbers require additional issues to be addressed. When an integer variable is declared in a program, many programming languages automatically allocate a storage area that includes a sign as the first bit of the storage location. By convention, a "1" in the high-order bit indicates a negative number. The storage location can be as small as an 8-bit byte or as large as several words, depending on the programming language and the computer system. The remaining bits (after the sign bit) are used to represent the number itself.

How this number is represented depends on the method used. There are three commonly used approaches. The most intuitive method, signed magnitude, uses the remaining bits to represent the magnitude of the number. This method and the other two approaches, which both use the concept of *complements*, are introduced in the following sections.

2.4.1 Signed Magnitude

Up to this point, we have ignored the possibility of binary representations for negative numbers. The set of positive and negative integers is referred to as the set of *signed integers*. The problem with representing signed integers as binary values is the sign—how should we encode the actual sign of the number? *Signed-magnitude representation* is one method of solving this problem. As its name implies, a signed-magnitude number has a sign as its left-most bit (also referred to as the high-order bit or the most significant bit) while the remaining bits represent the magnitude (or absolute value) of the numeric value. For example, in an 8-bit word, −1 would be represented as 10000001, and +1 as 00000001. In a computer system that uses signed-magnitude representation and 8 bits to store integers, 7 bits can be used for the actual representation of the magnitude of the number. This means that the largest integer an 8-bit word can represent is $2^7 - 1$ or 127 (a zero in the high-order bit, followed by 7 ones). The smallest integer is 8 ones, or −127. Therefore, N bits can represent $-2^{(N-1)} - 1$ to $2^{(N-1)} - 1$.

Computers must be able to perform arithmetic calculations on integers that are represented using this notation. Signed-magnitude arithmetic is carried out using essentially the same methods as humans use with pencil and paper, but it can get confusing very quickly. As an example, consider the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one. If you consider these rules carefully, this is the method you use for signed arithmetic by hand.

We arrange the operands in a certain way based on their signs, perform the calculation without regard to the signs, and then supply the sign as appropriate when the calculation is complete. When modeling this idea in an 8-bit word, we must be careful to include only 7 bits in the magnitude of the answer, discarding any carries that take place over the high-order bit.

EXAMPLE 2.10 Add 01001111_2 to 00100011_2 using signed-magnitude arithmetic.

			1	1	1	1	⇐ carries	
0	1	0	0	1	1	1	1	(79)
0 +	0	1	0	0	0	1	1	+ (35)
0	1	1	1	0	0	1	0	(114)

The arithmetic proceeds just as in decimal addition, including the carries, until we get to the seventh bit from the right. If there is a carry here, we say that we have an overflow condition and the carry is discarded, resulting in an incorrect sum. There is no overflow in this example.

We find $01001111_2 + 00100011_2 = 01110010_2$ in signed-magnitude representation.

Sign bits are segregated because they are relevant only after the addition is complete. In this case, we have the sum of two positive numbers, which is positive. *Overflow* (and thus an erroneous result) in signed numbers occurs when the sign of the result is incorrect.

In signed magnitude, the sign bit is used only for the sign, so we can't "carry into" it. If there is a carry emitting from the seventh bit, our result will be truncated as the seventh bit overflows, giving an incorrect sum. (Example 2.11 illustrates this overflow condition.) Prudent programmers avoid "million dollar" mistakes by checking for overflow conditions whenever there is the slightest possibility that they could occur. If we did not discard the overflow bit, it would carry into the sign, causing the more outrageous result of the sum of two positive numbers being negative. (Imagine what would happen if the next step in a program were to take the square root or log of that result!)

EXAMPLE 2.11 Add 01001111_2 to 01100011_2 using signed-magnitude arithmetic.

Last carry	$1 \leftarrow$	1 1 1 1	$\Leftarrow$ carries
overflows and is	0	1 0 0 1 1 1 1	(79)
discarded.	0 +	1 1 0 0 0 1 1	+ (99)
	0	0 1 1 0 0 1 0	(50)

We obtain the erroneous result of 79 + 99 = 50.

What Is Double-Dabble?

The fastest way to convert a binary number to decimal is a method called *double-dabble* (or *double-dibble*). This method builds on the idea that a subsequent power of two is double the previous power of two in a binary number. The calculation starts with the leftmost bit and works toward the rightmost bit. The first bit is doubled and added to the next bit. This sum is then doubled and added to the following bit. The process is repeated for each bit until the rightmost bit has been used.

EXAMPLE 1

Convert 10010011_2 to decimal.

Step 1: Write down the binary number, leaving space between the bits.

1 0 0 1 0 0 1 1

Step 2: Double the high-order bit and copy it under the next bit.

1 0 0 1 0 0 1 1
2
× 2
2

Step 3: Add the next bit and double the sum. Copy this result under the next bit.

1 0 0 1 0 0 1 1
2 4
+ 0
2
× 2 × 2
2 4

Step 4: Repeat Step 3 until you run out of bits.

1	0	0	1	0	0	1	1
	2	4	8	18	36	72	146
	+ 0	+ 0	+ 1	+ 0	+ 0	+ 1	+ 1
	2	4	9	18	36	73	147
× 2	× 2	× 2	× 2	× 2	× 2	× 2	
2	4	8	18	36	72	146	

⇐ The answer: $10010011_2 = 147_{10}$

When we combine hextet grouping (in reverse) with the double-dabble method, we find that we can convert hexadecimal to decimal with ease.

EXAMPLE 2

Convert $02CA_{16}$ to decimal.

First, convert the hex to binary by grouping into hextets.

0	2	C	A
0000	0010	1100	1010

Then apply the double-dabble method on the binary form:

1	0	1	1	0	0	1	0	1	0
	2	4	10	22	44	88	178	356	714
	+ 0	+ 1	+ 1	+ 0	+ 0	+ 1	+ 0	+ 1	+ 0
	2	5	11	22	44	89	178	357	714
× 2	× 2	× 2	× 2	× 2	× 2	× 2	× 2	× 2	
2	4	10	22	44	88	178	356	714	

$02CA_{16} = 1011001010_2 = 714_{10}$

As with addition, signed-magnitude subtraction is carried out in a manner similar to pencil and paper decimal arithmetic, where it is sometimes necessary to borrow from digits in the *minuend*.

EXAMPLE 2.12 Subtract 01001111_2 from 01100011_2 using signed-magnitude arithmetic.

```
          0 1 1 2        ⇐ borrows
  0     1 ✝ 0 0 0 1 1      (99)
  0  −  1 0 0 1 1 1 1   −  (79)
  0     0 0 1 0 1 0 0      (20)
```

We find $01100011_2 - 01001111_2 = 00010100_2$ in signed-magnitude representation.

EXAMPLE 2.13 Subtract 01100011_2 (99) from 01001111_2 (79) using signed-magnitude arithmetic.

By inspection, we see that the subtrahend, 01100011, is larger than the minuend, 01001111. With the result obtained in Example 2.12, we know that the difference of these two numbers is 0010100_2. Because the subtrahend is larger than the minuend, all that we need to do is change the sign of the difference. So we find $01001111_2 - 01100011_2 = 10010100_2$ in signed-magnitude representation.

We know that subtraction is the same as "adding the opposite," which equates to negating the value we wish to subtract and then adding instead (which is often much easier than performing all the borrows necessary for subtraction, particularly in dealing with binary numbers). Therefore, we need to look at some examples involving both positive and negative numbers. Recall the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one.

EXAMPLE 2.14 Add 10010011_2 (−19) to 00001101_2 (+13) using signed-magnitude arithmetic.

The first number (the augend) is negative because its sign bit is set to 1. The second number (the addend) is positive. What we are asked to do is in fact a subtraction. First, we determine which of the two numbers is larger in magnitude and use that number for the augend. Its sign will be the sign of the result.

```
          0 1 2          ⇐ borrows
  1     0 0 ✝ 0 0 1 1      (−19)
  0  −  0 0 0 1 1 0 1   +  (13)
  1     0 0 0 0 1 1 0      (−6)
```


With the inclusion of the sign bit, we see that $10010011_2 - 00001101_2 = 10000110_2$ in signed-magnitude representation.

EXAMPLE 2.15 Subtract 10011000_2 (−24) from 10101011_2 (−43) using signed-magnitude arithmetic.

We can convert the subtraction to an addition by negating −24, which gives us 24, and then we can add this to −43, giving us a new problem of −43 + 24. However, we know from the addition rules above that because the signs now differ, we must actually subtract the smaller magnitude from the larger magnitude (or subtract 24 from 43) and make the result negative (since 43 is larger than 24).

```
          0 2
  0     0 ✝ 0 1 0 1 1      (43)
  −     0 0 1 1 0 0 0   −  (24)
        0 0 1 0 0 1 1      (19)
```

Note that we are not concerned with the sign until we have performed the subtraction. We know the answer must be positive. So we end up with $10101011_2 - 10001100_2 = 00010011_2$ in signed-magnitude representation.

While reading the preceding examples, you may have noticed how many questions we had to ask ourselves: Which number is larger? Am I subtracting a negative number? How many times do I have to borrow from the minuend? A computer engineered to perform arithmetic in this manner must make just as many decisions (though a whole lot faster). The logic (and circuitry) is further complicated by the fact that signed magnitude has two representations for zero, 10000000 and 00000000 (and mathematically speaking, this simply shouldn't happen!). Simpler methods for representing signed numbers would allow simpler and less expensive circuits. These simpler methods are based on radix complement systems.

2.4.2 Complement Systems

Number theorists have known for hundreds of years that one decimal number can be subtracted from another by adding the difference of the subtrahend from all nines and adding back a carry. This is called taking the nine's complement of the subtrahend, or more formally, finding the *diminished radix complement* of the subtrahend. Let's say we wanted to find 167 − 52. Taking the difference of 52 from 999, we have 947. Thus, in nine's complement arithmetic we have 167 − 52 = 167 + 947 = 114. The "carry" from the hundreds column is added back to the units place, giving us a correct 167 − 52 = 115. This method was commonly called "casting out 9s" and has been extended to binary operations to simplify computer arithmetic. The advantage that complement systems give us over signed magnitude is that there is no need to process sign bits separately, but we can still easily check the sign of a number by looking at its high-order bit.

Another way to envision complement systems is to imagine an odometer on a bicycle. Unlike cars, when you go backward on a bike, the odometer will go backward as well. Assuming an odometer with three digits, if we start at zero and end with 700, we can't be sure whether the bike went forward 700 miles or backward 300 miles! The easiest solution to this dilemma is simply to cut the number space in half and use 001–500 for positive miles and 501–999 for negative miles. We have, effectively, cut down the distance our odometer can measure. But now if it reads 997, we know the bike has backed up 3 miles instead of riding forward 997 miles. The numbers 501–999 represent the *radix complements* (the second of the two methods introduced below) of the numbers 001–500 and are being used to represent negative distance.

One's Complement

As illustrated above, the diminished radix complement of a number in base 10 is found by subtracting the subtrahend from the base minus one, which is 9 in decimal. More formally, given a number N in base r having d digits, the diminished radix complement of N is defined to be $(r^d - 1) - N$. For decimal numbers, $r = 10$, and the diminished radix is $10 - 1 = 9$. For example, the nine's complement of 2468 is $9999 - 2468 = 7531$. For an equivalent operation in binary, we subtract from one less the base (2), which is 1. For example, the one's complement of 0101_2 is $1111_2 - 0101 = 1010$. Although we could tediously borrow and subtract as discussed above, a few experiments will convince you that forming the one's complement of a binary number amounts to nothing more than switching all of the 1s with 0s and vice versa. This sort of bit-flipping is very simple to implement in computer hardware.

It is important to note at this point that although we can find the nine's complement of any decimal number or the one's complement of any binary number, we are most interested in using complement notation to represent negative numbers. We know that performing a subtraction, such as $10 - 7$, can be also be thought of as "adding the opposite," as in $10 + (-7)$. Complement notation allows us to simplify subtraction by turning it into addition, but it also gives us a method to represent negative numbers. Because we do not wish to use a special bit to represent the sign (as we did in signed-magnitude representation), we need to remember that if a number is negative, we should convert it to its complement. The result should have a 1 in the leftmost bit position to indicate the number is negative. If the number is positive, we do not have to convert it to its complement. All positive numbers should have a zero in the leftmost bit position. Example 2.16 illustrates these concepts.

EXAMPLE 2.16 Express 23_{10} and -9_{10} in 8-bit binary one's complement form.

$23_{10} = +(00010111_2) = 00010111_2$
$-9_{10} = -(00001001_2) = 11110110_2$

Suppose we wish to subtract 9 from 23. To carry out a one's complement subtraction, we first express the subtrahend (9) in one's complement, then add it to the minuend (23); we are effectively now adding -9 to 23. The high-order bit will have a 1 or a 0 carry, which is added to the low-order bit of the sum. (This is called *end carry-around* and results from using the diminished radix complement.)

EXAMPLE 2.17 Add 23_{10} to -9_{10} using one's complement arithmetic.

	$1 \leftarrow 1\ 1\ 1\ \ 1\ 1$	⇐ carries
	$0\ 0\ 0\ 1\ 0\ 1\ 1\ 1$	(23)
The last	$+\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0$	$+(-9)$
carry is added	$0\ 0\ 0\ 0\ 1\ 1\ 0\ 1$	
	$+\ 1$	
to the sum.	$0\ 0\ 0\ 0\ 1\ 1\ 1\ 0$	14_{10}

EXAMPLE 2.18 Add 9_{10} to -23_{10} using one's complement arithmetic.

The last	$0 \leftarrow 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1$	(9)
carry is zero	$+\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0$	$+(-23)$
so we are done.	$1\ 1\ 1\ 1\ 0\ 0\ 0\ 1$	-14_{10}

How do we know that 11110001_2 is actually -14_{10}? We simply need to take the one's complement of this binary number (remembering it must be negative because the leftmost bit is negative). The one's complement of 11110001_2 is 00001110_2, which is 14.

The primary disadvantage of one's complement is that we still have two representations for zero: 00000000 and 11111111. For this and other reasons, computer engineers long ago stopped using one's complement in favor of the more efficient two's complement representation for binary numbers.

Two's Complement

Two's complement is an example of a *radix complement*. Given a number N in base r having d digits, the radix complement of N is defined to be $r^d - N$ for $N \neq 0$ and 0 for $N = 0$. The radix complement is often more intuitive than the diminished radix complement. Using our odometer example, the ten's complement of going forward 2 miles is $10^2 - 2 = 998$, which we have already agreed indicates a negative (backward) distance. Similarly, in binary, the two's complement of the 4-bit number 0011_2 is $2^4 - 0011_2 = 10000_2 - 0011_2 = 1101_2$.

Upon closer examination, you will discover that two's complement is nothing more than one's complement incremented by 1. To find the two's complement of a binary number, simply flip bits and add 1. This simplifies addition and subtraction

as well. Since the subtrahend (the number we complement and add) is incremented at the outset, however, there is no end carry-around to worry about. We simply discard any carries involving the high-order bits. Remember, only negative numbers need to be converted to two's complement notation, as indicated in Example 2.19.

EXAMPLE 2.19 Express 23_{10}, -23_{10}, and -9_{10} in 8-bit binary two's complement form.

$$23_{10} = +(00010111_2) = 00010111_2$$
$$-23_{10} = -(00010111_2) = 11101000_2 + 1 = 11101001_2$$
$$-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$$

Suppose we are given the binary representation for a number and want to know its decimal equivalent? Positive numbers are easy. For example, to convert the two's complement value of 00010111_2 to decimal, we simply convert this binary number to a decimal number to get 23. However, converting two's complement negative numbers requires a reverse procedure similar to the conversion from decimal to binary. Suppose we are given the two's complement binary value of 11110111_2, and we want to know the decimal equivalent. We know this is a negative number but must remember it is represented using two's complement. We first flip the bits and then add 1 (find the one's complement and add 1). This results in the following: $00001000_2 + 1 = 00001001_2$. This is equivalent to the decimal value 9. However, the original number we started with was negative, so we end up with -9 as the decimal equivalent to 11110111_2.

The following two examples illustrate how to perform addition (and hence subtraction, because we subtract a number by adding its opposite) using two's complement notation.

EXAMPLE 2.20 Add 9_{10} to -23_{10} using two's complement arithmetic.

```
  0 0 0 0 1 0 0 1     (9)
+ 1 1 1 0 1 0 0 1  +(–23)
  1 1 1 1 0 0 1 0   –14₁₀
```


It is left as an exercise for you to verify that 11110010_2 is actually -14_{10} using two's complement notation.

EXAMPLE 2.21 Find the sum of 23_{10} and -9_{10} in binary using two's complement arithmetic.

```
          1←  1 1 1    1 1 1   ⇐ carries
Discard       0 0 0 1 0 1 1 1      (23)
carry.    +   1 1 1 1 0 1 1 1   +(–9)
          ---------------------------
          1 0 0 0 1 1 1 0      14₁₀
```

Notice that the discarded carry in Example 2.21 did not cause an erroneous result. An overflow occurs if two positive numbers are added and the result is negative, or if two negative numbers are added and the result is positive. It is not possible to have overflow when using two's complement notation if a positive and a negative number are being added together.

Simple computer circuits can easily detect an overflow condition using a rule that is easy to remember. You'll notice in Example 2.21 that the carry going into the sign bit (a 1 is carried from the previous bit position into the sign bit position) is the same as the carry going out of the sign bit (a 1 is carried out and discarded). When these carries are equal, no overflow occurs. When they differ, an overflow indicator is set in the arithmetic logic unit, indicating the result is incorrect.

> **A Simple Rule for Detecting an Overflow Condition:** *If the carry into the sign bit equals the carry out of the bit, no overflow has occurred. If the carry into the sign bit is different from the carry out of the sign bit, overflow (and thus an error) has occurred.*

The hard part is getting programmers (or compilers) to consistently check for the overflow condition. Example 2.22 indicates overflow because the carry into the sign bit (a 1 is carried in) is not equal to the carry out of the sign bit (a 0 is carried out).

EXAMPLE 2.22 Find the sum of 126_{10} and 8_{10} in binary using two's complement arithmetic.

```
          0←  1 1 1 1          ⇐ carries
Discard last  0 1 1 1 1 1 1 0      (126)
carry.    +   0 0 0 0 1 0 0 0    +(8)
          ---------------------------
          1 0 0 0 0 1 1 0   (–122???)
```

INTEGER MULTIPLICATION AND DIVISION

Unless sophisticated algorithms are used, multiplication and division can consume a considerable number of computation cycles before a result is obtained. Here, we discuss only the most straightforward approach to these operations. In real systems, dedicated hardware is used to optimize throughput, sometimes carrying out portions of the calculation in parallel. Curious readers will want to investigate some of these advanced methods in the references cited at the end of this chapter.

The simplest multiplication algorithms used by computers are similar to traditional pencil and paper methods used by humans. The complete multiplication table for binary numbers couldn't be simpler: zero times any number is zero, and one times any number is that number.

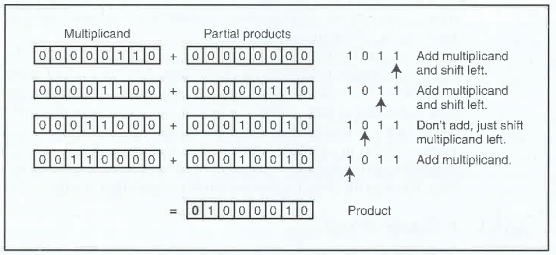
To illustrate simple computer multiplication, we begin by writing the multiplicand and the multiplier to two separate storage areas. We also need a third storage area for the product. Starting with the low-order bit, a pointer is set to each digit of the multiplier. For each digit in the multiplier, the multiplicand is "shifted" one bit to the left. When the multiplier is 1, the "shifted" multiplicand is added to a running sum of partial products. Because we shift the multiplicand by one bit for each bit in the multiplier, a product requires double the working space of either the multiplicand or the multiplier.

There are two simple approaches to binary division: We can either iteratively subtract the denominator from the divisor, or we can use the same trial-and-error method of long division that we were taught in grade school. As mentioned above with multiplication, the most efficient methods used for binary division are beyond the scope of this text and can be found in the references at the end of this chapter.

Regardless of the relative efficiency of any algorithms that are used, division is an operation that can always cause a computer to crash. This is the case particularly when division by zero is attempted or when two numbers of enormously different magnitudes are used as operands. When the divisor is much smaller than the dividend, we get a condition known as *divide underflow*, which the computer sees as the equivalent of division by zero, which is impossible.

Computers make a distinction between integer division and floating-point division. With integer division, the answer comes in two parts: a quotient and a remainder. Floating-point division results in a number that is expressed as a binary fraction. These two types of division are sufficiently different from each other as to warrant giving each its own special circuitry. Floating-point calculations are carried out in dedicated circuits called *floating-point units*, or *FPUs*.

EXAMPLE Find the product of 00000110_2 and 00001011_2.



A one is carried into the leftmost bit, but a zero is carried out. Because these carries are not equal, an overflow has occurred. (We can easily see that two positive numbers are being added but the result is negative.)

Two's complement is the most popular choice for representing signed numbers. The algorithm for adding and subtracting is quite easy, has the best representation for 0 (all 0 bits), is self-inverting, and is easily extended to larger numbers of bits. The biggest drawback is in the asymmetry seen in the range of values that can be represented by N bits. With signed-magnitude numbers, for example, 4 bits allow us to represent the values -7 through $+7$. However, using two's complement, we can represent the values -8 through $+7$, which is often confusing to anyone learning about complement representations. To see why $+7$ is the largest number we can represent using 4-bit two's complement representation, we need only remember the first bit must be 0. If the remaining bits are all 1s (giving us the largest magnitude possible), we have 0111_2, which is 7. An immediate reaction to this is that the smallest negative number should then be 1111_2, but we can see that 1111_2 is actually -1 (flip the bits, add one, and make the number negative). So how do we represent -8 in two's complement notation using 4 bits? It is represented as 1000_2. We know this is a negative number. If we flip the bits (0111), add 1 (to get 1000, which is 8), and make it negative, we get -8.

2.5 FLOATING-POINT REPRESENTATION

If we wanted to build a real computer, we could use any of the integer representations that we just studied. We would pick one of them and proceed with our design tasks. Our next step would be to decide the word size of our system. If we want our

system to be really inexpensive, we would pick a small word size, say 16 bits. Allowing for the sign bit, the largest integer that this system can store is 32,767. So now what do we do to accommodate a potential customer who wants to keep a tally of the number of spectators paying admission to professional sports events in a given year? Certainly, the number is larger than 32,767. No problem. Let's just make the word size larger. Thirty-two bits ought to do it. Our word is now big enough for just about anything that anyone wants to count. But what if this customer also needs to know the amount of money each spectator spends per minute of playing time? This number is likely to be a decimal fraction. Now we're really stuck.

The easiest and cheapest approach to this problem is to keep our 16-bit system and say, "Hey, we're building a cheap system here. If you want to do fancy things with it, get yourself a good programmer." Although this position sounds outrageously flippant in the context of today's technology, it was a reality in the earliest days of each generation of computers. There simply was no such thing as a floating-point unit in many of the first mainframes or microcomputers. For many years, clever programming enabled these integer systems to act as if they were, in fact, floating-point systems.

If you are familiar with scientific notation, you may already be thinking of how you could handle floating-point operations—how you could provide *floating-point emulation*—in an integer system. In scientific notation, numbers are expressed in two parts: a fractional part, called a *mantissa*, and an exponential part that indicates the power of ten to which the mantissa should be raised to obtain the value we need. So to express 32,767 in scientific notation, we could write 3.2767×10^4. Scientific notation simplifies pencil and paper calculations that involve very large or very small numbers. It is also the basis for floating-point computation in today's digital computers.

2.5.1 A Simple Model

In digital computers, floating-point numbers consist of three parts: a sign bit, an exponent part (representing the exponent on a power of 2), and a fractional part called a *significand* (which is a fancy word for a mantissa). The number of bits used for the exponent and significand depends on whether we would like to optimize for range (more bits in the exponent) or precision (more bits in the significand). For the remainder of this section, we will use a 14-bit model with a 5-bit exponent, an 8-bit significand, and a sign bit (see Figure 2.2). More general forms are described in Section 2.5.2.

Let's say that we wish to store the decimal number 17 in our model. We know that $17 = 17.0 \times 10^0 = 1.7 \times 10^1 = 0.17 \times 10^2$. Analogously, in binary, $17_{10} = 10001_2 \times 2^0 = 1000.1_2 \times 2^1 = 100.01_2 \times 2^2 = 10.001_2 \times 2^3 = 1.0001_2 \times 2^4 =$



FIGURE 2.2 Floating-Point Representation

$0.10001_2 \times 2^5$. If we use this last form, our fractional part will be 10001000 and our exponent will be 00101, as shown here:



Using this form, we can store numbers of much greater magnitude than we could using a *fixed-point* representation of 14 bits (which uses a total of 14 binary digits plus a binary, or radix, point). If we want to represent $65536 = 0.1_2 \times 2^{17}$ in this model, we have:



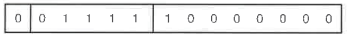
One obvious problem with this model is that we haven't provided for negative exponents. If we wanted to store 0.25 we would have no way of doing so because 0.25 is 2^{-2} and the exponent -2 cannot be represented. We could fix the problem by adding a sign bit to the exponent, but it turns out that it is more efficient to use a *biased* exponent, because we can use simpler integer circuits when comparing the values of two floating-point numbers.

The idea behind using a bias value is to convert every integer in the range into a non-negative integer, which is then stored as a binary numeral. The integers in the desired range of exponents are first adjusted by adding this fixed bias value to each exponent. The bias value is a number near the middle of the range of possible values that we select to represent zero. In this case, we could select 16 because it is midway between 0 and 31 (our exponent has 5 bits, thus allowing for 2^5 or 32 values). Any number larger than 16 in the exponent field will represent a positive value. Values less than 16 will indicate negative values. This is called an *excess-16* representation because we have to subtract 16 to get the true value of the exponent. Note that exponents of all zeros or all ones are typically reserved for special numbers (such as zero or infinity). In our simple model, we allow exponents of all zeros and ones.

Returning to our example of storing 17, we calculated $17_{10} = 0.10001_2 \times 2^5$. The biased exponent is now $16 + 5 = 21$:



If we wanted to store $0.25 = 0.1 \times 2^{-1}$ we would have:



There is still one rather large problem with this system: We do not have a unique representation for each number. All of the following are equivalent:

"Every program has at least one bug and can be shortened by at least one instruction—from which, by induction, one can deduce that every program can be reduced to one instruction which doesn't work."

—Anonymous

CHAPTER 5 A Closer Look at Instruction Set Architectures

5.1 INTRODUCTION

We saw in Chapter 4 that machine instructions consist of opcodes and operands. The opcodes specify the operations to be executed; the operands specify register or memory locations of data. Why, when we have languages such as C++, Java, and Ada available, should we be concerned with machine instructions? When programming in a high-level language, we frequently have little awareness of the topics discussed in Chapter 4 (or in this chapter) because high-level languages hide the details of the architecture from the programmer. Employers frequently prefer to hire people with assembly language backgrounds not because they need an assembly language programmer, but because they need someone who can understand computer architecture to write more efficient and more effective programs.

In this chapter, we expand on the topics presented in the last chapter, the objective being to provide you with a more detailed look at machine instruction sets. We look at different instruction types and operand types, and how instructions access data in memory. You will see that the variations in instruction sets are integral in distinguishing different computer architectures. Understanding how instruction sets are designed and how they function can help you understand the more intricate details of the architecture of the machine itself.

5.2 INSTRUCTION FORMATS

We know that a machine instruction has an opcode and zero or more operands. In Chapter 4 we saw that MARIE had an instruction length of 16 bits and could have,

at most, 1 operand. Encoding an instruction set can be done in a variety of ways. Architectures are differentiated from one another by the number of bits allowed per instruction (16, 32, and 64 are the most common), by the number of operands allowed per instruction, and by the types of instructions and data each can process. More specifically, instruction sets are differentiated by the following features:

- Operand storage in the CPU (data can be stored in a stack structure or in registers)
- Number of explicit operands per instruction (zero, one, two, and three being the most common)
- Operand location (instructions can be classified as register-to-register, register-to-memory or memory-to-memory, which simply refer to the combinations of operands allowed per instruction)
- Operations (including not only types of operations but also which instructions can access memory and which cannot)
- Type and size of operands (operands can be addresses, numbers, or even characters)

5.2.1 Design Decisions for Instruction Sets

When a computer architecture is in the design phase, the instruction set format must be determined before many other decisions can be made. Selecting this format is often quite difficult because the instruction set must match the architecture, and the architecture, if well designed, could last for decades. Decisions made during the design phase have long-lasting ramifications.

Instruction set architectures (ISAs) are measured by several different factors, including: (1) the amount of space a program requires; (2) the complexity of the instruction set, in terms of the amount of decoding necessary to execute an instruction, and the complexity of the tasks performed by the instructions; (3) the length of the instructions; and (4) the total number of instructions. Things to consider when designing an instruction set include:

- Short instructions are typically better because they take up less space in memory and can be fetched quickly. However, this limits the number of instructions, because there must be enough bits in the instruction to specify the number of instructions we need. Shorter instructions also have tighter limits on the size and number of operands.
- Instructions of a fixed length are easier to decode but waste space.
- Memory organization affects instruction format. If memory has, for example, 16- or 32-bit words and is not byte-addressable, it is difficult to access a single character. For this reason, even machines that have 16-, 32-, or 64-bit words are often byte-addressable, meaning every byte has a unique address even though words are longer than 1 byte.
- A fixed length instruction does not necessarily imply a fixed number of operands. We could design an ISA with fixed overall instruction length, but allow the number of bits in the operand field to vary as necessary. (This is called an *expanding opcode* and is covered in more detail in Section 5.2.5.)

- There are many different types of addressing modes. In Chapter 4, MARIE used two addressing modes: direct and indirect; however, we see in this chapter that a large variety of addressing modes exist.
- If words consist of multiple bytes, in what order should these bytes be stored on a byte-addressable machine? Should the least significant byte be stored at the highest or lowest byte address? This little versus big endian debate is discussed in the following section.
- How many registers should the architecture contain and how should these registers be organized? How should operands be stored in the CPU?

The little versus big endian debate, expanding opcodes, and CPU register organization are examined further in the following sections. In the process of discussing these topics, we also touch on the other design issues listed.

5.2.2 Little versus Big Endian

The term *endian* refers to a computer architecture's "byte order," or the way the computer stores the bytes of a multiple-byte data element. Virtually all computer architectures today are byte-addressable and must, therefore, have a standard for storing information requiring more than a single byte. Some machines store a two-byte integer, for example, with the least significant byte first (at the lower address) followed by the most significant byte. Therefore, a byte at a lower address has lower significance. These machines are called *little endian* machines. Other machines store this same two-byte integer with its most significant byte first, followed by its least significant byte. These are called *big endian* machines because they store the most significant bytes at the lower addresses. Most UNIX machines are big endian, whereas most PCs are little endian machines. Most newer RISC architectures are also big endian.

These two terms, little and big endian, are from the book *Gulliver's Travels*. You may remember the story in which the Lilliputians (the tiny people) were divided into two camps: those who ate their eggs by opening the "big" end (big endians) and those who ate their eggs by opening the "little" end (little endians). CPU manufacturers are also divided into two factions. For example, Intel has always done things the "little endian" way whereas Motorola has always done things the "big endian" way. (It is also worth noting that some CPUs can handle both little and big endian.)

For example, consider an integer requiring 4 bytes:



On a little endian machine, this is arranged in memory as follows:

```
Base Address + 0 = Byte0
Base Address + 1 = Byte1
Base Address + 2 = Byte2
Base Address + 3 = Byte3
```

On a big endian machine, this long integer would then be stored as:

```
Base Address + 0 = Byte3
Base Address + 1 = Byte2
Base Address + 2 = Byte1
Base Address + 3 = Byte0
```

Let's assume that on a byte-addressable machine, the 32-bit hex value 12345678 is stored at address 0. Each digit requires a nibble, so one byte holds two digits. This hex value is stored in memory as shown in Figure 5.1, where the shaded cells represent the actual contents of memory.

There are advantages and disadvantages to each method, although one method is not necessarily better than the other. Big endian is more natural to most people and thus makes it easier to read hex dumps. By having the high-order byte come first, you can always test whether the number is positive or negative by looking at the byte at offset zero. (Compare this to little endian where you must know how long the number is and then must skip over bytes to find the one containing the sign information.) Big endian machines store integers and strings in the same order and are faster in certain string operations. Most bitmapped graphics are mapped with a "most significant bit on the left" scheme, which means working with graphical elements larger than one byte can be handled by the architecture itself. This is a performance limitation for little endian computers because they must continually reverse the byte order when working with large graphical objects. When decoding compressed data encoded with such schemes as Huffman and LZW (discussed in Chapter 7), the actual codeword can be used as an index into a lookup table if it is stored in big endian (this is also true for encoding).

However, big endian also has disadvantages. Conversion from a 32-bit integer address to a 16-bit integer address requires a big endian machine to perform addition. High-precision arithmetic on little endian machines is faster and easier. Most architectures using the big endian scheme do not allow words to be written on non-word address boundaries (for example, if a word is 2 or 4 bytes, it must always begin on an even-numbered byte address). This wastes space. Little endian architectures, such as Intel, allow odd address reads and writes, which makes programming on these machines much easier. If a programmer writes an instruction to read a value of the wrong word size, on a big endian machine it is always read as an incorrect value; on a little endian machine, it can sometimes result in the correct data being read. (Note that Intel finally has added an instruction to reverse the byte order within registers.)

Address ⟶	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

FIGURE 5.1 The Hex Value 12345678 Stored in Both Big and Little Endian Format

Computer networks are big endian, which means that when little endian computers are going to pass integers over the network (network device addresses, for example), they need to convert them to network byte order. Likewise, when they receive integer values over the network, they need to convert them back to their own native representation.

Although you may not be familiar with this little versus big endian debate, it is important to many current software applications. Any program that writes data to or reads data from a file must be aware of the byte ordering on the particular machine. For example, the Windows BMP graphics format was developed on a little endian machine, so to view BMPs on a big endian machine, the application used to view them must first reverse the byte order. Software designers of popular software are well aware of these byte-ordering issues. For example, Adobe Photoshop uses big endian, GIF is little endian, JPEG is big endian, MacPaint is big endian, PC Paintbrush is little endian, RTF by Microsoft is little endian, and Sun raster files are big endian. Some applications support both formats: Microsoft WAV and AVI files, TIFF files, and XWD (X windows Dump) support both, typically by encoding an identifier into the file.

5.2.3 Internal Storage in the CPU: Stacks versus Registers

Once byte ordering in memory is determined, the hardware designer must make some decisions on how the CPU should store data. This is the most basic means to differentiate ISAs. There are three choices:

1. A stack architecture
2. An accumulator architecture
3. A general purpose register (GPR) architecture

Stack architectures use a stack to execute instructions, and the operands are (implicitly) found on top of the stack. Even though stack-based machines have good code density and a simple model for evaluation of expressions, a stack cannot be accessed randomly, which makes it difficult to generate efficient code. *Accumulator architectures* such as MARIE, with one operand implicitly in the accumulator, minimize the internal complexity of the machine and allow for very short instructions. But because the accumulator is only temporary storage, memory traffic is very high. *General purpose register architectures*, which use sets of general purpose registers, are the most widely accepted models for machine architectures today. These register sets are faster than memory, easy for compilers to deal with, and can be used very effectively and efficiently. In addition, hardware prices have decreased significantly, making it possible to add a large number of registers at a minimal cost. If memory access is fast, a stack-based design may be a good idea; if memory is slow, it is often better to use registers. These are the reasons why most computers over the past 10 years have been general-register based. However, because all operands must be named, using registers results in longer instructions, causing longer fetch and decode times. (A very important goal for ISA designers is short instructions.) Designers choosing an

ISA must decide which will work best in a particular environment and examine the tradeoffs carefully.

The general-purpose architecture can be broken into three classifications, depending on where the operands are located. *Memory-memory* architectures may have two or three operands in memory, allowing an instruction to perform an operation without requiring any operand to be in a register. *Register-memory* architectures require a mix, where at least one operand is in a register and one is in memory. *Load-store* architectures require data to be moved into registers before any operations on that data are performed. Intel and Motorola are examples of register-memory architectures; Digital Equipment's VAX architecture allows memory-memory operations; and SPARC, MIPS, ALPHA, and the PowerPC are all load-store machines.

Given that most architectures today are GPR-based, we now examine two major instruction set characteristics that divide general-purpose register architectures. Those two characteristics are the number of operands and how the operands are addressed. In Section 5.2.4 we look at the instruction length and number of operands an instruction can have. (Two or three operands are the most common for GPR architectures, and we compare these to zero and one operand architectures.) We then investigate instruction types. Finally, in Section 5.4 we investigate the various addressing modes available.

5.2.4 Number of Operands and Instruction Length

The traditional method for describing a computer architecture is to specify the maximum number of operands, or addresses, contained in each instruction. This has a direct impact on the length of the instruction itself. MARIE uses a fixed-length instruction with a 4-bit opcode and a 12-bit operand. Instructions on current architectures can be formatted in two ways:

- *Fixed length*—Wastes space but is fast and results in better performance when instruction-level pipelining is used, as we see in Section 5.5.
- *Variable length*—More complex to decode but saves storage space.

Typically, the real-life compromise involves using two to three instruction lengths, which provides bit patterns that are easily distinguishable and simple to decode. The instruction length must also be compared to the word length on the machine. If the instruction length is exactly equal to the word length, the instructions align perfectly when stored in main memory. Instructions always need to be word aligned for addressing reasons. Therefore, instructions that are half, quarter, double, or triple the actual word size can waste space. Variable length instructions are clearly not the same size and need to be word aligned, resulting in loss of space as well.

The most common instruction formats include zero, one, two, or three operands. We saw in Chapter 4 that some instructions for MARIE have no operands, whereas others have one operand. Arithmetic and logic operations typically have two operands, but can be executed with one operand (as we saw in MARIE), if the accumulator is implicit. We can extend this idea to three operands

if we consider the final destination as a third operand. We can also use a stack that allows us to have zero operand instructions. The following are some common instruction formats:

- *OPCODE only* (zero addresses)
- *OPCODE + 1 Address* (usually a memory address)
- *OPCODE + 2 Addresses* (usually registers, or one register and one memory address)
- *OPCODE + 3 Addresses* (usually registers, or combinations of registers and memory)

All architectures have a limit on the maximum number of operands allowed per instruction. For example, in MARIE, the maximum was one, although some instructions had no operands (`Halt` and `Skipcond`). We mentioned that zero-, one-, two-, and three-operand instructions are the most common. One-, two-, and even three-operand instructions are reasonably easy to understand; an entire ISA built on zero-operand instructions can, at first, be somewhat confusing.

Machine instructions that have no operands must use a stack (the last-in, first-out data structure, introduced in Chapter 4 and described in detail in Appendix A, where all insertions and deletions are made from the top) to perform those operations that logically require one or two operands (such as an `Add`). Instead of using general purpose registers, a stack-based architecture stores the operands on the top of the stack, making the top element accessible to the CPU. (Note that one of the most important data structures in machine architectures is the stack. Not only does this structure provide an efficient means of storing intermediate data values during complex calculations, but it also provides an efficient method for passing parameters during procedure calls as well as a means to save local block structure and define the scope of variables and subroutines.)

In architectures based on stacks, most instructions consist of opcodes only; however, there are special instructions (those that add elements to and remove elements from the stack) that have just one operand. Stack architectures need a push instruction and a pop instruction, each of which is allowed one operand. `Push X` places the data value found at memory location *X* onto the stack; `Pop X` removes the top element in the stack and stores it at location *X*. Only certain instructions are allowed to access memory; all others must use the stack for any operands required during execution.

For operations requiring two operands, the top two elements of the stack are used. For example, if we execute an `Add` instruction, the CPU adds the top two elements of the stack, popping them both and then pushing the sum onto the top of the stack. For noncommutative operations such as subtraction, the top stack element is subtracted from the next-to-the-top element, both are popped, and the result is pushed onto the top of the stack.

This stack organization is very effective for evaluating long arithmetic expressions written in *reverse Polish notation* (*RPN*). This representation places the operator after the operands in what is known as *postfix notation* (as compared to *infix notation*, which places the operator between operands, and *prefix notation*, which places the operator before the operands). For example:

X + Y is in infix notation
+ X Y is in prefix notation
X Y + is in postfix notation

All arithmetic expressions can be written using any of these representations. However, postfix representation combined with a stack of registers is the most efficient means to evaluate arithmetic expressions. In fact, some electronic calculators (such as Hewlett-Packard) require the user to enter expressions in postfix notation. With a little practice on these calculators, it is possible to rapidly evaluate long expressions containing many nested parentheses without ever stopping to think about how terms are grouped.

Consider the following expression:

$$(X + Y) \times (W - Z) + 2$$

Written in RPN, this becomes:

$$XY + WZ - \times 2 +$$

Notice that the need for parentheses to preserve precedence is eliminated when using RPN.

To illustrate the concepts of zero, one, two, and three operands, let's write a simple program to evaluate an arithmetic expression, using each of these formats.

EXAMPLE 5.1 Suppose we wish to evaluate the following expression:

$$Z = (X \times Y) + (W \times U)$$

Typically, when three operands are allowed, at least one operand must be a register, and the first operand is normally the destination. Using three-address instructions, the code to evaluate the expression for *Z* is written as follows:

```
Mult    R1, X, Y
Mult    R2, W, U
Add     Z, R2, R1
```

When using two-address instructions, normally one address specifies a register (two-address instructions seldom allow for both operands to be memory addresses). The other operand could be either a register or a memory address. Using two-address instructions, our code becomes:

```
Load    R1, X
Mult    R1, Y
Load    R2, W
Mult    R2, U
Add     R1, R2
Store   Z, R1
```


Note that it is important to know whether the first operand is the source or the destination. In the above instructions, we assume it is the destination. (This tends to be a point of confusion for those programmers who must switch between Intel assembly language and Motorola assembly language—Intel assembly specifies the first operand as the destination, whereas in Motorola assembly, the first operand is the source.)

Using one-address instructions (as in MARIE), we must assume a register (normally the accumulator) is implied as the destination for the result of the instruction. To evaluate *Z*, our code now becomes:

```
Load    X
Mult    Y
Store   Temp
Load    W
Mult    U
Add     Temp
Store   Z
```

Note that as we reduce the number of operands allowed per instruction, the number of instructions required to execute the desired code increases. This is an example of a typical space/time trade-off in architecture design—shorter instructions but longer programs.

What does this program look like on a stack-based machine with zero-address instructions? Stack-based architectures use no operands for instructions such as `Add`, `Subt`, `Mult`, or `Divide`. We need a stack and two operations on that stack: `Pop` and `Push`. Operations that communicate with the stack must have an address field to specify the operand to be popped or pushed onto the stack (all other operations are zero-address). `Push` places the operand on the top of the stack. `Pop` removes the stack top and places it in the operand. This architecture results in the longest program to evaluate our equation. Assuming arithmetic operations use the two operands on the stack top, pop them, and push the result of the operation, our code is as follows:

```
Push    X
Push    Y
Mult
Push    W
Push    U
Mult
Add
Pop     Z
```

The instruction length is certainly affected by the opcode length and by the number of operands allowed in the instruction. If the opcode length is fixed, decoding is much easier. However, to provide for backward compatibility and flexibility, opcodes can have variable length. Variable length opcodes present the same problems as variable versus constant length instructions. A compromise used by many designers is expanding opcodes.

5.2.5 Expanding Opcodes

Expanding opcodes represent a compromise between the need for a rich set of opcodes and the desire to have short opcodes, and thus short instructions. The idea is to make some opcodes short, but have a means to provide longer ones when needed. When the opcode is short, a lot of bits are left to hold operands (which means we could have two or three operands per instruction). When you don't need any space for operands (for an instruction such as `Halt` or because the machine uses a stack), all the bits can be used for the opcode, which allows for many unique instructions. In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands.

Consider a machine with 16-bit instructions and 16 registers. Because we now have a register set instead of one simple accumulator (as in MARIE), we need to use 4 bits to specify a unique register. We could encode 16 instructions, each with 3 register operands (which implies any data to be operated on must first be loaded into a register), or use 4 bits for the opcode and 12 bits for a memory address (as in MARIE, assuming a memory of size 4K). Any memory reference requires 12 bits, leaving only 4 bits for other purposes. However, if all data in memory is first loaded into a register in this register set, the instruction can select that particular data element using only 4 bits (assuming 16 registers). These two choices are illustrated in Figure 5.2.

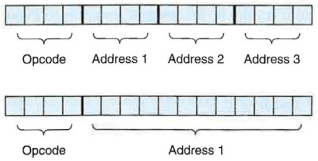


FIGURE 5.2 Two Possibilities for a 16-Bit Instruction Format

Suppose we wish to encode the following instructions:

- 15 instructions with 3 addresses
- 14 instructions with 2 addresses

- 31 instructions with 1 address
- 16 instructions with 0 addresses

Can we encode this instruction set in 16 bits? The answer is yes, as long as we use expanding opcodes. The encoding is as follows:

```
0000 R1   R2   R3
...                    } 15 3-address codes
1110 R1   R2   R3

1111 0000 R1   R2
...                    } 14 2-address codes
1111 1101 R1   R2

1111 1110 0000 R1
...                    } 31 1-address codes
1111 1111 1110 R1

1111 1111 1111 0000
...                    } 16 0-address codes
1111 1111 1111 1111
```

This expanding opcode scheme makes the decoding more complex. Instead of simply looking at a bit pattern and deciding which instruction it is, we need to decode the instruction something like this:

```
if (leftmost four bits != 1111 ) {
   Execute appropriate three-address instruction}
else if (leftmost seven bits != 1111 111 ) {
   Execute appropriate two-address instruction}
else if (leftmost twelve bits != 1111 1111 1111 ) {
   Execute appropriate one-address instruction }
else {
   Execute appropriate zero-address instruction
}
```

At each stage, one spare code is used to indicate that we should now look at more bits. This is another example of the types of trade-offs hardware designers continually face: Here, we trade opcode space for operand space.

5.3 INSTRUCTION TYPES

Most computer instructions operate on data; however, there are some that do not. Computer manufacturers regularly group instructions into the following categories:

- Data movement
- Arithmetic
- Boolean
- Bit manipulation (shift and rotate)
- I/O
- Transfer of control
- Special purpose

Data movement instructions are the most frequently used instructions. Data is moved from memory into registers, from registers to registers, and from registers to memory, and many machines provide different instructions depending on the source and destination. For example, there may be a `MOVER` instruction that always requires two register operands, whereas a `MOVE` instruction allows one register and one memory operand. Some architectures, such as RISC, limit the instructions that can move data to and from memory in an attempt to speed up execution. Many machines have variations of load, store, and move instructions to handle data of different sizes. For example, there may be a `LOADB` instruction for dealing with bytes and a `LOADW` instruction for handling words.

Arithmetic operations include those instructions that use integers and floating point numbers. Many instruction sets provide different arithmetic instructions for various data sizes. As with the data movement instructions, there are sometimes different instructions for providing various combinations of register and memory accesses in different addressing modes.

Boolean logic instructions perform Boolean operations, much in the same way that arithmetic operations work. There are typically instructions for performing AND, NOT, and often OR and XOR operations.

Bit manipulation instructions are used for setting and resetting individual bits (or sometimes groups of bits) within a given data word. These include both arithmetic and logical shift instructions and rotate instructions, both to the left and to the right. Logical shift instructions simply shift bits to either the left or the right by a specified amount, shifting in zeros from the opposite end. Arithmetic shift instructions, commonly used to multiply or divide by 2, do not shift the leftmost bit, because this represents the sign of the number. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, values are shifted left, zeros are shifted in, but the sign bit is never moved. Rotate instructions are simply shift instructions that shift in the bits that are shifted out. For example, on a rotate left 1 bit, the leftmost bit is shifted out and rotated around to become the rightmost bit.

I/O instructions vary greatly from architecture to architecture. The basic schemes for handling I/O are programmed I/O, interrupt-driven I/O, and DMA devices. These are covered in more detail in Chapter 7.

Control instructions include branches, skips, and procedure calls. Branching can be unconditional or conditional. Skip instructions are basically branch instructions with implied addresses. Because no operand is required, skip instructions often use bits of the address field to specify different situations (recall the `Skipcond` instruction used by MARIE). Procedure calls are special branch instructions that automatically save the return address. Different machines use different methods to save this address. Some store the address at a specific location in memory, others store it in a register, while still others push the return address on a stack. We have already seen that stacks can be used for other purposes.

Special purpose instructions include those used for string processing, high-level language support, protection, flag control, and cache management. Most architectures provide instructions for string processing, including string manipulation and searching.

5.4 ADDRESSING

Although addressing is an instruction design issue and is technically part of the instruction format, there are so many issues involved with addressing that it merits its own section. We now present the two most important of these addressing issues: the types of data that can be addressed and the various addressing modes. We cover only the fundamental addressing modes; more specialized modes are built using the basic modes in this section.

5.4.1 Data Types

Before we look at how data is addressed, we will briefly mention the various types of data an instruction can access. There must be hardware support for a particular data type if the instruction is to reference that type. In Chapter 2 we discussed data types, including numbers and characters. Numeric data consists of integers and floating point values. Integers can be signed or unsigned and can be declared in various lengths. For example, in C++ integers can be *short* (16 bits), *int* (the word size of the given architecture), or *long* (32 bits). Floating point numbers have lengths of 32, 64, or 128 bits. It is not uncommon for ISAs to have special instructions to deal with numeric data of varying lengths, as we have seen earlier. For example, there might be a `MOVE` for 16-bit integers and a different `MOVE` for 32-bit integers.

Nonnumeric data types consist of strings, Booleans, and pointers. String instructions typically include operations such as copy, move, search, or modify. Boolean operations include AND, OR, XOR, and NOT. Pointers are actually addresses in memory. Even though they are, in reality, numeric in nature, pointers are treated differently than integers and floating point numbers. MARIE allows