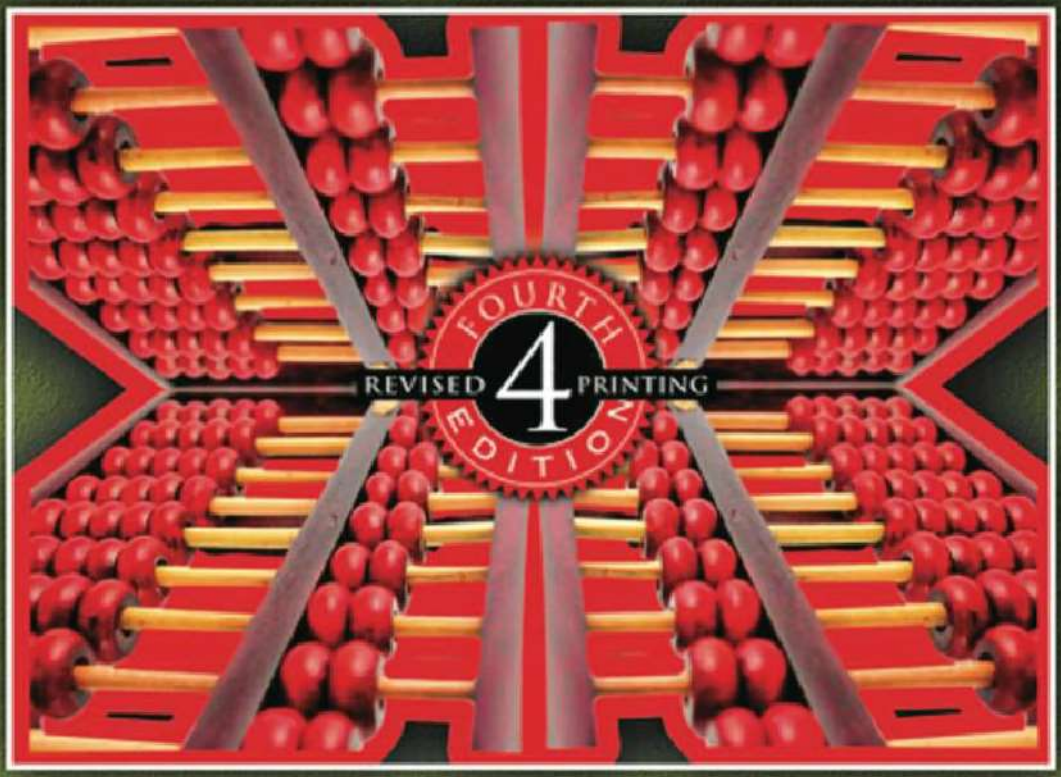
REVISED PRINTING

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE / SOFTWARE INTERFACE



DAVID A. PATTERSON
JOHN L. HENNESSY

MK
MORGAN KAUFMANN

In Praise of *Computer Organization and Design: The Hardware/Software Interface*, Revised Fourth Edition

"Patterson and Hennessy not only improve the pedagogy of the traditional material on pipelined processors and memory hierarchies, but also greatly expand the multiprocessor coverage to include emerging multicore processors and GPUs. The fourth edition of *Computer Organization and Design* sets a new benchmark against which all other architecture books must be compared."

—David A. Wood, *University of Wisconsin-Madison*

"Patterson and Hennessy have greatly improved what was already the gold standard of textbooks. In the rapidly evolving field of computer architecture, they have woven an impressive number of recent case studies and contemporary issues into a framework of time-tested fundamentals."

—Fred Chong, *University of California at Santa Barbara*

"Since the publication of the first edition in 1994, *Computer Organization and Design* has introduced a generation of computer science and engineering students to computer architecture. Now, many of those students have become leaders in the field. In academia, the tradition continues as faculty use the latest edition of the book that inspired them to engage the next generation. With the fourth edition, readers are prepared for the next era of computing."

—David I. August, *Princeton University*

"The new coverage of multiprocessors and parallelism lives up to the standards of this well-written classic. It provides well-motivated, gentle introductions to the new topics, as well as many details and examples drawn from current hardware."

—John Greiner, *Rice University*

"As computer hardware architecture moves from uniprocessor to multicores, the parallel programming environments used to take advantage of these cores will be a defining challenge to the success of these new systems. In the multicore systems, the interface between the hardware and software is of particular importance. This new edition of *Computer Organization and Design* is mandatory for any student who wishes to understand multicore architecture including the interface between programming it and its architecture."

—Jesse Fang, *Director of Programming System Lab at Intel*

"The fourth edition of *Computer Organization and Design* continues to improve the high standards set by the previous editions. The new content, on trends that are reshaping computer systems including multicores, Flash memory, GPUs, etc., makes this edition a must read—even for all of those who grew up on previous editions of the book."

—Parthasarathy Ranganathan, *Principal Research Scientist, HP Labs*

This page intentionally left blank

REVISED FOURTH EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

ACKNOWLEDGMENTS

Figures 1.7, 1.8 Courtesy of Other World Computing (*www.macsales.com*).

Figures 1.9, 1.19, 5.37 Courtesy of AMD.

Figure 1.10 Courtesy of Storage Technology Corp.

Figures 1.10.1, 1.10.2, 4.15.2 Courtesy of the Charles Babbage Institute, University of Minnesota Libraries, Minneapolis.

Figures 1.10.3, 4.15.1, 4.15.3, 5.12.3, 6.14.2 Courtesy of IBM.

Figure 1.10.4 Courtesy of Cray Inc.

Figure 1.10.5 Courtesy of Apple Computer, Inc.

Figure 1.10.6 Courtesy of the Computer History Museum.

Figures 5.12.1, 5.12.2 Courtesy of Museum of Science, Boston.

Figure 5.12.4 Courtesy of MIPS Technologies, Inc.

Figures 6.15, 6.16, 6.17 Courtesy of Sun Microsystems, Inc.

Figure 6.4 © Peg Skorpinski.

Figure 6.14.1 Courtesy of the Computer Museum of America.

Figure 6.14.3 Courtesy of the Commercial Computing Museum.

Figures 7.13.1 Courtesy of NASA Ames Research Center.

REVISED FOURTH EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

David A. Patterson
University of California, Berkeley

John L. Hennessy
Stanford University

With contributions by

Perry Alexander
The University of Kansas

Peter J. Ashenden
Ashenden Designs Pty Ltd

Javier Bruguera
Universidade de Santiago de Compostela

Jichuan Chang
Hewlett-Packard

Matthew Farrens
University of California, Davis

David Kaeli
Northeastern University

Nicole Kaiyan
University of Adelaide

David Kirk
NVIDIA

James R. Larus
Microsoft Research

Jacob Leverich
Hewlett-Packard

Kevin Lim
Hewlett-Packard

John Nickolls
NVIDIA

John Oliver
Cal Poly, San Luis Obispo

Milos Prvulovic
Georgia Tech

Partha Ranganathan
Hewlett-Packard

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN

Acquiring Editor: Todd Green
Development Editor: Nate McFadden
Project Manager: Jessica Vaughan
Designer: Eric DeCicco

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

© 2012 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: *www.elsevier.com/permissions*.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Patterson, David A.
Computer organization and design: the hardware/software interface / David A. Patterson, John L. Hennessy. — 4th ed.
p. cm. — (The Morgan Kaufmann series in computer architecture and design)
Rev. ed. of: Computer organization and design / John L. Hennessy, David A. Patterson. 1998.
Summary: "Presents the fundamentals of hardware technologies, assembly language, computer arithmetic, pipelining, memory hierarchies and I/O"— Provided by publisher.
ISBN 978-0-12-374750-1 (pbk.)
1. Computer organization. 2. Computer engineering. 3. Computer interfaces. I. Hennessy, John L. II. Hennessy, John L. Computer organization and design. III. Title.
QA76.9.C643H46 2011
004.2´2—dc23

2011029199

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-374750-1

For information on all MK publications visit our website at *www.mkp.com*

Printed in the United States of America

12 13 14 15 16 10 9 8 7 6 5 4 3 2 1

Working together to grow libraries in developing countries
www.elsevier.com | www.bookaid.org | www.sabre.org
ELSEVIER BOOK AID International Sabre Foundation

To Linda,
who has been, is, and always will be the love of my life

This page intentionally left blank

Contents

Preface xv

CHAPTERS

1 Computer Abstractions and Technology 2

1.1 Introduction 3
1.2 Below Your Program 10
1.3 Under the Covers 13
1.4 Performance 26
1.5 The Power Wall 39
1.6 The Sea Change: The Switch from Uniprocessors to Multiprocessors 41
1.7 Real Stuff: Manufacturing and Benchmarking the AMD Opteron X4 44
1.8 Fallacies and Pitfalls 51
1.9 Concluding Remarks 54
1.10 Historical Perspective and Further Reading 55
1.11 Exercises 56

2 Instructions: Language of the Computer 74

2.1 Introduction 76
2.2 Operations of the Computer Hardware 77
2.3 Operands of the Computer Hardware 80
2.4 Signed and Unsigned Numbers 87
2.5 Representing Instructions in the Computer 94
2.6 Logical Operations 102
2.7 Instructions for Making Decisions 105
2.8 Supporting Procedures in Computer Hardware 112
2.9 Communicating with People 122
2.10 MIPS Addressing for 32-Bit Immediates and Addresses 128
2.11 Parallelism and Instructions: Synchronization 137
2.12 Translating and Starting a Program 139
2.13 A C Sort Example to Put It All Together 149

2.14 Arrays versus Pointers 157
2.15 Advanced Material: Compiling C and Interpreting Java 161
2.16 Real Stuff: ARM Instructions 161
2.17 Real Stuff: x86 Instructions 165
2.18 Fallacies and Pitfalls 174
2.19 Concluding Remarks 176
2.20 Historical Perspective and Further Reading 179
2.21 Exercises 179

3 Arithmetic for Computers 222

3.1 Introduction 224
3.2 Addition and Subtraction 224
3.3 Multiplication 230
3.4 Division 236
3.5 Floating Point 242
3.6 Parallelism and Computer Arithmetic: Associativity 270
3.7 Real Stuff: Floating Point in the x86 272
3.8 Fallacies and Pitfalls 275
3.9 Concluding Remarks 280
3.10 Historical Perspective and Further Reading 283
3.11 Exercises 283

4 The Processor 298

4.1 Introduction 300
4.2 Logic Design Conventions 303
4.3 Building a Datapath 307
4.4 A Simple Implementation Scheme 316
4.5 An Overview of Pipelining 330
4.6 Pipelined Datapath and Control 344
4.7 Data Hazards: Forwarding versus Stalling 363
4.8 Control Hazards 375
4.9 Exceptions 384
4.10 Parallelism and Advanced Instruction-Level Parallelism 391
4.11 Real Stuff: the AMD Opteron X4 (Barcelona) Pipeline 404
4.12 Advanced Topic: an Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations 406
4.13 Fallacies and Pitfalls 407
4.14 Concluding Remarks 408
4.15 Historical Perspective and Further Reading 409
4.16 Exercises 409

5 Large and Fast: Exploiting Memory Hierarchy 450

5.1 Introduction 452
5.2 The Basics of Caches 457
5.3 Measuring and Improving Cache Performance 475
5.4 Virtual Memory 492
5.5 A Common Framework for Memory Hierarchies 518
5.6 Virtual Machines 525
5.7 Using a Finite-State Machine to Control a Simple Cache 529
5.8 Parallelism and Memory Hierarchies: Cache Coherence 534
5.9 Advanced Material: Implementing Cache Controllers 538
5.10 Real Stuff: the AMD Opteron X4 (Barcelona) and Intel Nehalem Memory Hierarchies 539
5.11 Fallacies and Pitfalls 543
5.12 Concluding Remarks 547
5.13 Historical Perspective and Further Reading 548
5.14 Exercises 548

6 Storage and Other I/O Topics 568

6.1 Introduction 570
6.2 Dependability, Reliability, and Availability 573
6.3 Disk Storage 575
6.4 Flash Storage 580
6.5 Connecting Processors, Memory, and I/O Devices 582
6.6 Interfacing I/O Devices to the Processor, Memory, and Operating System 586
6.7 I/O Performance Measures: Examples from Disk and File Systems 596
6.8 Designing an I/O System 598
6.9 Parallelism and I/O: Redundant Arrays of Inexpensive Disks 599
6.10 Real Stuff: Sun Fire x4150 Server 606
6.11 Advanced Topics: Networks 612
6.12 Fallacies and Pitfalls 613
6.13 Concluding Remarks 617
6.14 Historical Perspective and Further Reading 618
6.15 Exercises 619

7 Multicores, Multiprocessors, and Clusters 630

7.1 Introduction 632
7.2 The Difficulty of Creating Parallel Processing Programs 634
7.3 Shared Memory Multiprocessors 638

7.4 Clusters and Other Message-Passing Multiprocessors 641
7.5 Hardware Multithreading 645
7.6 SISD, MIMD, SIMD, SPMD, and Vector 648
7.7 Introduction to Graphics Processing Units 654
7.8 Introduction to Multiprocessor Network Topologies 660
7.9 Multiprocessor Benchmarks 664
7.10 Roofline: A Simple Performance Model 667
7.11 Real Stuff: Benchmarking Four Multicores Using the Roofline Model 675
7.12 Fallacies and Pitfalls 684
7.13 Concluding Remarks 686
7.14 Historical Perspective and Further Reading 688
7.15 Exercises 688

APPENDICES

A Graphics and Computing GPUs A-2

A.1 Introduction A-3
A.2 GPU System Architectures A-7
A.3 Programming GPUs A-12
A.4 Multithreaded Multiprocessor Architecture A-25
A.5 Parallel Memory System A-36
A.6 Floating Point Arithmetic A-41
A.7 Real Stuff: The NVIDIA GeForce 8800 A-46
A.8 Real Stuff: Mapping Applications to GPUs A-55
A.9 Fallacies and Pitfalls A-72
A.10 Concluding Remarks A-76
A.11 Historical Perspective and Further Reading A-77

B Assemblers, Linkers, and the SPIM Simulator B-2

B.1 Introduction B-3
B.2 Assemblers B-10
B.3 Linkers B-18
B.4 Loading B-19
B.5 Memory Usage B-20
B.6 Procedure Call Convention B-22
B.7 Exceptions and Interrupts B-33
B.8 Input and Output B-38
B.9 SPIM B-40

B.10 MIPS R2000 Assembly Language B-45
B.11 Concluding Remarks B-81
B.12 Exercises B-82
Index I-1

CD-ROM CONTENT

C The Basics of Logic Design C-2

C.1 Introduction C-3
C.2 Gates, Truth Tables, and Logic Equations C-4
C.3 Combinational Logic C-9
C.4 Using a Hardware Description Language C-20
C.5 Constructing a Basic Arithmetic Logic Unit C-26
C.6 Faster Addition: Carry Lookahead C-38
C.7 Clocks C-48
C.8 Memory Elements: Flip-Flops, Latches, and Registers C-50
C.9 Memory Elements: SRAMs and DRAMs C-58
C.10 Finite-State Machines C-67
C.11 Timing Methodologies C-72
C.12 Field Programmable Devices C-78
C.13 Concluding Remarks C-79
C.14 Exercises C-80

D Mapping Control to Hardware D-2

D.1 Introduction D-3
D.2 Implementing Combinational Control Units D-4
D.3 Implementing Finite-State Machine Control D-8
D.4 Implementing the Next-State Function with a Sequencer D-22
D.5 Translating a Microprogram to Hardware D-28
D.6 Concluding Remarks D-32
D.7 Exercises D-33

E A Survey of RISC Architectures for Desktop, Server, and Embedded Computers E-2

E.1 Introduction E-3
E.2 Addressing Modes and Instruction Formats E-5
E.3 Instructions: The MIPS Core Subset E-9

E.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs E-16
E.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs E-19
E.6 Instructions: Common Extensions to MIPS Core E-20
E.7 Instructions Unique to MIPS-64 E-25
E.8 Instructions Unique to Alpha E-27
E.9 Instructions Unique to SPARC v.9 E-29
E.10 Instructions Unique to PowerPC E-32
E.11 Instructions Unique to PA-RISC 2.0 E-34
E.12 Instructions Unique to ARM E-36
E.13 Instructions Unique to Thumb E-38
E.14 Instructions Unique to SuperH E-39
E.15 Instructions Unique to M32R E-40
E.16 Instructions Unique to MIPS-16 E-40
E.17 Concluding Remarks E-43

Glossary G-1

Further Reading FR-1

For the convenience of readers who have purchased an ebook edition, all CD-ROM content is available as a download from the book's companion page. Visit ***http://www.elsevierdirect.com/companion.jsp?ISBN=9780123747501*** *to download your CD-ROM files.*

Preface

The most beautiful thing we can experience is the mysterious.
It is the source of all true art and science.

Albert Einstein, *What I Believe*, 1930

About This Book

We believe that learning in computer science and engineering should reflect the current state of the field, as well as introduce the principles that are shaping computing. We also feel that readers in every specialty of computing need to appreciate the organizational paradigms that determine the capabilities, performance, and, ultimately, the success of computer systems.

Modern computer technology requires professionals of every computing specialty to understand both hardware and software. The interaction between hardware and software at a variety of levels also offers a framework for understanding the fundamentals of computing. Whether your primary interest is hardware or software, computer science or electrical engineering, the central ideas in computer organization and design are the same. Thus, our emphasis in this book is to show the relationship between hardware and software and to focus on the concepts that are the basis for current computers.

The recent switch from uniprocessor to multicore microprocessors confirmed the soundness of this perspective, given since the first edition. While programmers could ignore the advice and rely on computer architects, compiler writers, and silicon engineers to make their programs run faster without change, that era is over. For programs to run faster, they must become parallel. While the goal of many researchers is to make it possible for programmers to be unaware of the underlying parallel nature of the hardware they are programming, it will take many years to realize this vision. Our view is that for at least the next decade, most programmers are going to have to understand the hardware/software interface if they want programs to run efficiently on parallel computers.

The audience for this book includes those with little experience in assembly language or logic design who need to understand basic computer organization as well as readers with backgrounds in assembly language and/or logic design who want to learn how to design a computer or understand how a system works and why it performs as it does.

About the Other Book

Some readers may be familiar with *Computer Architecture: A Quantitative Approach*, popularly known as Hennessy and Patterson. (This book in turn is often called Patterson and Hennessy.) Our motivation in writing the earlier book was to describe the principles of computer architecture using solid engineering fundamentals and quantitative cost/performance tradeoffs. We used an approach that combined examples and measurements, based on commercial systems, to create realistic design experiences. Our goal was to demonstrate that computer architecture could be learned using quantitative methodologies instead of a descriptive approach. It was intended for the serious computing professional who wanted a detailed understanding of computers.

A majority of the readers for this book do not plan to become computer architects. The performance and energy efficiency of future software systems will be dramatically affected, however, by how well software designers understand the basic hardware techniques at work in a system. Thus, compiler writers, operating system designers, database programmers, and most other software engineers need a firm grounding in the principles presented in this book. Similarly, hardware designers must understand clearly the effects of their work on software applications.

Thus, we knew that this book had to be much more than a subset of the material in *Computer Architecture*, and the material was extensively revised to match the different audience. We were so happy with the result that the subsequent editions of *Computer Architecture* were revised to remove most of the introductory material; hence, there is much less overlap today than with the first editions of both books.

Changes for the Fourth Edition

We had five major goals for the fourth edition of *Computer Organization and Design:* given the multicore revolution in microprocessors, highlight parallel hardware and software topics throughout the book; streamline the existing material to make room for topics on parallelism; enhance pedagogy in general; update the technical content to reflect changes in the industry since the publication of the third edition in 2004; and restore the usefulness of exercises in this Internet age.

Before discussing the goals in detail, let's look at the table on the next page. It shows the hardware and software paths through the material. Chapters 1, 4, 5, and 7 are found on both paths, no matter what the experience or the focus. Chapter 1 is a new introduction that includes a discussion on the importance of power and how it motivates the switch from single core to multicore microprocessors. It also includes performance and benchmarking material that was a separate chapter in the third edition. Chapter 2 is likely to be review material for the hardware-oriented, but it is essential reading for the software-oriented, especially for those readers interested in learning more about compilers and object-oriented programming

Chapter or appendix	*Sections*	*Software focus*	*Hardware focus*
1. Computer Abstractions and Technology	1.1 to 1.9	Read carefully	Read carefully
	1.10 (History)	Read if have time	Read if have time
2. Instructions: Language of the Computer	2.1 to 2.14	Read carefully	Read carefully
	2.15 (Compilers & Java)	Read if have time	
	2.16 to 2.19	Read carefully	Review or read
	2.20 (History)	Read if have time	Read if have time
E. RISC Instruction-Set Architectures	E.1 to E.19	Read if have time	
3. Arithmetic for Computers	3.1 to 3.9	Read carefully	Review or read
	3.10 (History)	Read if have time	Read if have time
C. The Basics of Logic Design	C.1 to C.13		Review or read
4. The Processor	4.1 (Overview)	Read carefully	Read carefully
	4.2 (Logic Conventions)		Read carefully
	4.3 to 4.4 (Simple Implementation)	Review or read	Read carefully
	4.5 (Pipelining Overview)	Read carefully	Read carefully
	4.6 (Pipelined Datapath)	Review or read	Read carefully
	4.7 to 4.9 (Hazards, Exceptions)		Read carefully
	4.10 to 4.11 (Parallel, Real Stuff)	Read carefully	Read carefully
	4.12 (Verilog Pipeline Control)		Read if have time
	4.13 to 4.14 (Fallacies)	Read carefully	Read carefully
	4.15 (History)	Read if have time	Read if have time
D. Mapping Control to Hardware	D.1 to D.6		Read if have time
5. Large and Fast: Exploiting Memory Hierarchy	5.1 to 5.8	Read carefully	Read carefully
	5.9 (Verilog Cache Controller)		Read if have time
	5.10 to 5.12	Read carefully	Read carefully
	5.13 (History)	Read if have time	Read if have time
6. Storage and Other I/O Topics	6.1 to 6.10	Read carefully	Read if have time
	6.11 (Networks)	Read if have time	Read if have time
	6.12 to 6.13	Read carefully	Read if have time
	6.14 (History)	Read if have time	Read if have time
7. Multicores, Multiprocessors, and Clusters	7.1 to 7.13	Read carefully	Read carefully
	7.14 (History)	Read if have time	Read if have time
A. Graphics Processor Units	A.1 to A.12	Read if have time	Read if have time
B. Assemblers, Linkers, and the SPIM Simulator	B.1 to B.12	Read if have time	Read if have time

Read carefully — Read if have time — Reference
Review or read — Read for culture

languages. It includes material from Chapter 3 in the third edition so that the complete MIPS architecture is now in a single chapter, minus the floating-point instructions. Chapter 3 is for readers interested in constructing a datapath or in learning more about floating-point arithmetic. Some will skip Chapter 3, either because they don't need it or because it is a review. Chapter 4 combines two chapters from the third edition to explain pipelined processors. Sections 4.1, 4.5, and 4.10 give overviews for those with a software focus. Those with a hardware focus, however, will find that this chapter presents core material; they may also, depending on their background, want to read Appendix C on logic design first. Chapter 6 on storage is critical to readers with a software focus, and should be read by others if time permits. The last chapter on multicores, multiprocessors, and clusters is mostly new content and should be read by everyone.

The first goal was to make parallelism a first class citizen in this edition, as it was a separate chapter on the CD in the last edition. The most obvious example is Chapter 7. In particular, this chapter introduces the Roofline performance model, and shows its value by evaluating four recent multicore architectures on two kernels. This model could prove to be as insightful for multicore microprocessors as the 3Cs model is for caches.

Given the importance of parallelism, it wasn't wise to wait until the last chapter to talk about, so there is a section on parallelism in each of the preceding six chapters:

- *Chapter 1: Parallelism and Power.* It shows how power limits have forced the industry to switch to parallelism, and why parallelism helps.
- *Chapter 2: Parallelism and Instructions: Synchronization.* This section discusses locks for shared variables, specifically the MIPS instructions Load Linked and Store Conditional.
- *Chapter 3: Parallelism and Computer Arithmetic: Floating-Point Associativity.* This section discusses the challenges of numerical precision and floating-point calculations.
- *Chapter 4: Parallelism and Advanced Instruction-Level Parallelism.* It covers advanced ILP—superscalar, speculation, VLIW, loop-unrolling, and OOO—as well as the relationship between pipeline depth and power consumption.
- *Chapter 5: Parallelism and Memory Hierarchies: Cache Coherence.* It introduces coherency, consistency, and snooping cache protocols.
- *Chapter 6: Parallelism and I/O: Redundant Arrays of Inexpensive Disks.* It describes RAID as a parallel I/O system as well as a highly available ICO system.

Chapter 7 concludes with reasons for optimism why this foray into parallelism should be more successful than those of the past.

I am particularly excited about the addition of an appendix on Graphical Processing Units written by NVIDIA's chief scientist, David Kirk, and chief architect, John Nickolls. Appendix A is the first in-depth description of GPUs, which is a new and interesting thrust in computer architecture. The appendix builds upon the parallel themes of this edition to present a style of computing that allows the programmer to think MIMD yet the hardware tries to execute in SIMD-style whenever possible. As GPUs are both inexpensive and widely available—they are even found in many laptops—and their programming environments are freely available, they provide a parallel hardware platform that many could experiment with.

The second goal was to streamline the book to make room for new material in parallelism. The first step was simply going through all the paragraphs accumulated over three editions with a fine-toothed comb to see if they were still necessary. The coarse-grained changes were the merging of chapters and dropping of topics. Mark Hill suggested dropping the multicycle processor implementation and instead adding a multicycle cache controller to the memory hierarchy chapter. This allowed the processor to be presented in a single chapter instead of two, enhancing the processor material by omission. The performance material from a separate chapter in the third edition is now blended into the first chapter.

The third goal was to improve the pedagogy of the book. Chapter 1 is now meatier, including performance, integrated circuits, and power, and it sets the stage for the rest of the book. Chapters 2 and 3 were originally written in an evolutionary style, starting with a "single celled" architecture and ending up with the full MIPS architecture by the end of Chapter 3. This leisurely style is not a good match to the modern reader. This edition merges all of the instruction set material for the integer instructions into Chapter 2—making Chapter 3 optional for many readers—and each section now stands on its own. The reader no longer needs to read all of the preceding sections. Hence, Chapter 2 is now even better as a reference than it was in prior editions. Chapter 4 works better since the processor is now a single chapter, as the multicycle implementation is a distraction today. Chapter 5 has a new section on building cache controllers, along with a new CD section containing the Verilog code for that cache.

The accompanying CD-ROM introduced in the third edition allowed us to reduce the cost of the book by saving pages as well as to go into greater depth on topics that were of interest to some but not all readers. Alas, in our enthusiasm to save pages, readers sometimes found themselves going back and forth between the CD and book more often than they liked. This should not be the case in this edition. Each chapter now has the Historical Perspectives section on the CD and four chapters also have one advanced material section on the CD. Additionally, all

exercises are in the printed book, so flipping between book and CD should be rare in this edition.

For those of you who wonder why we include a CD-ROM with the book, the answer is simple: the CD contains content that we feel should be easily and immediately accessible to the reader no matter where they are. If you are interested in the advanced content, or would like to review a VHDL tutorial (for example), it is on the CD, ready for you to use. The CD-ROM also includes a feature that should greatly enhance your study of the material: a search engine is included that allows you to search for any string of text, in the printed book or on the CD itself. If you are hunting for content that may not be included in the book's printed index, you can simply enter the text you're searching for and the page number it appears on will be displayed in the search results. This is a very useful feature that we hope you make frequent use of as you read and review the book.

This is a fast-moving field, and as is always the case for our new editions, an important goal is to update the technical content. The AMD Opteron X4 model 2356 (code named "Barcelona") serves as a running example throughout the book, and is found in Chapters 1, 4, 5, and 7. Chapters 1 and 6 add results from the new power benchmark from SPEC. Chapter 2 adds a section on the ARM architecture, which is currently the world's most popular 32-bit ISA. Chapter 5 adds a new section on Virtual Machines, which are resurging in importance. Chapter 5 has detailed cache performance measurements on the Opteron X4 multicore and a few details on its rival, the Intel Nehalem, which will not be announced until after this edition is published. Chapter 6 describes Flash Memory for the first time as well as a remarkably compact server from Sun, which crams 8 cores, 16 DIMMs, and 8 disks into a single 1U bit. It also includes the recent results on long-term disk failures. Chapter 7 covers a wealth of topics regarding parallelism—including multithreading, SIMD, vector, GPUs, performance models, benchmarks, multiprocessor networks—and describes three multicores plus the Opteron X4: Intel Xeon model e5345 (Clovertown), IBM Cell model QS20, and the Sun Microsystems T2 model 5120 (Niagara 2).

The final goal was to try to make the exercises useful to instructors in this Internet age, for homework assignments have long been an important way to learn material. Alas, answers are posted today almost as soon as the book appears. We have a two-part approach. First, expert contributors have worked to develop entirely new exercises for each chapter in the book. Second, most exercises have a qualitative description supported by a table that provides several alternative quantitative parameters needed to answer this question. The sheer number plus flexibility in terms of how the instructor can choose to assign variations of exercises will make it hard for students to find the matching solutions online. Instructors will also be able to change these quantitative parameters as they wish, again frustrating those students who have come to rely on the Internet to provide solutions for a static and unchanging set of exercises. We feel this new approach is a valuable new addition to the book—please let us know how well it works for you, either as a student or instructor!

We have preserved useful book elements from prior editions. To make the book work better as a reference, we still place definitions of new terms in the margins at their first occurrence. The book element called "Understanding Program Performance" sections helps readers understand the performance of their programs and how to improve it, just as the "Hardware/Software Interface" book element helped readers understand the tradeoffs at this interface. "The Big Picture" section remains so that the reader sees the forest even despite all the trees. "Check Yourself" sections help readers to confirm their comprehension of the material on the first time through with answers provided at the end of each chapter. This edition also includes the green MIPS reference card, which was inspired by the "Green Card" of the IBM System/360. The removable card has been updated and should be a handy reference when writing MIPS assembly language programs.

Instructor Support

We have collected a great deal of material to help instructors teach courses using this book. Solutions to exercises, chapter quizzes, figures from the book, lecture notes, lecture slides, and other materials are available to adopters from the publisher. Check the publisher's Web site for more information:

textbooks.elsevier.com/9780123744937

Concluding Remarks

If you read the following acknowledgments section, you will see that we went to great lengths to correct mistakes. Since a book goes through many printings, we have the opportunity to make even more corrections. If you uncover any remaining, resilient bugs, please contact the publisher by electronic mail at *cod4bugs@mkp.com* or by low-tech mail using the address found on the copyright page.

This edition marks a break in the long-standing collaboration between Hennessy and Patterson, which started in 1989. The demands of running one of the world's great universities meant that President Hennessy could no longer make the substantial commitment to create a new edition. The remaining author felt like a juggler who had always performed with a partner who suddenly is thrust on the stage as a solo act. Hence, the people in the acknowledgments and Berkeley colleagues played an even larger role in shaping the contents of this book. Nevertheless, this time around there is only one author to blame for the new material in what you are about to read.

Acknowledgments for the Fourth Edition

I'd like to thank **David Kirk, John Nickolls,** and their colleagues at NVIDIA (Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman, and Vasily Volkov) for writing

the first in-depth appendix on GPUs. I'd like to express again my appreciation to **Jim Larus** of Microsoft Research for his willingness in contributing his expertise on assembly language programming, as well as for welcoming readers of this book to use the simulator he developed and maintains.

I am also very grateful for the contributions of the many experts who developed the new exercises for this new edition. Writing good exercises is not an easy task, and each contributor worked long and hard to develop problems that are both challenging and engaging:

- *Chapter 1:* **Javier Bruguera** (Universidade de Santiago de Compostela)
- *Chapter 2:* **John Oliver** (Cal Poly, San Luis Obispo), with contributions from **Nicole Kaiyan** (University of Adelaide) and **Milos Prvulovic** (Georgia Tech)
- *Chapter 3:* **Matthew Farrens** (University of California, Davis)
- *Chapter 4:* **Milos Prvulovic** (Georgia Tech)
- *Chapter 5:* **Jichuan Chang, Jacob Leverich, Kevin Lim,** and **Partha Ranganathan** (all from Hewlett-Packard), with contributions from Nicole Kaiyan (University of Adelaide)
- *Chapter 6:* **Perry Alexander** (The University of Kansas)
- *Chapter 7:* **David Kaeli** (Northeastern University)

Peter Ashenden took on the Herculean task of editing and evaluating *all* of the new exercises. Moreover, he even added the substantial burden of developing the companion CD and new lecture slides.

Thanks to **David August** and **Prakash Prabhu** of Princeton University for their work on the chapter quizzes that are available for instructors on the publisher's Web site.

I relied on my Silicon Valley colleagues for much of the technical material that this book relies upon:

- **AMD**—for the details and measurements of the Opteron X4 (Barcelona): **William Brantley, Vasileios Liaskovitis, Chuck Moore,** and **Brian Waldecker.**
- **Intel**—for the prereleased information on the Intel Nehalem: **Faye Briggs**.
- **Micron**—for background on Flash Memory in Chapter 6: **Dean Klein.**
- **Sun Microsystems**—for the measurements of the instruction mixes for the SPEC CPU2006 benchmarks in Chapter 2 and details and measurements of the Sun Server x4150 in Chapter 6: **Yan Fisher, John Fowler, Darryl Gove, Paul Joyce, Shenik Mehta, Pierre Reynes, Dimitry Stuve, Durgam Vahia,** and **David Weaver.**
- **U.C. Berkeley**—**Krste Asanovic** (who supplied the idea for software concurrency versus hardware parallelism in Chapter 7), **James Demmel**

and **Velvel Kahan** (who commented on parallelism and floating-point calculations), **Zhangxi Tan** (who designed the cache controller and wrote the Verilog for it in Chapter 5), **Sam Williams** (who supplied the roofline model and the multicore measurements in Chapter 7), and the rest of my colleagues in the **Par Lab** who gave extensive suggestions and feedback on parallelism topics found throughout the book.

I am grateful to the many instructors who answered the publisher's surveys, reviewed our proposals, and attended focus groups to analyze and respond to our plans for this edition. They include the following individuals: *Focus Group:* Mark Hill (University of Wisconsin, Madison), E.J. Kim (Texas A&M University), Jihong Kim (Seoul National University), Lu Peng (Louisiana State University), Dean Tullsen (UC San Diego), Ken Vollmar (Missouri State University), David Wood (University of Wisconsin, Madison), Ki Hwan Yum (University of Texas, San Antonio); *Surveys and Reviews:* Mahmoud Abou-Nasr (Wayne State University), Perry Alexander (The University of Kansas), Hakan Aydin (George Mason University), Hussein Badr (State University of New York at Stony Brook), Mac Baker (Virginia Military Institute), Ron Barnes (George Mason University), Douglas Blough (Georgia Institute of Technology), Kevin Bolding (Seattle Pacific University), Miodrag Bolic (University of Ottawa), John Bonomo (Westminster College), Jeff Braun (Montana Tech), Tom Briggs (Shippensburg University), Scott Burgess (Humboldt State University), Fazli Can (Bilkent University), Warren R. Carithers (Rochester Institute of Technology), Bruce Carlton (Mesa Community College), Nicholas Carter (University of Illinois at Urbana-Champaign), Anthony Cocchi (The City University of New York), Don Cooley (Utah State University), Robert D. Cupper (Allegheny College), Edward W. Davis (North Carolina State University), Nathaniel J. Davis (Air Force Institute of Technology), Molisa Derk (Oklahoma City University), Derek Eager (University of Saskatchewan), Ernest Ferguson (Northwest Missouri State University), Rhonda Kay Gaede (The University of Alabama), Etienne M. Gagnon (UQAM), Costa Gerousis (Christopher Newport University), Paul Gillard (Memorial University of Newfoundland), Michael Goldweber (Xavier University), Georgia Grant (College of San Mateo), Merrill Hall (The Master's College), Tyson Hall (Southern Adventist University), Ed Harcourt (Lawrence University), Justin E. Harlow (University of South Florida), Paul F. Hemler (Hampden-Sydney College), Martin Herbordt (Boston University), Steve J. Hodges (Cabrillo College), Kenneth Hopkinson (Cornell University), Dalton Hunkins (St. Bonaventure University), Baback Izadi (State University of New York—New Paltz), Reza Jafari, Robert W. Johnson (Colorado Technical University), Bharat Joshi (University of North Carolina, Charlotte), Nagarajan Kandasamy (Drexel University), Rajiv Kapadia, Ryan Kastner (University of California, Santa Barbara), Jim Kirk (Union University), Geoffrey S. Knauth (Lycoming College), Manish M. Kochhal (Wayne State), Suzan Koknar-Tezel (Saint Joseph's University), Angkul Kongmunvattana (Columbus State University), April Kontostathis (Ursinus College), Christos Kozyrakis (Stanford University), Danny Krizanc (Wesleyan University), Ashok Kumar, S. Kumar (The University of Texas), Robert N. Lea (University of Houston),

Baoxin Li (Arizona State University), Li Liao (University of Delaware), Gary Livingston (University of Massachusetts), Michael Lyle, Douglas W. Lynn (Oregon Institute of Technology), Yashwant K Malaiya (Colorado State University), Bill Mark (University of Texas at Austin), Ananda Mondal (Claflin University), Alvin Moser (Seattle University), Walid Najjar (University of California, Riverside), Danial J. Neebel (Loras College), John Nestor (Lafayette College), Joe Oldham (Centre College), Timour Paltashev, James Parkerson (University of Arkansas), Shaunak Pawagi (SUNY at Stony Brook), Steve Pearce, Ted Pedersen (University of Minnesota), Gregory D Peterson (The University of Tennessee), Dejan Raskovic (University of Alaska, Fairbanks) Brad Richards (University of Puget Sound), Roman Rozanov, Louis Rubinfield (Villanova University), Md Abdus Salam (Southern University), Augustine Samba (Kent State University), Robert Schaefer (Daniel Webster College), Carolyn J. C. Schauble (Colorado State University), Keith Schubert (CSU San Bernardino), William L. Schultz, Kelly Shaw (University of Richmond), Shahram Shirani (McMaster University), Scott Sigman (Drury University), Bruce Smith, David Smith, Jeff W. Smith (University of Georgia, Athens), Philip Snyder (Johns Hopkins University), Alex Sprintson (Texas A&M), Timothy D. Stanley (Brigham Young University), Dean Stevens (Morningside College), Nozar Tabrizi (Kettering University), Yuval Tamir (UCLA), Alexander Taubin (Boston University), Will Thacker (Winthrop University), Mithuna Thottethodi (Purdue University), Manghui Tu (Southern Utah University), Rama Viswanathan (Beloit College), Guoping Wang (Indiana-Purdue University), Patricia Wenner (Bucknell University), Kent Wilken (University of California, Davis), David Wolfe (Gustavus Adolphus College), David Wood (University of Wisconsin, Madison), Mohamed Zahran (City College of New York), Gerald D. Zarnett (Ryerson University), Nian Zhang (South Dakota School of Mines & Technology), Jiling Zhong (Troy University), Huiyang Zhou (The University of Central Florida), Weiyu Zhu (Illinois Wesleyan University).

I would especially like to thank the Berkeley people who gave key feedback for Chapter 7 and Appendix A, which were the most challenging pieces to write for this edition: **Krste Asanovic, Christopher Batten, Rastilav Bodik, Bryan Catanzaro, Jike Chong, Kaushik Data, Greg Giebling, Anik Jain, Jae Lee, Vasily Volkov**, and **Samuel Williams.**

A special thanks also goes to **Mark Smotherman** for making multiple passes to find technical and writing glitches that significantly improved the quality of this edition. He played an even more important role this time given that this edition was done as a solo act.

We wish to thank the extended Morgan Kaufmann family for agreeing to publish this book again under the able leadership of **Denise Penrose. Nathaniel McFadden** was the developmental editor for this edition and worked with me weekly on the contents of the book. **Kimberlee Honjo** coordinated the surveying of users and their responses.

Dawnmarie Simpson managed the book production process. We thank also the many freelance vendors who contributed to this volume, especially Alan Rose of Multiscience Press and diacriTech, our compositor.

The contributions of the nearly 200 people we mentioned here have helped make this fourth edition what I hope will be our best book yet. Enjoy!

David A. Patterson

1

Computer Abstractions and Technology

Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred North Whitehead
An Introduction to Mathematics, 1911

1.1 Introduction 3
1.2 Below Your Program 10
1.3 Under the Covers 13
1.4 Performance 26
1.5 The Power Wall 39
1.6 The Sea Change: The Switch from Uniprocessors to Multiprocessors 41

Computer Organization and Design. DOI: 10.1016/B978-0-12-374750-1.00001-3
© 2012 Elsevier, Inc. All rights reserved.

1.7 Real Stuff: Manufacturing and Benchmarking the AMD Opteron X4 44
1.8 Fallacies and Pitfalls 51
1.9 Concluding Remarks 54
1.10 Historical Perspective and Further Reading 55
1.11 Exercises 56

1.1 Introduction

Welcome to this book! We're delighted to have this opportunity to convey the excitement of the world of computer systems. This is not a dry and dreary field, where progress is glacial and where new ideas atrophy from neglect. No! Computers are the product of the incredibly vibrant information technology industry, all aspects of which are responsible for almost 10% of the gross national product of the United States, and whose economy has become dependent in part on the rapid improvements in information technology promised by Moore's law. This unusual industry embraces innovation at a breathtaking rate. In the last 25 years, there have been a number of new computers whose introduction appeared to revolutionize the computing industry; these revolutions were cut short only because someone else built an even better computer.

This race to innovate has led to unprecedented progress since the inception of electronic computing in the late 1940s. Had the transportation industry kept pace with the computer industry, for example, today we could travel from New York to London in about a second for roughly a few cents. Take just a moment to contemplate how such an improvement would change society—living in Tahiti while working in San Francisco, going to Moscow for an evening at the Bolshoi Ballet—and you can appreciate the implications of such a change.

Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and the industrial revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected our everyday lives profoundly and changed the ways in which the search for new knowledge is carried out. There is now a new vein of scientific investigation, with computational scientists joining theoretical and experimental scientists in the exploration of new frontiers in astronomy, biology, chemistry, and physics, among others.

The computer revolution continues. Each time the cost of computing improves by another factor of 10, the opportunities for computers multiply. Applications that were economically infeasible suddenly become practical. In the recent past, the following applications were "computer science fiction."

- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce pollution, improve fuel efficiency via engine controls, and increase safety through the prevention of dangerous skids and through the inflation of air bags to protect occupants in a crash.
- *Cell phones:* Who would have dreamed that advances in computer systems would lead to mobile phones, allowing person-to-person communication almost anywhere in the world?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences is hundreds of millions of dollars. It's unlikely that anyone would have considered this project had the computer costs been 10 to 100 times higher, as they would have been 10 to 20 years ago. Moreover, costs continue to drop; you may be able to acquire your own genome, allowing medical care to be tailored to you.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the World Wide Web has transformed our society. For many, the WWW has replaced libraries.
- *Search engines:* As the content of the WWW grew in size and in value, finding relevant information became increasingly important. Today, many people rely on search engines for such a large part of their lives that it would be a hardship to go without them.

Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, which explains why computers are omnipresent. Today's science fiction suggests tomorrow's killer applications: already on their way are virtual worlds, practical speech recognition, and personalized health care.

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (see Sections 1.3 and 1.7) is used in computers ranging from smart home appliances to cell phones to the largest supercomputers, these different applications have different design requirements and employ the core hardware technologies in different ways. Broadly speaking, computers are used in three different classes of applications.

Desktop computers are possibly the best-known form of computing and are characterized by the personal computer, which readers of this book have likely used extensively. Desktop computers emphasize delivery of good performance to single users at low cost and usually execute third-party software. The evolution of many computing technologies is driven by this class of computing, which is only about 30 years old!

Servers are the modern form of what were once mainframes, minicomputers, and supercomputers, and are usually accessed only via a network. Servers are oriented to carrying large workloads, which may consist of either single complex applications—usually a scientific or engineering application—or handling many small jobs, such as would occur in building a large Web server. These applications are usually based on software from another source (such as a database or simulation system), but are often modified or customized for a particular function. Servers are built from the same basic technology as desktop computers, but provide for greater expandability of both computing and input/output capacity. In general, servers also place a greater emphasis on dependability, since a crash is usually more costly than it would be on a single-user desktop computer.

Servers span the widest range in cost and capability. At the low end, a server may be little more than a desktop computer without a screen or keyboard and cost a thousand dollars. These low-end servers are typically used for file storage, small business applications, or simple Web serving (see Section 6.10). At the other extreme are **supercomputers**, which at the present consist of hundreds to thousands of processors and usually **terabytes** of memory and **petabytes** of storage, and cost millions to hundreds of millions of dollars. Supercomputers are usually used for high-end scientific and engineering calculations, such as weather forecasting, oil exploration, protein structure determination, and other large-scale problems. Although such supercomputers represent the peak of computing capability, they represent a relatively small fraction of the servers and a relatively small fraction of the overall computer market in terms of total revenue.

Although not called supercomputers, Internet **datacenters** used by companies like eBay and Google also contain thousands of processors, terabytes of memory, and petabytes of storage. These are usually considered as large clusters of computers (see Chapter 7).

Embedded computers are the largest class of computers and span the widest range of applications and performance. Embedded computers include the

desktop computer A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.

server A computer used for running larger programs for multiple users, often simultaneously, and typically accessed only via a network.

supercomputer A class of computers with the highest performance and cost; they are configured as servers and typically cost millions of dollars.

terabyte Originally 1,099,511,627,776 (2^{40}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000,000 (10^{12}) bytes.

petabyte Depending on the situation, either 1000 or 1024 terabytes.

datacenter A room or building designed to handle the power, cooling, and networking needs of a large number of servers.

embedded computer A computer inside another device used for running one predetermined application or collection of software.

microprocessors found in your car, the computers in a cell phone, the computers in a video game or television, and the networks of processors that control a modern airplane or cargo ship. Embedded computing systems are designed to run one application or one set of related applications, that are normally integrated with the hardware and delivered as a single system; thus, despite the large number of embedded computers, most users never really see that they are using a computer!

Figure 1.1 shows that during the last several years, the growth in cell phones that rely on embedded computers has been much faster than the growth rate of desktop computers. Note that the embedded computers are also found in digital TVs and set-top boxes, automobiles, digital cameras, music players, video games, and a variety of other such consumer devices, which further increases the gap between the number of embedded computers and desktop computers.

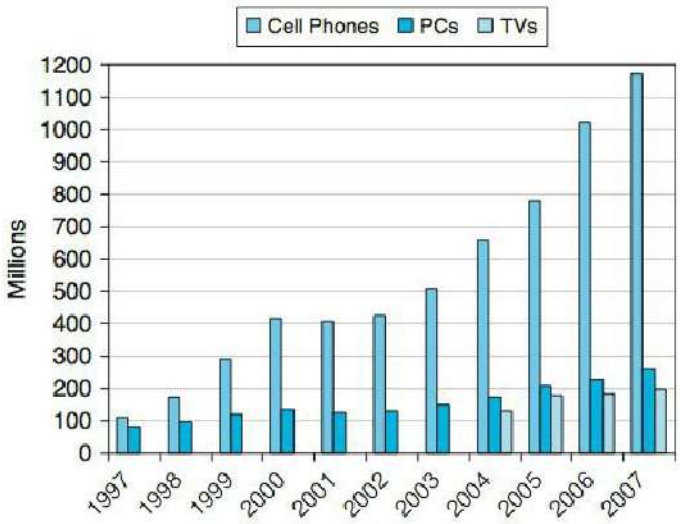


FIGURE 1.1 The number of cell phones, personal computers, and televisions manufactured per year between 1997 and 2007. (We have television data only from 2004.) More than a billion new cell phones were shipped in 2006. Cell phones sales exceeded PCs by only a factor of 1.4 in 1997, but the ratio grew to 4.5 in 2007. The total number in use in 2004 is estimated to be about 2.0B televisions, 1.8B cell phones, and 0.8B PCs. As the world population was about 6.4B in 2004, there were approximately one PC, 2.2 cell phones, and 2.5 televisions for every eight people on the planet. A 2006 survey of U.S. families found that they owned on average 12 gadgets, including three TVs, 2 PCs, and other devices such as game consoles, MP3 players, and cell phones.

Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a music player: the processor need only be as fast as necessary to handle its limited function, and beyond that, minimizing cost and power are the most important objectives. Despite their low cost, embedded computers often have lower tolerance for failure, since the results can vary from upsetting (when your new television crashes) to devastating (such as might occur when the computer in a plane or cargo ship crashes). In consumer-oriented embedded applications, such as a digital home appliance, dependability is achieved primarily through simplicity—the emphasis is on doing one function as perfectly as possible. In large embedded systems, techniques of redundancy from the server world are often employed (see Section 6.9). Although this book focuses on general-purpose computers, most concepts apply directly, or with slight modifications, to embedded computers.

Elaboration: Elaborations are short sections used throughout the text to provide more detail on a particular subject that may be of interest. Disinterested readers may skip over an elaboration, since the subsequent material will never depend on the contents of the elaboration.

Many embedded processors are designed using *processor cores*, a version of a processor written in a hardware description language, such as Verilog or VHDL (see Chapter 4). The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip.

What You Can Learn in This Book

Successful programmers have always been concerned about the performance of their programs, because getting results to the user quickly is critical in creating successful software. In the 1960s and 1970s, a primary constraint on computer performance was the size of the computer's memory. Thus, programmers often followed a simple credo: minimize memory space to make programs fast. In the last decade, advances in computer design and memory technology have greatly reduced the importance of small memory size in most applications other than those in embedded computing systems.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. Programmers who seek to build competitive versions of compilers, operating systems, databases, and even applications will therefore need to increase their knowledge of computer organization.

We are honored to have the opportunity to explain what's inside this revolutionary machine, unraveling the software below your program and the hardware under the covers of your computer. By the time you complete this book, we believe you will be able to answer the following questions:

- How are programs written in a high-level language, such as C or Java, translated into the language of the hardware, and how does the hardware execute the resulting program? Comprehending these concepts forms the basis of understanding the aspects of both the hardware and software that affect program performance.
- What is the interface between the software and the hardware, and how does software instruct the hardware to perform needed functions? These concepts are vital to understanding how to write many kinds of software.
- What determines the performance of a program, and how can a programmer improve the performance? As we will see, this depends on the original program, the software translation of that program into the computer's language, and the effectiveness of the hardware in executing the program.
- What techniques can be used by hardware designers to improve performance? This book will introduce the basic concepts of modern computer design. The interested reader will find much more material on this topic in our advanced book, *Computer Architecture: A Quantitative Approach.*
- What are the reasons for and the consequences of the recent switch from sequential processing to parallel processing? This book gives the motivation, describes the current hardware mechanisms to support parallelism, and surveys the new generation of **"multicore" microprocessors** (see Chapter 7).

multicore microprocessor A microprocessor containing multiple processors ("cores") in a single integrated circuit.

Without understanding the answers to these questions, improving the performance of your program on a modern computer, or evaluating what features might make one computer better than another for a particular application, will be a complex process of trial and error, rather than a scientific procedure driven by insight and analysis.

This first chapter lays the foundation for the rest of the book. It introduces the basic ideas and definitions, places the major components of software and hardware in perspective, shows how to evaluate performance and power, introduces integrated circuits (the technology that fuels the computer revolution), and explains the shift to multicores.

In this chapter and later ones, you will likely see many new words, or words that you may have heard but are not sure what they mean. Don't panic! Yes, there is a lot of special terminology used in describing modern computers, but the terminology actually helps, since it enables us to describe precisely a function or capability. In addition, computer designers (including your authors) *love* using **acronyms**, which are *easy* to understand once you know what the letters stand for! To help you remember and locate terms, we have included a **highlighted** definition of every term in the margins the first time it appears in the text. After a short time of working with the terminology, you will be fluent, and your friends will be impressed as you correctly use acronyms such as BIOS, CPU, DIMM, DRAM, PCIE, SATA, and many others.

acronym A word constructed by taking the initial letters of a string of words. For example: **RAM** is an acronym for Random Access Memory, and **CPU** is an acronym for Central Processing Unit.

To reinforce how the software and hardware systems used to run a program will affect performance, we use a special section, *Understanding Program Performance*, throughout the book to summarize important insights into program performance. The first one appears below.

Understanding Program Performance

The performance of a program depends on a combination of the effectiveness of the algorithms used in the program, the software systems used to create and translate the program into machine instructions, and the effectiveness of the computer in executing those instructions, which may include input/output (I/O) operations. This table summarizes how the hardware and software affect performance.

Hardware or software component	How this component affects performance	Where is this topic covered?
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed	Other books!
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 4, 5, and 7
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapter 6

Check Yourself

Check Yourself sections are designed to help readers assess whether they comprehend the major concepts introduced in a chapter and understand the implications of those concepts. Some *Check Yourself* questions have simple answers; others are for discussion among a group. Answers to the specific questions can be found at the end of the chapter. *Check Yourself* questions appear only at the end of a section, making it easy to skip them if you are sure you understand the material.

1. Section 1.1 showed that the number of embedded processors sold every year greatly outnumbers the number of desktop processors. Can you confirm or deny this insight based on your own experience? Try to count the number of embedded processors in your home. How does it compare with the number of desktop computers in your home?

2. As mentioned earlier, both the software and hardware affect the performance of a program. Can you think of examples where each of the following is the right place to look for a performance bottleneck?

   - The algorithm chosen
   - The programming language or compiler
   - The operating system
   - The processor
   - The I/O system and devices

In Paris they simply stared when I spoke to them in French; I never did succeed in making those idiots understand their own language.

Mark Twain, *The Innocents Abroad*, 1869

1.2 Below Your Program

A typical application, such as a word processor or a large database system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. As we will see, the hardware in a computer can only execute extremely simple low-level instructions. To go from a complex application to the simple instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions.

Figure 1.2 shows that these layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of **systems software** sitting between the hardware and applications software.

systems software Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.

There are many types of systems software, but two types of systems software are central to every computer system today: an operating system and a compiler. An **operating system** interfaces between a user's program and the hardware and provides a variety of services and supervisory functions. Among the most important functions are

operating system Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.

- Handling basic input and output operations
- Allocating storage and memory
- Providing for protected sharing of the computer among multiple applications using it simultaneously.

Examples of operating systems in use today are Linux, MacOS, and Windows.

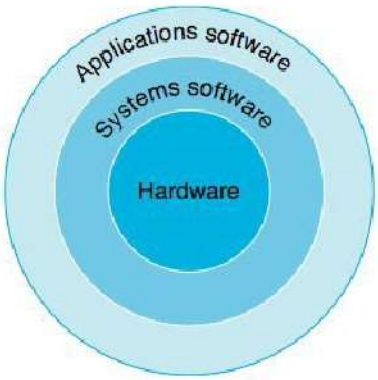


FIGURE 1.2 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.

Compilers perform another vital function: the translation of a program written in a high-level language, such as C, C++, Java, or Visual Basic into instructions that the hardware can execute. Given the sophistication of modern programming languages and the simplicity of the instructions executed by the hardware, the translation from a high-level language program to hardware instructions is complex. We give a brief overview of the process here and then go into more depth in Chapter 2 and Appendix B.

compiler A program that translates high-level language statements into assembly language statements.

From a High-Level Language to the Language of Hardware

To actually speak to electronic hardware, you need to send electrical signals. The easiest signals for computers to understand are *on* and *off*, and so the computer alphabet is just two letters. Just as the 26 letters of the English alphabet do not limit how much can be written, the two letters of the computer alphabet do not limit what computers can do. The two symbols for these two letters are the numbers 0 and 1, and we commonly think of the computer language as numbers in base 2, or *binary numbers*. We refer to each "letter" as a **binary digit** or **bit**. Computers are slaves to our commands, which are called **instructions**. Instructions, which are just collections of bits that the computer understands and obeys, can be thought of as numbers. For example, the bits

binary digit Also called a **bit**. One of the two numbers in base 2 (0 or 1) that are the components of information.

```
1000110010100000
```

instruction A command that computer hardware understands and obeys.

tell one computer to add two numbers. Chapter 2 explains why we use numbers for instructions *and* data; we don't want to steal that chapter's thunder, but using numbers for both instructions and data is a foundation of computing.

The first programmers communicated to computers in binary numbers, but this was so tedious that they quickly invented new notations that were closer to the way humans think. At first, these notations were translated to binary by hand, but this process was still tiresome. Using the computer to help program the computer, the pioneers invented programs to translate from symbolic notation to binary. The first of these programs was named an **assembler**. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

assembler A program that translates a symbolic version of instructions into the binary version.

```
add A,B
```

and the assembler would translate this notation into

```
1000110010100000
```

This instruction tells the computer to add the two numbers A and B. The name coined for this symbolic language, still used today, is **assembly language**. In contrast, the binary language that the machine understands is the **machine language**.

assembly language A symbolic representation of machine instructions.

machine language A binary representation of machine instructions.

Although a tremendous improvement, assembly language is still far from the notations a scientist might like to use to simulate fluid flow or that an accountant might use to balance the books. Assembly language requires the programmer

to write one line for every instruction that the computer will follow, forcing the programmer to think like the computer.

The recognition that a program could be written to translate a more powerful language into computer instructions was one of the great breakthroughs in the early days of computing. Programmers today owe their productivity—and their sanity—to the creation of **high-level programming languages** and compilers that translate programs in such languages into instructions. Figure 1.3 shows the relationships among these programs and languages.

high-level programming language A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for MIPS)

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

Assembler

Binary machine language program (for MIPS)

```
00000000101000100000000100011000
00000000100000100001000001000001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

FIGURE 1.3 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in Chapter 2.

A compiler enables a programmer to write this high-level language expression:

```
A + B
```

The compiler would compile it into this assembly language statement:

```
add A,B
```

As shown above, the assembler would translate this statement into the binary instructions that tell the computer to add the two numbers A and B.

High-level programming languages offer several important benefits. First, they allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols (see Figure 1.3). Moreover, they allow languages to be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on. There are also domain-specific languages for even narrower groups of users, such as those interested in simulation of fluids, for example.

The second advantage of programming languages is improved programmer productivity. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea. Conciseness is a clear advantage of high-level languages over assembly language.

The final advantage is that programming languages allow programs to be independent of the computer on which they were developed, since compilers and assemblers can translate high-level language programs to the binary instructions of any computer. These three advantages are so strong that today little programming is done in assembly language.

1.3 Under the Covers

Now that we have looked below your program to uncover the underlying software, let's open the covers of your computer to learn about the underlying hardware. The underlying hardware in any computer performs the same basic functions: inputting data, outputting data, processing data, and storing data. How these functions are performed is the primary topic of this book, and subsequent chapters deal with different parts of these four tasks.

When we come to an important point in this book, a point so important that we hope you will remember it forever, we emphasize it by identifying it as a *Big Picture* item. We have about a dozen Big Pictures in this book, the first being

the five components of a computer that perform the tasks of inputting, outputting, processing, and storing data.

The BIG Picture

The five classic components of a computer are input, output, memory, datapath, and control, with the last two sometimes combined and called the processor. Figure 1.4 shows the standard organization of a computer. This organization is independent of hardware technology: you can place every piece of every computer, past and present, into one of these five categories. To help you keep all this in perspective, the five components of a computer are shown on the front page of each of the following chapters, with the portion of interest to that chapter highlighted.

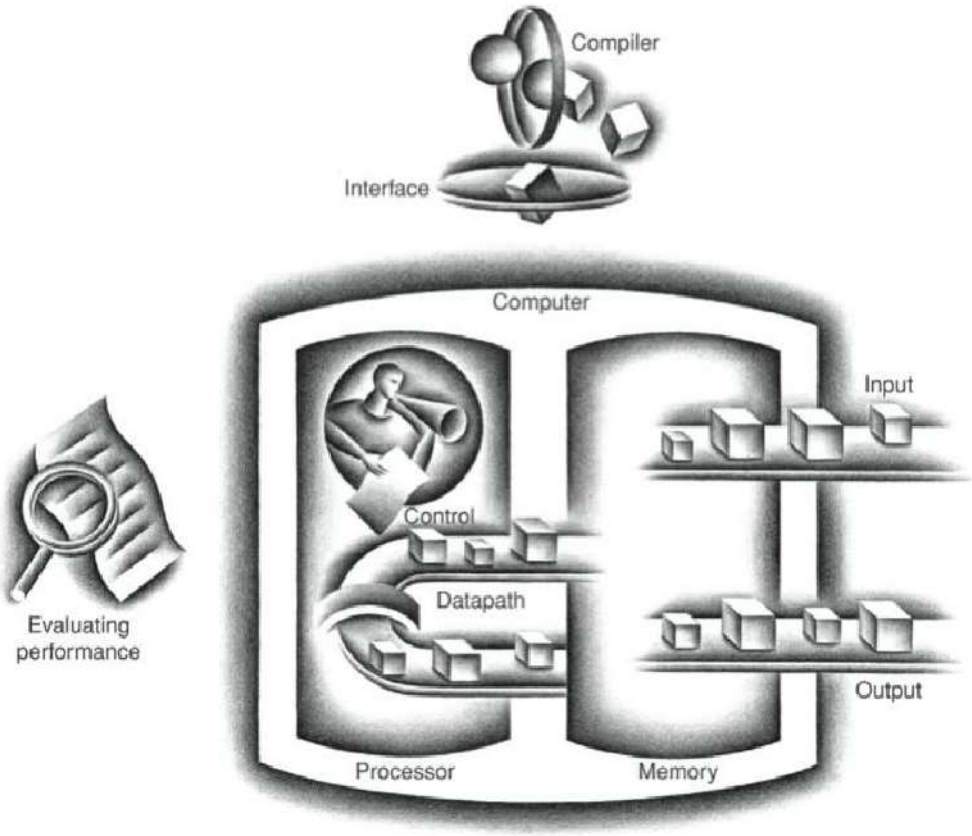


FIGURE 1.4 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

FIGURE 1.5 A desktop computer. The liquid crystal display (LCD) screen is the primary output device, and the keyboard and mouse are the primary input devices. On the right side is an Ethernet cable that connected the laptop to the network and the Web. The laptop contains the processor, memory, and additional I/O devices. This system is a Macbook Pro 15" laptop connected to an external display.

Figure 1.5 shows a computer with keyboard, wireless mouse, and screen. This photograph reveals two of the key components of computers: **input devices**, such as the keyboard and mouse, and **output devices**, such as the screen. As the names suggest, input feeds the computer, and output is the result of computation sent to the user. Some devices, such as networks and disks, provide both input and output to the computer.

Chapter 6 describes input/output (I/O) devices in more detail, but let's take an introductory tour through the computer hardware, starting with the external I/O devices.

input device
A mechanism through which the computer is fed information, such as the keyboard or mouse.

output device
A mechanism that conveys the result of a computation to a user or another computer.

I got the idea for the mouse while attending a talk at a computer conference. The speaker was so boring that I started daydreaming and hit upon the idea.

Doug Engelbart

Anatomy of a Mouse

Although many users now take mice for granted, the idea of a pointing device such as a mouse was first shown by Doug Engelbart using a research prototype in 1967. The Alto, which was the inspiration for all workstations as well as for the Macintosh and Windows OS, included a mouse as its pointing device in 1973. By the 1990s, all desktop computers included this device, and new user interfaces based on graphics displays and mice became the norm.

The original mouse was electromechanical and used a large ball that when rolled across a surface would cause an x and y counter to be incremented. The amount of increase in each counter told how far the mouse had been moved.

The electromechanical mouse has largely been replaced by the newer all-optical mouse. The optical mouse is actually a miniature optical processor including an LED to provide lighting, a tiny black-and-white camera, and a simple optical processor. The LED illuminates the surface underneath the mouse; the camera takes 1500 sample pictures a second under the illumination. Successive pictures are sent to a simple optical processor that compares the images and determines whether the mouse has moved and how far. The replacement of the electromechanical mouse by the electro-optical mouse is an illustration of a common phenomenon where the decreasing costs and higher reliability of electronics cause an electronic solution to replace the older electromechanical technology. On page 22 we'll see another example: flash memory.

Through computer displays I have landed an airplane on the deck of a moving carrier, observed a nuclear particle hit a potential well, flown in a rocket at nearly the speed of light and watched a computer reveal its innermost workings.

Ivan Sutherland, the "father" of computer graphics, *Scientific American*, 1984

Through the Looking Glass

The most fascinating I/O device is probably the graphics display. All laptop and handheld computers, calculators, cellular phones, and almost all desktop computers now use **liquid crystal displays (LCDs)** to get a thin, low-power display. The LCD is not the source of light; instead, it controls the transmission of light. A typical LCD includes rod-shaped molecules in a liquid that form a twisting helix that bends light entering the display, from either a light source behind the display or less often from reflected light. The rods straighten out when a current is applied and no longer bend the light. Since the liquid crystal material is between two screens polarized at 90 degrees, the light cannot pass through unless it is bent. Today, most LCD displays use an **active matrix** that has a tiny transistor switch at each pixel to precisely control current and make sharper images. A red-green-blue mask associated with each dot on the display determines the intensity of the three color components in the final image; in a color active matrix LCD, there are three transistor switches at each point.

The image is composed of a matrix of picture elements, or **pixels**, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix ranges in size from 640×480 to 2560×1600 pixels in 2008. A color display might use 8 bits for each of the three colors (red, blue, and green), for 24 bits per pixel, permitting millions of different colors to be displayed.

liquid crystal display A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.

active matrix display A liquid crystal display using a transistor to control the transmission of light at each individual pixel.

pixel The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

The computer hardware support for graphics consists mainly of a *raster refresh buffer*, or *frame buffer*, to store the bit map. The image to be represented onscreen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 1.6 shows a frame buffer with a simplified design of just 4 bits per pixel.

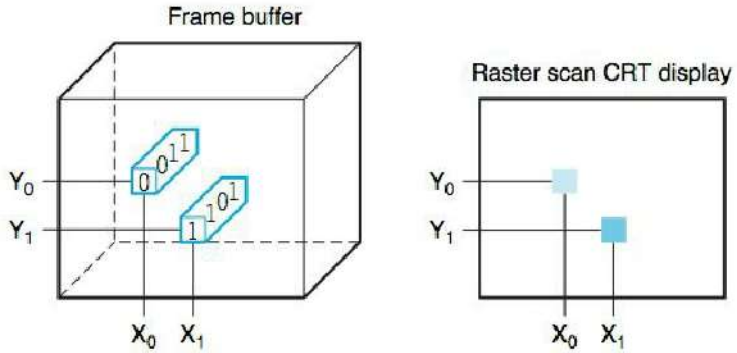


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1).

The goal of the bit map is to faithfully represent what is on the screen. The challenges in graphics systems arise because the human eye is very good at detecting even subtle changes on the screen.

Opening the Box

If we open the box containing the computer, we see a fascinating board of thin plastic, covered with dozens of small gray or black rectangles. Figure 1.7 shows the contents of the laptop computer in Figure 1.5. The **motherboard** is shown in the upper part of the photo. Two disk drives are in front—the hard drive on the left and a DVD drive on the right. The hole in the middle is for the laptop battery.

The small rectangles on the motherboard contain the devices that drive our advancing technology, called **integrated circuits** and nicknamed **chips**. The board is composed of three pieces: the piece connecting to the I/O devices mentioned earlier, the memory, and the processor.

The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. Figure 1.8 shows that memory is found on the two small boards, and each small memory board contains eight integrated circuits. The memory in Figure 1.8 is built from DRAM chips. *DRAM*

motherboard A plastic board containing packages of integrated circuits or chips, including processor, cache, memory, and connectors for I/O devices such as networks and disks.

integrated circuit Also called a **chip**. A device combining dozens to millions of transistors.

memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

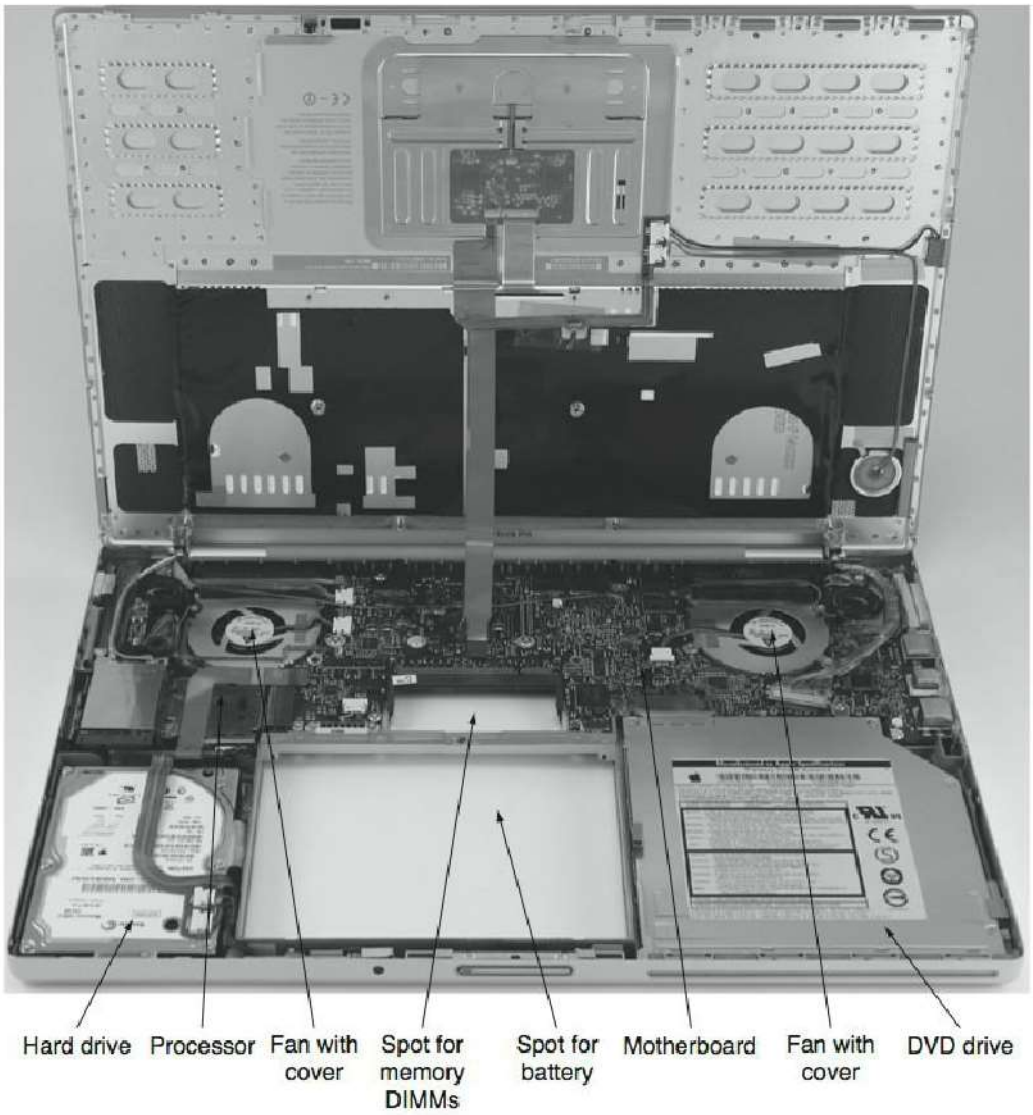


FIGURE 1.7 Inside the laptop computer of Figure 1.5. The shiny box with the white label on the lower left is a 100 GB SATA hard disk drive, and the shiny metal box on the lower right side is the DVD drive. The hole between them is where the laptop battery would be located. The small hole above the battery hole is for memory DIMMs. Figure 1.8 is a close-up of the DIMMs, which are inserted from the bottom in this laptop. Above the battery hole and DVD drive is a printed circuit board (PC board), called the *motherboard*, which contains most of the electronics of the computer. The two shiny circles in the upper half of the picture are two fans with covers. The processor is the large raised rectangle just below the left fan. Photo courtesy of OtherWorldComputing.com.

stands for **dynamic random access memory**. Several DRAMs are used together to contain the instructions and data of a program. In contrast to sequential access memories, such as magnetic tapes, the *RAM* portion of the term DRAM means that memory accesses take basically the same amount of time no matter what portion of the memory is read.

dynamic random access memory (DRAM) Memory built as an integrated circuit; it provides random access to any location.

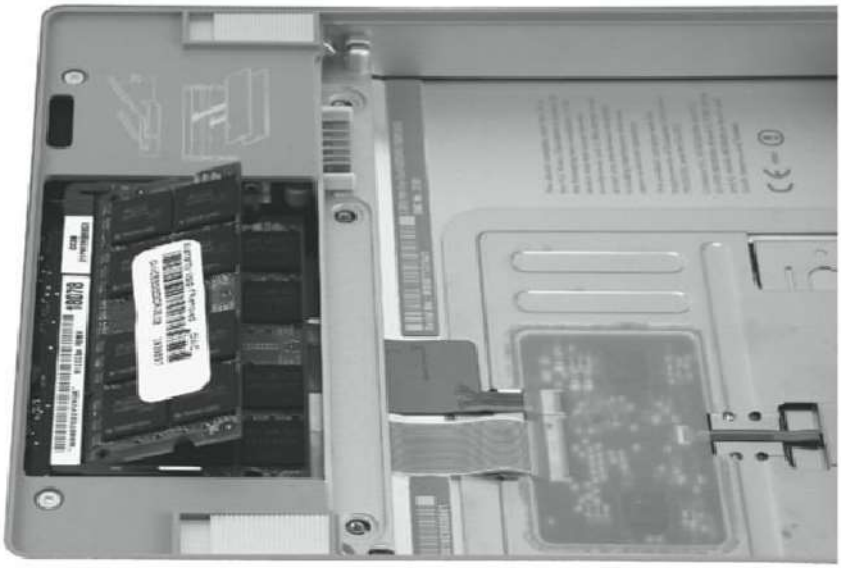


FIGURE 1.8 Close-up of the bottom of the laptop reveals the memory. The main memory is contained on one or more small boards shown on the left. The hole for the battery is to the right. The DRAM chips are mounted on these boards (called **DIMMs**, for dual inline memory modules) and then plugged into the connectors. Photo courtesy of OtherWorldComputing.com.

dual inline memory module (DIMM) A small board that contains DRAM chips on both sides. (SIMMs have DRAMs on only one side.)

The *processor* is the active part of the board, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. The processor is under the fan and covered by a heat sink on the left side of Figure 1.7. Occasionally, people call the processor the **CPU**, for the more bureaucratic-sounding **central processor unit**.

central processor unit (CPU) Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

Descending even lower into the hardware, Figure 1.9 reveals details of a microprocessor. The processor logically comprises two main components: datapath and control, the respective brawn and brain of the processor. The **datapath** performs the arithmetic operations, and **control** tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program. Chapter 4 explains the datapath and control for a higher-performance design.

datapath The component of the processor that performs arithmetic operations

control The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

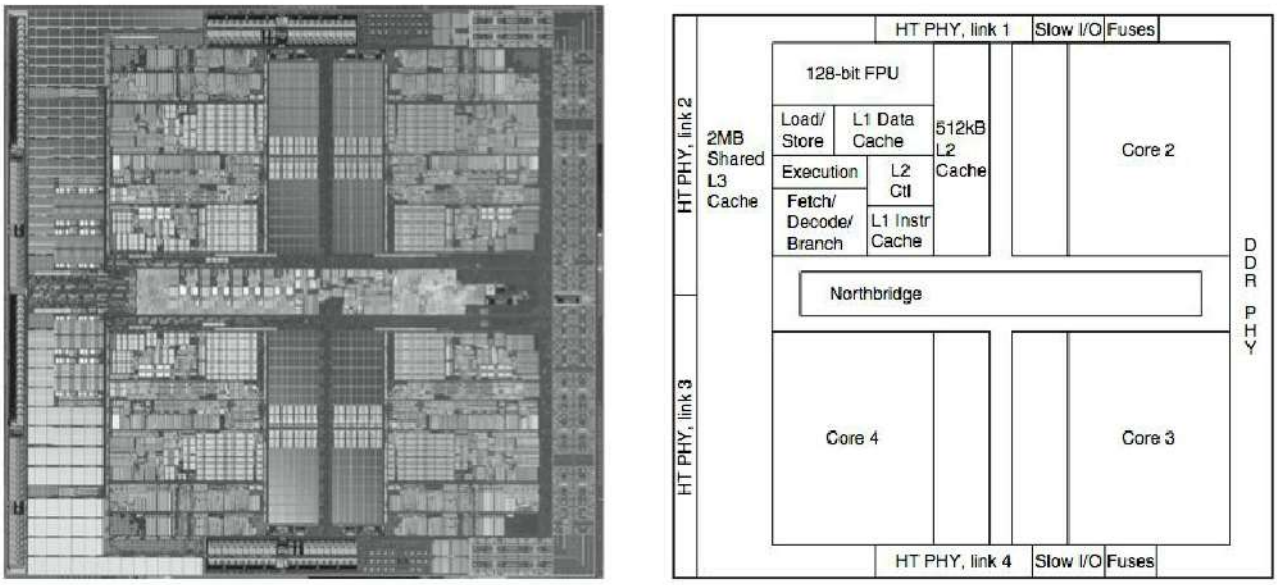


FIGURE 1.9 Inside the AMD Barcelona microprocessor. The left-hand side is a microphotograph of the AMD Barcelona processor chip, and the right-hand side shows the major blocks in the processor. This chip has four processors or "cores". The microprocessor in the laptop in Figure 1.7 has two cores per chip, called an Intel Core 2 Duo.

cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

static random access memory (SRAM) Also memory built as an integrated circuit, but faster and less dense than DRAM.

abstraction A model that renders lower-level details of computer systems temporarily invisible to facilitate design of sophisticated systems.

Descending into the depths of any component of the hardware reveals insights into the computer. Inside the processor is another type of memory—cache memory. **Cache memory** consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, **static random access memory (SRAM)**. SRAM is faster but less dense, and hence more expensive, than DRAM (see Chapter 5).

You may have noticed a common theme in both the software and the hardware descriptions: delving into the depths of hardware or software reveals more information or, conversely, lower-level details are hidden to offer a simpler model at higher levels. The use of such layers, or **abstractions**, is a principal technique for designing very sophisticated computer systems.

One of the most important abstractions is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special

name: the **instruction set architecture**, or simply **architecture**, of a computer. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details. The combination of the basic instruction set and the operating system interface provided for application programmers is called the **application binary interface** (**ABI**).

An instruction set architecture allows computer designers to talk about functions independently from the hardware that performs them. For example, we can talk about the functions of a digital clock (keeping time, displaying the time, setting the alarm) independently from the clock hardware (quartz crystal, LED displays, plastic buttons). Computer designers distinguish architecture from an **implementation** of an architecture along the same lines: an implementation is hardware that obeys the architecture abstraction. These ideas bring us to another Big Picture.

instruction set architecture Also called **architecture**. An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, ...

application binary interface (ABI) The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

implementation Hardware that obeys the architecture abstraction.

The BIG Picture

Both hardware and software consist of hierarchical layers, with each lower layer hiding details from the level above. This principle of *abstraction* is the way both hardware designers and software designers cope with the complexity of computer systems. One key interface between the levels of abstraction is the *instruction set architecture*—the interface between the hardware and low-level software. This abstract interface enables many *implementations* of varying cost and performance to run identical software.

A Safe Place for Data

Thus far, we have seen how to input data, compute using the data, and display data. If we were to lose power to the computer, however, everything would be lost because the memory inside the computer is **volatile**—that is, when it loses power, it forgets. In contrast, a DVD doesn't forget the recorded film when you turn off the power to the DVD player and is thus a **nonvolatile memory** technology.

To distinguish between the volatile memory used to hold data and programs while they are running and this nonvolatile memory used to store data and programs between runs, the term **main memory** or **primary memory** is used for the

volatile memory Storage, such as DRAM, that retains data only if it is receiving power.

nonvolatile memory A form of memory that retains data even in the absence of a power source and that is used to store programs between runs. Magnetic disk is nonvolatile.

main memory Also called **primary memory**. Memory used to hold programs while they are running; typically consists of DRAM in today's computers.

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of magnetic disks in today's computers.

magnetic disk Also called **hard disk**. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.

flash memory A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive and faster than magnetic disks.

former, and **secondary memory** for the latter. DRAMs have dominated main memory since 1975, but **magnetic disks** have dominated secondary memory since 1965. The primary nonvolatile storage used in all server computers and workstations is the magnetic **hard disk**. **Flash memory**, a nonvolatile semiconductor memory, is used instead of disks in mobile devices such as cell phones and is increasingly replacing disks in music players and even laptops.

As Figure 1.10 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

FIGURE 1.10 A disk showing 10 disk platters and the read/write heads.

Diameters of hard disks vary by more than a factor of 3 today, from 1 inch to 3.5 inches, and have been shrunk over the years to fit into new products; workstation servers, personal computers, laptops, palmtops, and digital cameras have all inspired new disk form factors. Traditionally, the widest disks have the highest performance and the smallest disks have the lowest unit cost. The best cost per **gigabyte** varies. Although most hard drives appear inside computers, as in Figure 1.7, hard drives can also be attached using external interfaces such as universal serial bus (USB).

gigabyte Traditionally 1,073,741,824 (2^{30}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000 (10^9) bytes. Similarly, depending on the context, megabyte is either 2^{20} or 10^6 bytes.

The use of mechanical components means that access times for magnetic disks are much slower than for DRAMs: disks typically take 5–20 milliseconds, while DRAMs take 50–70 nanoseconds—making DRAMs about 100,000 times faster. Yet disks have much lower costs than DRAM for the same storage capacity, because the production costs for a given amount of disk storage are lower than for the same amount of integrated circuit. In 2008, the cost per gigabyte of disk is 30 to 100 times less expensive than DRAM.

Thus, there are three primary differences between magnetic disks and main memory: disks are nonvolatile because they are magnetic; they have a slower access time because they are mechanical devices; and they are cheaper per gigabyte because they have very high storage capacity at a modest cost.

Many have tried to invent a technology cheaper than DRAM but faster than disk to fill that gap, but many have failed. Challengers have never had a product to market at the right time. By the time a new product would ship, DRAMs and disks had continued to make rapid advances, costs had dropped accordingly, and the challenging product was immediately obsolete.

Flash memory, however, is a serious challenger. This semiconductor memory is nonvolatile like disks and has about the same bandwidth, but latency is 100 to 1000 times faster than disk. Flash is popular in cameras and portable music players because it comes in much smaller capacities, it is more rugged, and it is more power efficient than disks, despite the cost per gigabyte in 2008 being about 6 to 10 times higher than disk. Unlike disks and DRAM, flash memory bits wear out after 100,000 to 1,000,000 writes. Thus, file systems must keep track of the number of writes and have a strategy to avoid wearing out storage, such as by moving popular data. Chapter 6 describes flash in more detail.

Although hard drives are not removable, there are several storage technologies in use that include the following:

- Optical disks, including both compact disks (CDs) and digital video disks (DVDs), constitute the most common form of removable storage. The Blu-Ray (BD) optical disk standard is the heir-apparent to DVD.
- Flash-based removable memory cards typically attach to a USB connection and are often used to transfer files.
- Magnetic tape provides only slow serial access and has been used to back up disks, a role now often replaced by duplicate hard drives.

Optical disk technology works differently than magnetic disk technology. In a CD, data is recorded in a spiral fashion, with individual bits being recorded by burning small pits—approximately 1 micron (10^{-6} meters) in diameter—into the disk surface. The disk is read by shining a laser at the CD surface and determining by examining the reflected light whether there is a pit or flat (reflective) surface. DVDs use the same approach of bouncing a laser beam off a series of pits and flat surfaces. In addition, there are multiple layers that the laser beam can focus on, and the size of each bit is much smaller, which together increase capacity significantly. Blu-Ray uses shorter wavelength lasers that shrink the size of the bits and thereby increase capacity.

Optical disk writers in personal computers use a laser to make the pits in the recording layer on the CD or DVD surface. This writing process is relatively slow, taking from minutes (for a full CD) to tens of minutes (for a full DVD). Thus, for large quantities a different technique called *pressing* is used, which costs only pennies per optical disk.

Rewritable CDs and DVDs use a different recording surface that has a crystalline, reflective material; pits are formed that are not reflective in a manner similar to that for a write-once CD or DVD. To erase the CD or DVD, the surface is heated and cooled slowly, allowing an annealing process to restore the surface recording layer to its crystalline structure. These rewritable disks are the most expensive, with write-once being cheaper; for read-only disks—used to distribute software, music, or movies—both the disk cost and recording cost are much lower.

Communicating with Other Computers

We've explained how we can input, compute, display, and save data, but there is still one missing item found in today's computers: computer networks. Just as the processor shown in Figure 1.4 is connected to memory and I/O devices, networks interconnect whole computers, allowing computer users to extend the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new computer without an optional network interface would be ridiculed. Networked computers have several major advantages:

- *Communication:* Information is exchanged between computers at high speeds.
- *Resource sharing:* Rather than each computer having its own I/O devices, devices can be shared by computers on the network.
- *Nonlocal access:* By connecting computers over long distances, users need not be near the computer they are using.

Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is *Ethernet*. It can be up to a kilometer long and transfer at upto 10 gigabits per second. Its length and

speed make Ethernet useful to connect computers on the same floor of a building; hence, it is an example of what is generically called a **local area network**. Local area networks are interconnected with switches that can also provide routing services and security. **Wide area networks** cross continents and are the backbone of the Internet, which supports the World Wide Web. They are typically based on optical fibers and are leased from telecommunication companies.

local area network (LAN) A network designed to carry data within a geographically confined area, typically within a single building.

wide area network (WAN) A network extended over hundreds of kilometers that can span a continent.

Networks have changed the face of computing in the last 25 years, both by becoming much more ubiquitous and by making dramatic increases in performance. In the 1970s, very few individuals had access to electronic mail, the Internet and Web did not exist, and physically mailing magnetic tapes was the primary way to transfer large amounts of data between two locations. Local area networks were almost nonexistent, and the few existing wide area networks had limited capacity and restricted access.

As networking technology improved, it became much cheaper and had a much higher capacity. For example, the first standardized local area network technology, developed about 25 years ago, was a version of Ethernet that had a maximum capacity (also called bandwidth) of 10 million bits per second, typically shared by tens of, if not a hundred, computers. Today, local area network technology offers a capacity of from 100 million bits per second to 10 gigabits per second, usually shared by at most a few computers. Optical communications technology has allowed similar growth in the capacity of wide area networks, from hundreds of kilobits to gigabits and from hundreds of computers connected to a worldwide network to millions of computers connected. This combination of dramatic rise in deployment of networking combined with increases in capacity have made network technology central to the information revolution of the last 25 years.

For the last decade another innovation in networking is reshaping the way computers communicate. Wireless technology is widespread, and laptops now incorporate this technology. The ability to make a radio in the same low-cost semiconductor technology (CMOS) used for memory and microprocessors enabled a significant improvement in price, leading to an explosion in deployment. Currently available wireless technologies, called by the IEEE standard name 802.11, allow for transmission rates from 1 to nearly 100 million bits per second. Wireless technology is quite a bit different from wire-based networks, since all users in an immediate area share the airwaves.

Check Yourself

- Semiconductor DRAM and disk storage differ significantly. Describe the fundamental difference for each of the following: volatility, access time, and cost.

Technologies for Building Processors and Memory

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. Figure 1.11 shows the technologies that have been

used over time, with an estimate of the relative performance per unit cost for each technology. Section 1.7 explores the technology that has fueled the computer industry since 1975 and will continue to do so for the foreseeable future. Since this technology shapes what computers will be able to do and how quickly they will evolve, we believe all computer professionals should be familiar with the basics of integrated circuits.

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2005	Ultra large-scale integrated circuit	6,200,000,000

FIGURE 1.11 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2005 extrapolated by the authors. See Section 1.10 on the CD.

vacuum tube An electronic component, predecessor of the transistor, that consists of a hollow glass tube about 5 to 10 cm long from which as much air has been removed as possible and that uses an electron beam to transfer data.

transistor An on/off switch controlled by an electric signal.

very large-scale integrated (VLSI) circuit A device containing hundreds of thousands to millions of transistors.

A **transistor** is simply an on/off switch controlled by electricity. The *integrated circuit* (IC) combined dozens to hundreds of transistors into a single chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**.

This rate of increasing integration has been remarkably stable. Figure 1.12 shows the growth in DRAM capacity since 1977. For 20 years, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times! This increase in transistor count for an integrated circuit is popularly known as Moore's law, which states that transistor capacity doubles every 18–24 months. Moore's law resulted from a prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel during the 1960s.

Sustaining this rate of progress for almost 40 years has required incredible innovation in manufacturing techniques. In Section 1.7, we discuss how to manufacture integrated circuits.

1.4 Performance

Assessing the performance of computers can be quite challenging. The scale and intricacy of modern software systems, together with the wide range of performance improvement techniques employed by hardware designers, have made performance assessment much more difficult.

When trying to choose among different computers, performance is an important attribute. Accurately measuring and comparing different computers is critical to

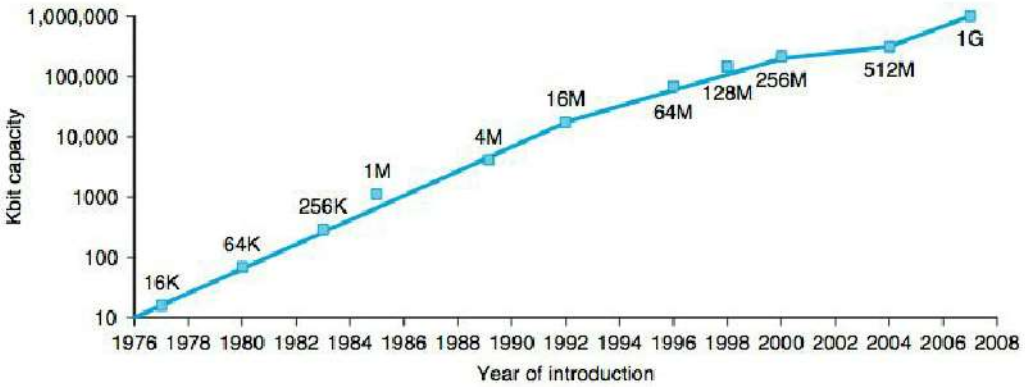


FIGURE 1.12 Growth of capacity per DRAM chip over time. The y-axis is measured in Kilobits, where K = 1024 (2^{10}). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two years to three years.

purchasers and therefore to designers. The people selling computers know this as well. Often, salespeople would like you to see their computer in the best possible light, whether or not this light accurately reflects the needs of the purchaser's application. Hence, understanding how best to measure performance and the limitations of performance measurements is important in selecting a computer.

The rest of this section describes different ways in which performance can be determined; then, we describe the metrics for measuring performance from the viewpoint of both a computer user and a designer. We also look at how these metrics are related and present the classical processor performance equation, which we will use throughout the text.

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. Figure 1.13 shows some typical passenger airplanes, together with their cruising speed, range, and capacity. If we wanted to know which of the planes in this table had the best performance, we would first need to define performance. For example, considering different measures of performance, we see that the plane with the highest cruising speed is the Concorde, the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

Let's suppose we define performance in terms of speed. This still leaves two possible definitions. You could define the fastest plane as the one with the highest cruising speed, taking a single passenger from one point to another in the least time. If you

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers × m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

FIGURE 1.13 The capacity, range, and speed for a number of commercial airplanes. The last column shows the rate at which the airplane transports passengers, which is the capacity times the cruising speed (ignoring range and takeoff and landing times).

response time Also called **execution time**. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

throughput Also called **bandwidth**. Another measure of performance, it is the number of tasks completed per unit time.

were interested in transporting 450 passengers from one point to another, however, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred to as **execution time**. Datacenter managers are often interested in increasing **throughput** or **bandwidth**—the total amount of work done in a given time. Hence, in most cases, we will need different performance metrics as well as different sets of applications to benchmark embedded and desktop computers, which are more focused on response time, versus servers, which are more focused on throughput.

EXAMPLE

Throughput and Response Time

Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the World Wide Web

ANSWER

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases.

If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other.

In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase "X is n times faster than Y"—or equivalently "X is n times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times faster than Y, then the execution time on Y is n times longer than it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Relative Performance

EXAMPLE

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

ANSWER

We know that A is n times faster than B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times faster than B.

In the above example, we could also say that computer B is 1.5 times *slower than* computer A, since

$$\frac{\text{Performance}_A}{\text{Performance}_B} = 1.5$$

means that

$$\frac{\text{Performance}_A}{1.5} = \text{Performance}_B$$

For simplicity, we will normally use the terminology *faster than* when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*, we usually say "improve performance" or "improve execution time" when we mean "increase performance" and "decrease execution time."

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. However, time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall clock time, response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.

CPU execution time Also called **CPU time**. The actual time the CPU spends computing for a specific task.

user CPU time The CPU time spent in a program itself.

system CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time that the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to

do accurately, because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems.

For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term *system performance* to refer to elapsed time on an unloaded system and *CPU performance* to refer to user CPU time. We will focus on CPU performance in this chapter, although our discussions of how to summarize performance can be applied to either elapsed time or CPU time measurements.

Understanding Program Performance

Different applications are sensitive to different aspects of the performance of a computer system. Many applications, especially those running on servers, depend as much on I/O performance, which, in turn, relies on both hardware and software. Total elapsed time measured by a wall clock is the measurement of interest. In some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to look for performance bottlenecks by measuring program execution and looking for the likely bottlenecks. In the following chapters, we will describe how to search for bottlenecks and improve performance in various parts of the system.

Although as computer users we care about time, when we examine the details of a computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that determines when events take place in the hardware. These discrete time intervals are called **clock cycles** (or **ticks**, **clock ticks**, **clock periods**, **clocks**, **cycles**). Designers refer to the length of a **clock period** both as the time for a complete *clock cycle* (e.g., 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

clock cycle Also called **tick**, **clock tick**, **clock period**, **clock**, **cycle**. The time for one clock period, usually of the processor clock, which runs at a constant rate.

clock period The length of each clock cycle.

Check Yourself

1. Suppose we know that an application that uses both a desktop client and a remote server is limited by network performance. For the following changes, state whether only the throughput improves, both response time and throughput improve, or neither improves.

   a. An extra network channel is added between the client and the server, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).

b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.

c. More memory is added to the computer.

2. Computer C's performance is 4 times faster than the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as experienced by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\begin{array}{c}\text{CPU execution time}\\ \text{for a program}\end{array} = \begin{array}{c}\text{CPU clock cycles}\\ \text{for a program}\end{array} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{array}{c}\text{CPU execution time}\\ \text{for a program}\end{array} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle. As we will see in later chapters, the designer often faces a trade-off between the number of clock cycles needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles may also increase the clock cycle time.

EXAMPLE

Improving Performance

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

ANSWER

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. (We'll see what the instructions that make up a program look like in the next chapter.) However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$

clock cycles per instruction (CPI) Average number of clock cycles per instruction for a program or program fragment.

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**. Since different

instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

Using the Performance Equation

EXAMPLE

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

ANSWER

We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}$$

$$= I \times 2.0 \times 250\,\text{ps} = 500 \times I\,\text{ps}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500\,\text{ps} = 600 \times I\,\text{ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I\,\text{ps}}{500 \times I\,\text{ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

The Classic CPU Performance Equation

We can now write this basic performance equation in terms of **instruction count** (the number of instructions executed by the program), CPI, and clock cycle time:

instruction count The number of instructions executed by the program.

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.

Comparing Code Segments

EXAMPLE

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

ANSWER

Sequence 1 executes $2 + 1 + 2 = 5$ instructions. Sequence 2 executes $4 + 1 + 1 = 6$ instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

The BIG Picture

Figure 1.14 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds per program:

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time or higher CPI that offsets the improvement in instruction count. Similarly, because CPI depends on type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

FIGURE 1.14 The basic components of performance and how each is measured.

How can we determine the value of these factors in the performance equation? We can measure the CPU execution time by running the program, and the clock cycle time is usually published as part of the documentation for a computer. The instruction count and CPI can be more difficult to obtain. Of course, if we know the clock rate and CPU execution time, we need only one of the instruction count or the CPI to determine the other.

We can measure the instruction count by using software tools that profile the execution or by using a simulator of the architecture. Alternatively, we can use hardware counters, which are included in most processors, to record a variety of measurements, including the number of instructions executed, the average CPI, and often, the sources of performance loss. Since the instruction count depends on the architecture, but not on the exact implementation, we can measure the instruction count without knowing all the details of the implementation. The CPI, however, depends on a wide variety of design details in the computer, including both the memory system and the processor structure (as we will see in Chapters 4 and 5), as well as on the mix of instruction types executed in an application. Thus, CPI varies by application, as well as among implementations with the same instruction set.

The above example shows the danger of using only one factor (instruction count) to assess performance. When comparing two computers, you must look at all three components, which combine to form execution time. If some of the factors are identical, like the clock rate in the above example, performance can be determined by comparing all the nonidentical factors. Since CPI varies by **instruction mix**, both instruction count and CPI must be compared, even if clock rates are identical. Several exercises at the end of this chapter ask you to evaluate a series of computer and compiler enhancements that affect clock rate, CPI, and instruction count. In Section 1.8, we'll examine a common performance measurement that does not incorporate all the terms and can thus be misleading.

instruction mix
A measure of the dynamic frequency of instructions across one or many programs.

Understanding Program Performance

The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more floating-point operations, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

Elaboration: Although you might expect that the minimum CPI is 1.0, as we'll see in Chapter 4, some processors fetch and execute multiple instructions per clock cycle. To reflect that approach, some designers invert CPI to talk about *IPC*, or *instructions per clock cycle*. If a processor executes on average 2 instructions per clock cycle, then it has an IPC of 2 and hence a CPI of 0.5.

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below

a. $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$

b. $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$

c. $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

1.5 The Power Wall

Figure 1.15 shows the increase in clock rate and power of eight generations of Intel microprocessors over 25 years. Both clock rate and power increased rapidly for decades, and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.

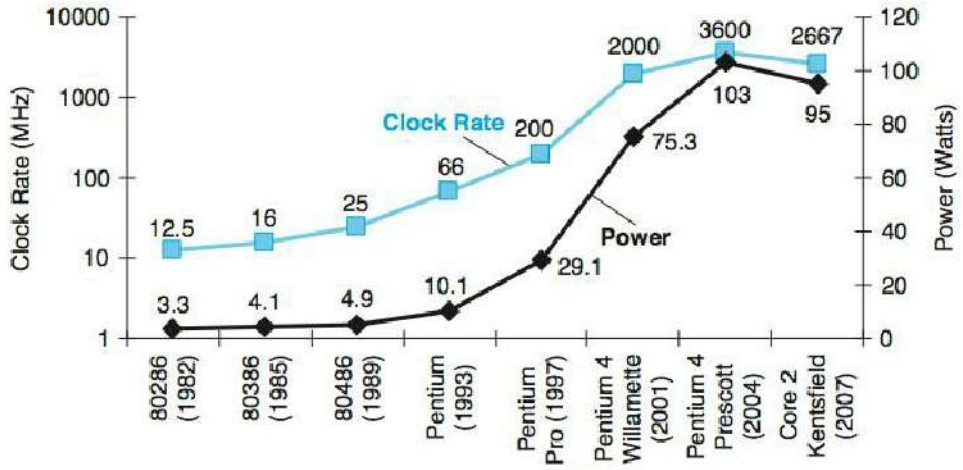


FIGURE 1.15 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip.

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of power dissipation is so-called dynamic power—that is, power that is consumed during switching. The dynamic power dissipation depends on the capacitive loading of each transistor, the voltage applied, and the frequency that the transistor is switched:

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

Frequency switched is a function of the clock rate. The capacitive load per transistor is a function of both the number of transistors connected to an output (called the *fanout*) and the technology, which determines the capacitance of both wires and transistors.

How could clock rates grow by a factor of 1000 while power grew by only a factor of 30? Power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. Typically, the voltage was reduced about 15% per generation. In 20 years, voltages have gone from 5V to 1V, which is why the increase in power is only 30 times.

EXAMPLE

Relative Power

Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it has adjustable voltage so that it can reduce voltage 15% compared to processor B, which results in a 15% shrink in frequency. What is the impact on dynamic power?

ANSWER

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{\langle \text{Capacitive load} \times 0.85 \rangle \times \langle \text{Voltage} \times 0.85 \rangle^2 \times \langle \text{Frequency switched} \times 0.85 \rangle}{\text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}}$$

Thus the power ratio is

$$0.85^4 = 0.52$$

Hence, the new processor uses about half the power of the old processor.

The problem today is that further lowering of the voltage appears to make the transistors too leaky, like water faucets that cannot be completely shut off. Even today about 40% of the power consumption is due to leakage. If transistors started leaking more, the whole process could become unwieldy.

To try to address the power problem, designers have already attached large devices to increase cooling, and they turn off parts of the chip that are not used in a given clock cycle. Although there are many more expensive ways to cool chips and thereby raise their power to, say, 300 watts, these techniques are too expensive for desktop computers.

Since computer designers slammed into a power wall, they needed a new way forward. They chose a different way from the way they designed microprocessors for their first 30 years.

Elaboration: Although dynamic power is the primary source of power dissipation in CMOS, static power dissipation occurs because of leakage current that flows even when a transistor is off. As mentioned above, leakage is typically responsible for 40% of the power consumption in 2008. Thus, increasing the number of transistors increases power dissipation, even if the transistors are always off. A variety of design techniques and technology innovations are being deployed to control leakage, but it's hard to lower voltage further.

1.6 The Sea Change: The Switch from Uniprocessors to Multiprocessors

"Up to now, most software has been like music written for a solo performer; with the current generation of chips we're getting a little experience with duets and quartets and other small ensembles; but scoring a work for large orchestra and chorus is a different kind of challenge."

Brian Hayes, *Computing in a Parallel Universe*, 2007.

The power limit has forced a dramatic change in the design of microprocessors. Figure 1.16 shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year.

Rather than continuing to decrease the response time of a single program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor and microprocessor, companies refer to processors as "cores," and such microprocessors are generically called multicore microprocessors. Hence, a "quadcore" microprocessor is a chip that contains four processors or four cores.

Figure 1.17 shows the number of processors (cores), power, and clock rates of recent microprocessors. The official plan of record for many companies is to double the number of cores per microprocessor per semiconductor technology generation, which is about every two years (see Chapter 7).

In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve performance of their code as the number of cores doubles.

To reinforce how the software and hardware systems work hand in hand, we use a special section, *Hardware/Software Interface*, throughout the book, with the first one appearing below. These elements summarize important insights at this critical interface.

Hardware/Software Interface

Parallelism has always been critical to performance in computing, but it was often hidden. Chapter 4 will explain pipelining, an elegant technique that runs programs faster by overlapping the execution of instructions. This is one example of *instruction-level parallelism*, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.

Forcing programmers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel had been the "third rail" of computer architecture, for companies in the past that depended on such a change in behavior failed (see **Section 7.14** on the CD). From this historical perspective, it's startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

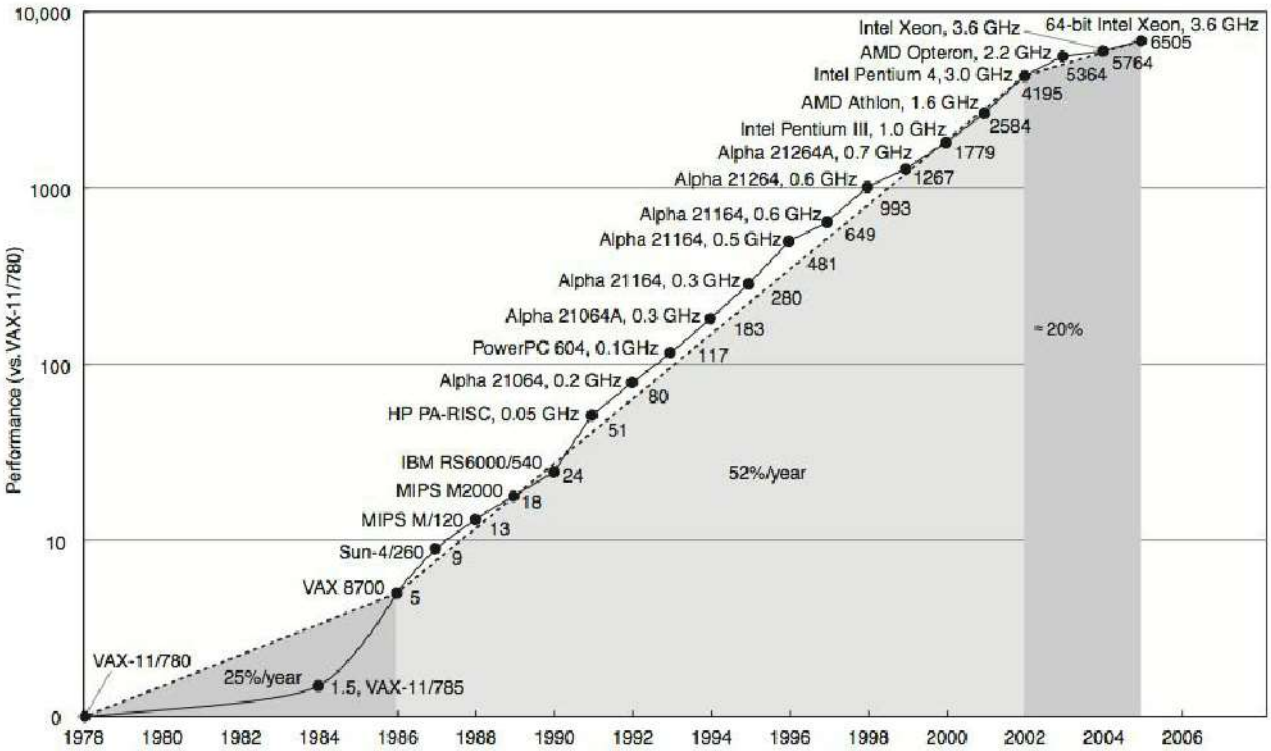


FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

Product	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~ 2.5 GHz ?	4.7 GHz	1.4 GHz
Microprocessor power	120 W	~ 100 W ?	~ 100 W ?	94 W

FIGURE 1.17 Number of cores per chip, clock rate, and power for 2008 multicore microprocessors.

Why has it been so hard for programmers to write explicitly parallel programs? The first reason is that parallel programming is by definition performance programming, which increases the difficulty of programming. Not only does the program need to be correct, solve an important problem, and provide a useful interface to the people or other programs that invoke it, the program must also be fast. Otherwise, if you don't need performance, just write a sequential program.

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to

do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism.

As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *schedule* the subtasks. If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must *balance the load* evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all of the other parts were completed. Thus, care must be taken to *reduce communication and synchronization overhead*. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

To reflect this sea change in the industry, the next five chapters in this edition of the book each have a section on the implications of the parallel revolution to that chapter:

- *Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization.* Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Associativity.* Often parallel programmers start from a working sequential program. A natural question to learn if their parallel version works is, "does it get the same answer?" If not, a logical conclusion is that there are bugs in the new version. This logic assumes that computer arithmetic is associative: you get the same sum when adding a million numbers, no matter what the order. This chapter explains that while this logic holds for integers, it doesn't hold for floating-point numbers.
- *Chapter 4, Section 4.10: Parallelism and Advanced Instruction-Level Parallelism.* Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions simultaneously and guessing on the outcomes of decisions, and executing instructions speculatively.

- *Chapter 5, Section 5.8: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.
- *Chapter 6, Section 6.9: Parallelism and I/O: Redundant Arrays of Inexpensive Disks.* If you ignore input and output in this parallel revolution, the unintended consequence of parallel programming may be to make your parallel program spend most of its time waiting for I/O. This chapter describes RAID, a technique to accelerate the performance of storage accesses. RAID points out another potential benefit of parallelism: by having many copies of resources, the system can continue to provide service despite a failure of one resource. Hence, RAID can improve both I/O performance and availability.

In addition to these sections, there is a full chapter on parallel processing. Chapter 7 goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors and finally describes and evaluates four examples of multicore microprocessors using this model.

I thought [computers] would be a universally applicable idea, like a book is. But I didn't think it would develop as fast as it did, because I didn't envision we'd be able to get as many parts on a chip as we finally got. The transistor came along unexpectedly. It all happened much faster than we expected.

J. Presper Eckert, coinventor of ENIAC, speaking in 1991

Starting with this edition of the book, Appendix A describes an increasingly popular hardware component that is included with desktop computers, the graphics processing unit (GPU). Invented to accelerate graphics, GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs are highly parallel. Appendix A describes the NVIDIA GPU and highlights parts of its parallel programming environment.

1.7 Real Stuff: Manufacturing and Benchmarking the AMD Opteron X4

Each chapter has a section entitled "Real Stuff" that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying modern computers. For this first "Real Stuff" section, we look at how integrated circuits are manufactured and how performance and power are measured, with the AMD Opteron X4 as the example.

Let's start at the beginning. The manufacture of a chip begins with **silicon**, a substance found in sand. Because silicon does not conduct electricity well, it is called a **semiconductor**. With a special chemical process, it is possible to add materials to silicon that allow tiny areas to transform into one of three devices:

- Excellent conductors of electricity (using either microscopic copper or aluminum wire)
- Excellent insulators from electricity (like plastic sheathing or glass)
- Areas that can conduct *or* insulate under special conditions (as a switch)

silicon A natural element that is a semiconductor.

semiconductor A substance that does not conduct electricity well.

Transistors fall in the last category. A VLSI circuit, then, is just billions of combinations of conductors, insulators, and switches manufactured in a single small package.

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.18 shows that process. The process starts with a **silicon crystal ingot**, which looks like a giant sausage. Today, ingots are 8–12 inches in diameter and about 12–24 inches long. An ingot is finely sliced into **wafers** no more than 0.1 inch thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed on

silicon crystal ingot A rod composed of a silicon crystal that is between 8 and 12 inches in diameter and about 12 to 24 inches long.

wafer A slice from a silicon ingot no more than 0.1 inch thick, used to create chips.

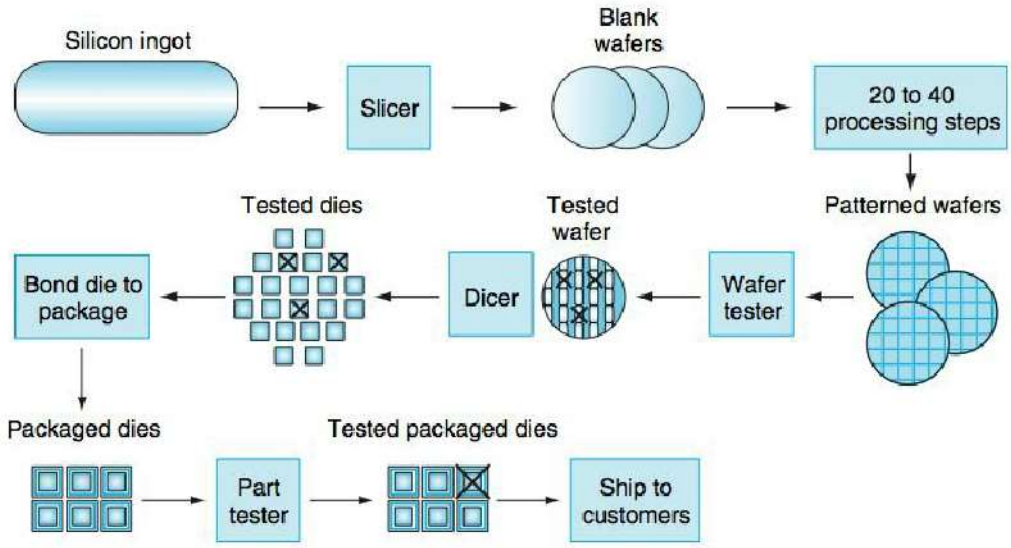


FIGURE 1.18 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.19). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9). In this figure, one wafer produced 20 dies, of which 17 passed testing. (*X* means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

each wafer, creating the transistors, conductors, and insulators discussed earlier. Today's integrated circuits contain only one layer of transistors but may have from two to eight levels of metal conductor, separated by layers of insulators.

defect A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

die The individual rectangular sections that are cut from a wafer, more informally known as **chips**.

yield The percentage of good dies from the total number of dies on the wafer.

A single microscopic flaw in the wafer itself or in one of the dozens of patterning steps can result in that area of the wafer failing. These **defects**, as they are called, make it virtually impossible to manufacture a perfect wafer. To cope with imperfection, several strategies have been used, but the simplest is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced,* into these components, called **dies** and more informally known as **chips**. Figure 1.19 is a photograph of a wafer containing microprocessors before they have been diced; earlier, Figure 1.9 on page 20 shows an individual microprocessor die and its major components.

Dicing enables you to discard only those dies that were unlucky enough to contain the flaws, rather than the whole wafer. This concept is quantified by the **yield** of a process, which is defined as the percentage of good dies from the total number of dies on the wafer.

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of dies that fit on a wafer. To reduce the cost, a large die is often "shrunk" by using the next generation process, which incorporates smaller sizes for both transistors and wires. This improves the yield and the die count per wafer.

Once you've found good dies, they are connected to the input/output pins of a package, using a process called *bonding*. These packaged parts are tested a final time, since mistakes can occur in packaging, and then they are shipped to customers.

As mentioned above, an increasingly important design constraint is power. Power is a challenge for two reasons. First, power must be brought in and distributed around the chip; modern microprocessors use hundreds of pins just for power and ground! Similarly, multiple levels of interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. An AMD Opteron X4 model 2356 2.0 GHz burns 120 watts in 2008, which must be removed from a chip whose surface area is just over 1 cm^2!

Elaboration: The cost of an integrated circuit can be expressed in three simple equations:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

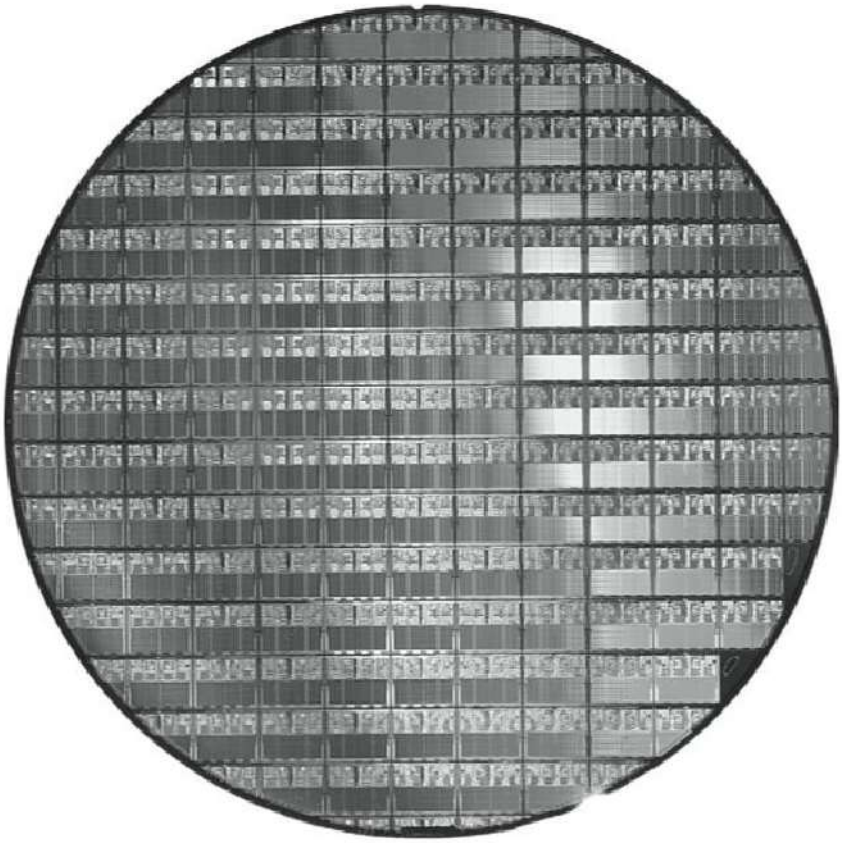


FIGURE 1.19 A 12-inch (300mm) wafer of AMD Opteron X2 chips, the predecessor of Opteron X4 chips (Courtesy AMD). The number of dies per wafer at 100% yield is 117. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the masks used to pattern the silicon. This die uses a 90-nanometer technology, which means that the smallest transistors are approximately 90 nm in size, although they are typically somewhat smaller than the actual feature size, which refers to the size of the transistors as "drawn" versus the final manufactured size.

The first equation is straightforward to derive. The second is an approximation, since it does not subtract the area near the border of the round wafer that cannot accommodate the rectangular dies (see Figure 1.19). The final equation is based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.

Hence, depending on the defect rate and the size of the die and wafer, costs are generally not linear in die area.

SPEC CPU Benchmark

workload A set of programs run on a computer that is either the actual collection of applications run by a user or constructed from real programs to approximate such a mix. A typical workload specifies both the programs and the relative frequencies.

benchmark A program selected for use in comparing computer performance.

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. The set of programs run would form a **workload**. To evaluate two computer systems, a user would simply compare the execution time of the workload on the two computers. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate computer, hoping that the methods will reflect how well the computer will perform with the user's workload. This alternative is usually followed by evaluating the computer using a set of **benchmarks**—programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload.

SPEC (System Performance Evaluation Cooperative) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems. In 1989, SPEC originally created a benchmark set focusing on processor performance (now called SPEC89), which has evolved through five generations. The latest is SPEC CPU2006, which consists of a set of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006). The integer benchmarks vary from part of a C compiler to a chess program to a quantum computer simulation. The floating-point benchmarks include structured grid codes for finite element modeling, particle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics.

Figure 1.20 describes the SPEC integer benchmarks and their execution time on the Opteron X4 and shows the factors that explain execution time: instruction count, CPI, and clock cycle time. Note that CPI varies by a factor of 13.

To simplify the marketing of computers, SPEC decided to report a single number to summarize all 12 integer benchmarks. The execution time measurements are first normalized by dividing the execution time on a reference processor by the execution time on the measured computer; this normalization yields a measure, called the *SPECratio,* which has the advantage that bigger numeric results indicate faster performance (i.e., the SPECratio is the inverse of execution time). A CINT2006 or CFP2006 summary measurement is obtained by taking the geometric mean of the SPECratios.

Elaboration: When comparing two computers using SPECratios, use the geometric mean so that it gives the same relative answer no matter what computer is used to normalize the results. If we averaged the normalized execution time values with an arithmetic mean, the results would vary depending on the computer we choose as the reference.

Description	Name	Instruction Count × 10^9	CPI	Clock cycle time (seconds × 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2,118	0.75	0.4	637	9,770	15.3
Block-sorting compression	bzip2	2,389	0.85	0.4	817	9,650	11.8
GNU C compiler	gcc	1,050	1.72	0.4	724	8,050	11.1
Combinatorial optimization	mcf	336	10.00	0.4	1,345	9,120	6.8
Go game (AI)	go	1,658	1.09	0.4	721	10,490	14.6
Search gene sequence	hmmer	2,783	0.80	0.4	890	9,330	10.5
Chess game (AI)	sjeng	2,176	0.96	0.4	837	12,100	14.5
Quantum computer simulation	libquantum	1,623	1.61	0.4	1,047	20,720	19.8
Video compression	h264avc	3,102	0.80	0.4	993	22,130	22.3
Discrete event simulation library	omnetpp	587	2.94	0.4	690	6,250	9.1
Games/path finding	astar	1,082	1.79	0.4	773	7,020	9.1
XML parsing	xalancbmk	1,058	2.70	0.4	1,143	6,900	6.0
Geometric Mean							11.7

FIGURE 1.20 SPECINTC2006 benchmarks running on AMD Opteron X4 model 2356 (Barcelona). As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios. Figure 5.40 on page 542 shows that mcf, libquantum, omnetpp, and xalancbmk have relatively high CPIs because they have high cache miss rates.

The formula for the geometric mean is

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

where Execution time ratio$_i$ is the execution time, normalized to the reference computer, for the ith program of a total of n in the workload, and

$$\prod_{i=1}^{n} a_i \text{ means the product } a_1 \times a_2 \times \ldots \times a_n$$

SPEC Power Benchmark

Today, SPEC offers a dozen different benchmark sets designed to test a wide variety of computing environments using real applications and strictly specified execution rules and reporting requirements. The most recent is SPECpower. It reports power consumption of servers at different workload levels, divided into 10% increments, over a period of time. Figure 1.21 shows the results for a server using Barcelona.

SPECpower started with the SPEC benchmark for Java business applications (SPECJBB2005), which exercises the processors, caches, and main memory as well as the Java virtual machine, compiler, garbage collector, and pieces of the operating

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	231,867	295
90%	211,282	286
80%	185,803	275
70%	163,427	265
60%	140,160	256
50%	118,324	246
40%	92,035	233
30%	70,500	222
20%	47,126	206
10%	23,066	180
0%	0	141
Overall Sum	1,283,590	2,605
Σssj_ops / Σpower =		493

FIGURE 1.21 SPECpower_ssj2008 running on dual socket 2.3 GHz AMD Opteron X4 2356 (Barcelona) with 16 GB Of DDR2-667 DRAM and one 500 GB disk.

system. Performance is measured in throughput, and the units are business operations per second. Once again, to simplify the marketing of computers, SPEC boils these numbers down to a single number, called "overall ssj_ops per Watt." The formula for this single summarizing metric is

$$\text{overall ssj_ops per Watt} = \left(\sum_{i=0}^{10} \text{ssj_ops}_i\right) / \left(\sum_{i=0}^{10} \text{power}_i\right)$$

where ssj_ops_i is performance at each 10% increment and power_i is power consumed at each performance level.

Check Yourself A key factor in determining the cost of an integrated circuit is volume. Which of the following are reasons why a chip made in high volume should cost less?

1. With high volumes, the manufacturing process can be tuned to a particular design, increasing the yield.
2. It is less work to design a high-volume part than a low-volume part.
3. The masks used to make the chip are expensive, so the cost per chip is lower for higher volumes.
4. Engineering development costs are high and largely independent of volume; thus, the development cost per die is lower with high-volume parts.
5. High-volume parts usually have smaller die sizes than low-volume parts and therefore have higher yield per wafer.

1.8 Fallacies and Pitfalls

Science must begin with myths, and the criticism of myths.

Sir Karl Popper, *The Philosophy of Science*, 1957

The purpose of a section on fallacies and pitfalls, which will be found in every chapter, is to explain some commonly held misconceptions that you might encounter. We call such misbeliefs *fallacies*. When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*, or easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these mistakes in the computers you may design or use. Cost/performance fallacies and pitfalls have ensnared many a computer architect, including us. Accordingly, this section suffers no shortage of relevant examples. We start with a pitfall that traps many designers and reveals an important relationship in computer design.

Pitfall: Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of the improvement.

This pitfall has visited designers of both hardware and software. A simple design problem illustrates it well. Suppose a program runs in 100 seconds on a computer, with multiply operations responsible for 80 seconds of this time. How much do I have to improve the speed of multiplication if I want my program to run five times faster?

The execution time of the program after making the improvement is given by the following simple equation known as **Amdahl's law**:

Amdahl's law A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. It is a quantitative version of the law of diminishing returns.

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

For this problem:

$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$20 \text{ seconds} = \frac{80 \text{ seconds}}{n} + 20 \text{ seconds}$$

$$0 = \frac{80 \text{ seconds}}{n}$$

That is, there is *no amount* by which we can enhance-multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload.

The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. This concept also yields what we call the law of diminishing returns in everyday life.

We can use Amdahl's law to estimate performance improvements when we know the time consumed for some function and its potential speedup. Amdahl's law, together with the CPU performance equation, is a handy tool for evaluating potential enhancements. Amdahl's law is explored in more detail in the exercises.

A common theme in hardware design is a corollary of Amdahl's law: *Make the common case fast.* This simple guideline reminds us that in many cases the frequency with which one event occurs may be much higher than the frequency of another. Amdahl's law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance.

Amdahl's law is also used to argue for practical limits to the number of parallel processors. We examine this argument in the Fallacies and Pitfalls section of Chapter 7.

Fallacy: Computers at low utilization use little power.

Power efficiency matters at low utilizations because server workloads vary. CPU utilization for servers at Google, for example, is between 10% and 50% most of the time and at 100% less than 1% of the time. Figure 1.22 shows power for servers with the best SPECpower results at 100% load, 50% load, 10% load, and idle. Even servers that are only 10% utilized burn about two-thirds of their peak power.

Since servers' workloads vary but use a large fraction of peak power, Luiz Barroso and Urs Hölzle [2007] argue that we should redesign hardware to achieve "energy-proportional computing." If future servers used, say, 10% of peak power at 10% workload, we could reduce the electricity bill of datacenters and become good corporate citizens in an era of increasing concern about CO_2 emissions.

Server Manufacturer	Micro-processor	Total Cores/ Sockets	Clock Rate	Peak Performance (ssj_ops)	100% Load Power	50% Load Power	50% Load/ 100% Power	10% Load Power	10% Load/ 100% Power	Active Idle Power	Active Idle/ 100% Power
HP	Xeon E5440	8/2	3.0 GHz	308,022	269 W	227 W	84%	174 W	65%	160 W	59%
Dell	Xeon E5440	8/2	2.8 GHz	305,413	276 W	230 W	83%	173 W	63%	157 W	57%
Fujitsu Seimens	Xeon X3220	4/1	2.4 GHz	143,742	132 W	110 W	83%	85 W	65%	80 W	60%

FIGURE 1.22 SPECPower results for three servers with the best overall ssj_ops per watt in the fourth quarter of 2007. The overall ssj_ops per watt of the three servers are 698, 682, and 667, respectively. The memory of the top two servers is 16 GB and the bottom is 8 GB.

Pitfall: Using a subset of the performance equation as a performance metric.

We have already shown the fallacy of predicting performance based on simply one of clock rate, instruction count, or CPI. Another common mistake is to use only

two of the three factors to compare performance. Although using two of the three factors may be valid in a limited context, the concept is also easily misused. Indeed, nearly all proposed alternatives to the use of time as the performance metric have led eventually to misleading claims, distorted results, or incorrect interpretations.

One alternative to time is **MIPS (million instructions per second)**. For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

million instructions per second (MIPS) A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6.

Since MIPS is an instruction execution rate, MIPS specifies performance inversely to execution time; faster computers have a higher MIPS rating. The good news about MIPS is that it is easy to understand, and faster computers mean bigger MIPS, which matches intuition.

There are three problems with using MIPS as a measure for comparing computers. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ. Second, MIPS varies between programs on the same computer; thus, a computer cannot have a single MIPS rating. For example, by substituting for execution time, we see the relationship between MIPS, clock rate, and CPI:

$$\text{MIPS} = \frac{\text{Instruction count}}{\dfrac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Recall that CPI varied by 13× for SPEC CPU2006 on Opteron X4, so MIPS does as well. Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance!

Check Yourself

Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

a. Which computer has the higher MIPS rating?

b. Which computer is faster?

Where . . . the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have 1,000 vacuum tubes and perhaps weigh just 1½ tons.

Popular Mechanics, March 1949

1.9 Concluding Remarks

Although it is difficult to predict exactly what level of cost/performance computers will have in the future, it's a safe bet that they will be much better than they are today. To participate in these advances, computer designers and programmers must understand a wider variety of issues.

Both hardware and software designers construct computer systems in hierarchical layers, with each lower layer hiding details from the level above. This principle of abstraction is fundamental to understanding today's computer systems, but it does not mean that designers can limit themselves to knowing a single abstraction. Perhaps the most important example of abstraction is the interface between hardware and low-level software, called the *instruction set architecture*. Maintaining the instruction set architecture as a constant enables many implementations of that architecture—presumably varying in cost and performance—to run identical software. On the downside, the architecture may preclude introducing innovations that require the interface to change.

There is a reliable method of determining and reporting performance by using the execution time of real programs as the metric. This execution time is related to other important measurements we can make by the following equation:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

We will use this equation and its constituent factors many times. Remember, though, that individually the factors do not determine performance: only the product, which equals execution time, is a reliable measure of performance.

The BIG Picture

Execution time is the only valid and unimpeachable measure of performance. Many other metrics have been proposed and found wanting. Sometimes these metrics are flawed from the start by not reflecting execution time; other times a metric that is valid in a limited context is extended and used beyond that context or without the additional clarification needed to make it valid.

The key hardware technology for modern processors is silicon. Equal in importance to an understanding of integrated circuit technology is an understanding of the expected rates of technological change. While silicon fuels the rapid advance of hardware, new ideas in the organization of computers have improved price/performance. Two of the key ideas are exploiting parallelism in the program,

typically today via multiple processors, and exploiting locality of accesses to a memory hierarchy, typically via caches.

Power has replaced die area as the most critical resource of microprocessor design. Conserving power while trying to increase performance has forced the hardware industry to switch to multicore microprocessors, thereby forcing the software industry to switch to programming parallel hardware.

Computer designs have always been measured by cost and performance, as well as other important factors such as power, reliability, cost of ownership, and scalability. Although this chapter has focused on cost, performance, and power, the best designs will strike the appropriate balance for a given market among all the factors.

Road Map for This Book

At the bottom of these abstractions are the five classic components of a computer: datapath, control, memory, input, and output (refer to Figure 1.4). These five components also serve as the framework for the rest of the chapters in this book:

- *Datapath:* Chapters 3, 4, 7, and Appendix A
- *Control:* Chapters 4, 7, and Appendix A
- *Memory:* Chapter 5
- *Input:* Chapter 6
- *Output:* Chapter 6

As mentioned above, Chapter 4 describes how processors exploit implicit parallelism, Chapter 7 describes the explicitly parallel multicore microprocessors that are at the heart of the parallel revolution, and Appendix A describes the highly parallel graphics processor chip. Chapter 5 describes how a memory hierarchy exploits locality. Chapter 2 describes instruction sets—the interface between compilers and the computer—and emphasizes the role of compilers and programming languages in using the features of the instruction set. Appendix B provides a reference for the instruction set of Chapter 2. Chapter 3 describes how computers handle arithmetic data. Appendix C, on the CD, introduces logic design.

An active field of science is like an immense anthill; the individual almost vanishes into the mass of minds tumbling over each other, carrying information from place to place, passing it around at the speed of light.

Lewis Thomas, "Natural Science," in *The Lives of a Cell*, 1974

1.10 Historical Perspective and Further Reading

For each chapter in the text, a section devoted to a historical perspective can be found on the CD that accompanies this book. We may trace the development of an idea through a series of computers or describe some important projects, and we provide references in case you are interested in probing further.

The historical perspective for this chapter provides a background for some of the key ideas presented in this opening chapter. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By understanding the past, you may be better able to understand the forces that will shape computing in the future. Each historical perspectives section on the CD ends with suggestions for further reading, which are also collected separately on the CD under the section "**Further Reading**." The rest of **Section 1.10** is found on the CD.

1.11 Exercises

Contributed by Javier Bruguera of Universidade de Santiago de Compostela

Most of the exercises in this edition are designed so that they feature a qualitative description supported by a table that provides alternative quantitative parameters. These parameters are needed to solve the questions that comprise the exercise. Individual questions can be solved using any or all of the parameters—you decide how many of the parameters should be considered for any given exercise question. For example, it is possible to say "complete Question 4.1.1 using the parameters given in row A of the table." Alternately, instructors can customize these exercises to create novel solutions by replacing the given parameters with your own unique values.

The number of quantitative exercises varies from chapter to chapter and depends largely on the topics covered. More conventional exercises are provided where the quantitative approach does not fit.

The relative time ratings of exercises are shown in square brackets after each exercise number. On average, an exercise rated [10] will take you twice as long as one rated [5]. Sections of the text that should be read before attempting an exercise will be given in angled brackets; for example, <1.3> means you should have read Section 1.3, Under the Covers, to help you solve this exercise.

Exercise 1.1

Find the word or phrase from the list below that best matches the description in the following questions. Use the numbers to the left of words in the answer. Each answer should be used only once.

1.	virtual worlds	**14.**	operating system
2.	desktop computers	**15.**	compiler
3.	servers	**16.**	bit
4.	low-end servers	**17.**	instruction
5.	supercomputers	**18.**	assembly language
6.	terabyte	**19.**	machine language
7.	petabyte	**20.**	C
8.	data centers	**21.**	assembler
9.	embedded computers	**22.**	high-level language
10.	multicore processors	**23.**	system software
11.	VHDL	**24.**	application software
12.	RAM	**25.**	Cobol
13.	CPU	**26.**	Fortran

1.1.1 [2] <1.1> Computer used to run large problems and usually accessed via a network

1.1.2 [2] <1.1> 10^{15} or 2^{50} bytes

1.1.3 [2] <1.1> A class of computers composed of hundred to thousand processors and terabytes of memory and having the highest performance and cost

1.1.4 [2] <1.1> Today's science fiction application that probably will be available in the near future

1.1.5 [2] <1.1> A kind of memory called random access memory

1.1.6 [2] <1.1> Part of a computer called central processor unit

1.1.7 [2] <1.1> Thousands of processors forming a large cluster

1.1.8 [2] <1.1> Microprocessors containing several processors in the same chip

1.1.9 [2] <1.1> Desktop computer without a screen or keyboard usually accessed via a network

1.1.10 [2] <1.1> A computer used to running one predetermined application or collection of software

1.1.11 [2] <1.1> Special language used to describe hardware components

1.1.12 [2] <1.1> Personal computer delivering good performance to single users at low cost

1.1.13 [2] <1.2> Program that translates statements in high-level language to assembly language

1.1.14 [2] <1.2> Program that translates symbolic instructions to binary instructions

1.1.15 [2] <1.2> High-level language for business data processing

1.1.16 [2] <1.2> Binary language that the processor can understand

1.1.17 [2] <1.2> Commands that the processors understand

1.1.18 [2] <1.2> High-level language for scientific computation

1.1.19 [2] <1.2> Symbolic representation of machine instructions

1.1.20 [2] <1.2> Interface between user's program and hardware providing a variety of services and supervision functions

1.1.21 [2] <1.2> Software/programs developed by the users

1.1.22 [2] <1.2> Binary digit (value 0 or 1)

1.1.23 [2] <1.2> Software layer between the application software and the hardware that includes the operating system and the compilers

1.1.24 [2] <1.2> High-level language used to write application and system software

1.1.25 [2] <1.2> Portable language composed of words and algebraic expressions that must be translated into assembly language before run in a computer

1.1.26 [2] <1.2> 10^{12} or 2^{40} bytes

Exercise 1.2

Consider the different configurations shown in the table

	Configuration	Resolution	Main Memory	Ethernet Network
a.	1	640 × 480	2 Gbytes	100 Mbit
	2	1280 × 1024	4 Gbytes	1 Gbit
b.	1	1024 × 768	2 Gbytes	100 Mbit
	2	2560 × 1600	4 Gbytes	1Gbit

1.2.1 [10] <1.3> For a color display using 8 bits for each of the primary colors (red, green, blue) per pixel, what should be the minimum size in bytes of the frame buffer to store a frame?

1.2.2 [5] <1.3> How many frames could it store, assuming the memory contains no other information?

1.2.3 [5] <1.3> If a 256 Kbytes file is sent through the Ethernet connection, how long it would take?

For problems below, use the information about access time for every type of memory in the following table.

	Cache	DRAM	Flash Memory	Magnetic Disk
a.	5 ns	50 ns	5 μs	5 ms
b.	7 ns	70 ns	15 μs	20 ms

1.2.4 [5] <1.3> Find how long it takes to read a file from a DRAM if it takes 2 microseconds from the cache memory.

1.2.5 [5] <1.3> Find how long it takes to read a file from a disk if it takes 2 microseconds from the cache memory.

1.2.6 [5] <1.3> Find how long it takes to read a file from a flash memory if it takes 2 microseconds from the cache memory.

Exercise 1.3

Consider three different processors P1, P2, and P3 executing the same instruction set with the clock rates and CPIs given in the following table.

	Processor	Clock Rate	CPI
a.	P1	3 GHz	1.5
	P2	2.5 GHz	1.0
	P3	4 GHz	2.2
b.	P1	2 GHz	1.2
	P2	3 GHz	0.8
	P3	4 GHz	2.0

1.3.1 [5] <1.4> Which processor has the highest performance expressed in instructions per second?

1.3.2 [10] <1.4> If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.

1.3.3 [10] <1.4> We are trying to reduce the time by 30% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?

For problems below, use the information in the following table.

	Processor	Clock Rate	No. Instructions	Time
a.	P1	3 GHz	20.00E+09	7 s
	P2	2.5 GHz	30.00E+09	10 s
	P3	4 GHz	90.00E+09	9 s
b.	P1	2 GHz	20.00E+09	5 s
	P2	3 GHz	30.00E+09	8 s
	P3	4 GHz	25.00E+09	7 s

1.3.4 [10] <1.4> Find the IPC (instructions per cycle) for each processor.

1.3.5 [5] <1.4> Find the clock rate for P2 that reduces its execution time to that of P1.

1.3.6 [5] <1.4> Find the number of instructions for P2 that reduces its execution time to that of P3.

Exercise 1.4

Consider two different implementations of the same instruction set architecture. There are four classes of instructions, A, B, C, and D. The clock rate and CPI of each implementation are given in the following table.

		Clock Rate	CPI Class A	CPI Class B	CPI Class C	CPI Class D
a.	P1	2.5 GHz	1	2	3	3
	P2	3 GHz	2	2	2	2
b.	P1	2.5 GHz	2	1.5	2	1
	P2	3 GHz	1	2	1	1

1.4.1 [10] <1.4> Given a program with 10^6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which implementation is faster?

1.4.2 [5] <1.4> What is the global CPI for each implementation?

1.4.3 [5] <1.4> Find the clock cycles required in both cases.

The following table shows the number of instructions for a program.

	Arith	Store	Load	Branch	Total
a.	650	100	600	50	1400
b.	750	250	500	500	2000

1.4.4 [5] <1.4> Assuming that arith instructions take 1 cycle, load and store 5 cycles, and branches 2 cycles, what is the execution time of the program in a 2 GHz processor?

1.4.5 [5] <1.4> Find the CPI for the program.

1.4.6 [10] <1.4> If the number of load instructions can be reduced by one half, what is the speedup and the CPI?

Exercise 1.5

Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set. The clock rate and CPI of each class is given below.

		Clock Rate	CPI Class A	CPI Class B	CPI Class C	CPI Class D	CPI Class E
a.	P1	2.0 GHz	1	2	3	4	3
	P2	4.0 GHz	2	2	2	4	4
b.	P1	2.0 GHz	1	1	2	3	2
	P2	3.0 GHz	1	2	3	4	3

1.5.1 [5] <1.4> Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

1.5.2 [10] <1.4> If the number of instructions executed in a certain program is divided equally among the classes of instructions except for class A, which occurs twice as often as each of the others, which computer is faster? How much faster is it?

1.5.3 [10] <1.4> If the number of instructions executed in a certain program is divided equally among the classes of instructions except for class E, which occurs twice as often as each of the others, which computer is faster? How much faster is it?

The table below shows instruction-type breakdown for different programs. Using this data, you will be exploring the performance trade-offs for different changes made to an MIPS processor.

		No. Instructions				
		Compute	Load	Store	Branch	Total
a.	Program1	600	600	200	50	1450
b.	Program 2	900	500	100	200	1700

1.5.4 [5] <1.4> Assuming that computes take 1 cycle, loads and store instructions take 10 cycles, and branches take 3 cycles, find the execution time on a 3 GHz MIPS processor.

1.5.5 [5] <1.4> Assuming that computes take 1 cycle, loads and store instructions take 2 cycles, and branches take 3 cycles, find the execution time on a 3 GHz MIPS processor.

1.5.6 [5] <1.4> Assuming that computes take 1 cycle, loads and store instructions take 2 cycles, and branches take 3 cycles, what is the speedup if the number of compute instruction can be reduced by one-half?

Exercise 1.6

Compilers can have a profound impact on the performance of an application on given a processor. This problem will explore the impact compilers have on execution time.

	Compiler A		Compiler B	
	No. Instructions	Execution Time	No. Instructions	Execution Time
a.	1.00E+09	1.8 s	1.20E+09	1.8 s
b.	1.00E+09	1.1 s	1.20E+09	1.5 s

1.6.1 [5] <1.4> For the same program, two different compilers are used. The table above shows the execution time of the two different compiled programs. Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.

1.6.2 [5] <1.4> Assume the average CPIs found in 1.6.1, but that the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler A's code versus the clock of the processor running compiler B's code?

1.6.3 [5] <1.4> A new compiler is developed that uses only 600 million instructions and has an average CPI of 1.1. What is the speedup of using this new compiler versus using Compiler A or B on the original processor of 1.6.1?

Consider two different implementations, P1 and P2, of the same instruction set. There are five classes of instructions (A, B, C, D, and E) in the instruction set. P1 has a clock rate of 4 GHz, and P2 has a clock rate of 6 GHz. The average number of cycles for each instruction class for P1 and P2 are listed in the following table.

		CPI Class A	CPI Class B	CPI Class C	CPI Class D	CPI Class E
a.	P1	1	2	3	4	5
	P2	3	3	3	5	5
b.	P1	1	2	3	4	5
	P2	2	2	2	2	6

1.6.4 [5] <1.4> Assume that peak performance is defined as the fastest rate that a computer can execute any instruction sequence. What are the peak performances of P1 and P2 expressed in instructions per second?

1.6.5 [5] <1.4> If the number of instructions executed in a certain program is divided equally among the five classes of instructions except for class A, which occurs twice as often as each of the others, how much faster is P2 than P1?

1.6.6 [5] <1.4> At what frequency does P1 have the same performance of P2 for the instruction mix given in 1.6.5?

Exercise 1.7

The following table shows the increase in clock rate and power of eight generations of Intel processors over 28 years.

Processor	Clock Rate	Power
80286 (1982)	12.5 MHz	3.3 W
80386 (1985)	16 MHz	4.1 W
80486 (1989)	25 MHz	4.9 W
Pentium (1993)	66 MHz	10.1 W
Pentium Pro (1997)	200 MHz	29.1 W
Pentium 4 Willamette (2001)	2 GHz	75.3 W
Pentium 4 Prescott (2004)	3.6 GHz	103 W
Core 2 Ketsfield (2007)	2.667 GHz	95 W

1.7.1 [5] <1.5> What is the geometric mean of the ratios between consecutive generations for both clock rate and power? (The geometric mean is described in Section 1.7.)

1.7.2 [5] <1.5> What is the largest relative change in clock rate and power between generations?

1.7.3 [5] <1.5> How much larger is the clock rate and power of the last generation with respect to the first generation?

Consider the following values for voltage in each generation.

Processor	Voltage
80286 (1982)	5
80386 (1985)	5
80486 (1989)	5
Pentium (1993)	5
Pentium Pro (1997)	3.3
Pentium 4 Willamette (2001)	1.75
Pentium 4 Prescott (2004)	1.25
Core 2 Ketsfield (2007)	1.1

1.7.4 [5] <1.5> Find the average capacitive loads, assuming a negligible static power consumption.

1.7.5 [5] <1.5> Find the largest relative change in voltage between generations.

1.7.6 [5] <1.5> Find the geometric mean of the voltage ratios in the generations since the Pentium.

Exercise 1.8

Suppose we have developed new versions of a processor with the following characteristics.

	Version	Voltage	Clock Rate
a.	Version 1	1.75 V	1.5 GHz
	Version 2	1.2 V	2 GHz
b.	Version 1	1.1 V	3 GHz
	Version 2	0.8 V	4 GHz

1.8.1 [5] <1.5> How much has the capacitive load varied between versions if the dynamic power has been reduced by 10%?

1.8.2 [5] <1.5> How much has the dynamic power been reduced if the capacitive load does not change?

1.8.3 [10] <1.5> Assuming that the capacitive load of version 2 is 80% the capacitive load of version 1, find the voltage for version 2 if the dynamic power of version 2 is reduced by 40% from version 1.

Suppose that the industry trends show that a new process generation varies as follows.

	Capacitance	Voltage	Clock Rate	Area
a.	1	$1/2^{1/2}$	1.15	$1/2^{1/2}$
b.	1	$1/2^{1/4}$	1.2	$1/2^{1/4}$

1.8.4 [5] <1.5> Find the scaling factor for the dynamic power.

1.8.5 [5] <1.5> Find the scaling of the capacitance per unit area unit.

1.8.6 [5] <1.5> Assuming a Core 2 processor with a clock rate of 2.667 GHz, a power consumption of 95 W, and a voltage of 1.1 V, find the voltage and clock rate of this processor for the next process generation.

Exercise 1.9

Although the dynamic power is the primary source of power dissipation in CMOS, leakage current produces a static power dissipation $V \times I_{leak}$. The smaller the on-chip dimensions, the more significant is the static power. Assume the figures shown in the following table for static and dynamic power dissipation for several generations of processors.

	Technology	Dynamic Power (W)	Static Power (W)	Voltage (V)
a.	180 nm	50	10	1.2
b.	70 nm	90	60	0.9

1.9.1 [5] <1.5> Find the percentage of the total dissipated power comprised by static power.

1.9.2 [5] <1.5> If the total dissipated power is reduced by 10% while maintaining the static to total power rate of problem 1.9.1, how much should the voltage be reduced to maintain the same leakage current?

1.9.3 [5] <1.5> Determine the ratio of static power to dynamic power for each technology.

Consider now the dynamic power dissipation of different versions of a given processor for three different voltages given in the following table.

	1.2 V	1.0 V	0.8 V
a.	75 W	60 W	35 W
b.	62 W	50 W	30 W

1.9.4 [5] <1.5> Determine the static power at 0.8 V, assuming a static to dynamic power ratio of 0.6.

1.9.5 [5] <1.5> Determine the static and dynamic power dissipation assuming the rates obtained in problem 1.9.1.

1.9.6 [10] <1.5> Determine the geometric mean of the power variations between versions.

Exercise 1.10

The table below shows the instruction type breakdown of a given application executed on 1, 2, 4, or 8 processors. Using this data, you will be exploring the speed-up of applications on parallel processors.

	Processors	No. Instructions per Processor			CPI		
		Arithmetic	Load/Store	Branch	Arithmetic	Load/Store	Branch
a.	1	2560	1280	256	1	4	2
	2	1280	640	128	1	5	2
	4	640	320	64	1	7	2
	8	320	160	32	1	12	2

	Processors	No. Instructions per Processor			CPI		
		Arithmetic	Load/Store	Branch	Arithmetic	Load/Store	Branch
b.	1	2560	1280	256	1	4	2
	2	1280	640	128	1	6	2
	4	640	320	64	1	8	2
	8	320	160	32	1	10	2

1.10.1 [5] <1.4, 1.6> The table above shows the number of instructions required per processor to complete a program on a multiprocessor with 1, 2, 4, or 8 processors. What is the total number of instructions executed per processor? What is the aggregate number of instructions executed across all processors?

1.10.2 [5] <1.4, 1.6> Given the CPI values on the right of the table above, find the total execution time for this program on 1, 2, 4, and 8 processors. Assume that each processor has a 2 GHz clock frequency.

1.10.3 [10] <1.4, 1.6> If the CPI of the arithmetic instructions was doubled, what would the impact be on the execution time of the program on 1, 2, 4, or 8 processors?

The table below shows the number of instructions per processor core on a multicore processor as well as the average CPI for executing the program on 1, 2, 4, or 8 cores. Using this data, you will be exploring the speedup of applications on multicore processors.

	Cores per Processor	Instructions per Core	Average CPI
a.	1	1.00E+10	1.2
	2	5.00E+09	1.4
	4	2.50E+09	1.8
	8	1.25E+09	2.6

	Cores per Processor	Instructions per Core	Average CPI
b.	1	1.00E+10	1.0
	2	5.00E+09	1.2
	4	2.50E+09	1.4
	8	1.25E+09	1.7

1.10.4 [10] <1.4, 1.6> Assuming a 3 GHz clock frequency, what is the execution time of the program using 1, 2, 4, or 8 cores?

1.10.5 [10] <1.5, 1.6> Assume that the power consumption of a processor core can be described by the following equation:

$$\text{Power} = \frac{5.0\text{mA}}{\text{MHz}} \text{Voltage}^2$$

where the operation voltage of the processor is described by the following equation:

$$\text{Voltage} = \frac{1}{5}\text{Frequency} + 0.4$$

with the frequency measured in GHz. So, at 5 GHz, the voltage would be 1.4 V. Find the power consumption of the program executing on 1, 2, 4, and 8 cores assuming that each core is operating at a 3 GHz clock frequency. Likewise, find the power consumption of the program executing on 1, 2, 4, or 8 cores assuming that each core is operating at 500 MHz.

1.10.6 [10] <1.5, 1.6> If using a single core, find the required CPI for this core to get an execution time equal to the time obtained by using the number of cores in the table above (execution times in problem 1.10.4). Note that the number of instructions should be the aggregate number of instructions executed across all the cores.

Exercise 1.11

The following table shows manufacturing data for various processors.

	Wafer Diameter	Dies per Wafer	Defects per Unit Area	Cost per Wafer
a.	15 cm	84	0.020 defects/cm^2	12
b.	20 cm	100	0.031 defects/cm^2	15

1.11.1 [10] <1.7> Find the yield.

1.11.2 [5] <1.7> Find the cost per die.

1.11.3 [10] <1.7> If the number of dies per wafer is increased by 10% and the defects per area unit increases by 15%, find the die area and yield.

Suppose that, with the evolution of the electronic devices manufacturing technology, the yield varies as shown in the following table.

	T1	T2	T3	T4
Yield	0.85	0.89	0.92	0.95

1.11.4 [10] <1.7> Find the defects per area unit for each technology given a die area of 200 mm^2.

1.11.5 [5] <1.7> Represent graphically the variation of the yield together with the variation of defects per unit area.

Exercise 1.12

The following table shows results for SPEC CPU2006 benchmark programs running on an AMD Barcelona.

	Name	Intr. Count × 10^9	Execution Time (seconds)	Reference Time (seconds)
a.	bzip2	2389	750	9650
b.	go	1658	700	10,490

1.12.1 [5] <1.7> Find the CPI if the clock cycle time is 0.333 ns.

1.12.2 [5] <1.7> Find the SPECratio.

1.12.3 [5] <1.7> For these two benchmarks, find the geometric mean of the SPECratio.

The following table shows data for further benchmarks.

	Name	CPI	Clock Rate	SPECratio
a.	libquantum	1.61	4 GHz	19.8
b.	astar	1.79	4 GHz	9.1

1.12.4 [5] <1.7> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% without affecting the CPI.

1.12.5 [5] <1.7> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% and the CPI is increased by 5%.

1.12.6 [5] <1.7> Find the change in the SPECratio for the change described in 1.12.5.

Exercise 1.13

Suppose that we are developing a new version of the AMD Barcelona processor with a 4 GHz clock rate. We have added some additional instructions to the instruction set in such a way that the number of instructions has been reduced by 15% from the values shown for each benchmark in Exercise 1.12. The execution times obtained are shown in the following table.

	Name	Execution Time (seconds)	Reference Time (seconds)	SPECratio
a.	bzip2	700	9650	13.7
b.	go	620	10490	16.9

1.13.1 [10] <1.8> Find the new CPI.

1.13.2 [10] <1.8> In general, these CPI values are larger than those obtained in previous exercises for the same benchmarks. This is due mainly to the clock rate used in both cases, 3 GHz and 4 GHz. Determine whether the increase in the CPI is similar to that of the clock rate. If they are dissimilar, why?

1.13.3 [5] <1.8> How much has the CPU time been reduced?

The following table shows data for further benchmarks.

	Name	Execution Time (seconds)	CPI	Clock Rate
a.	libquantum	960	1.61	3 GHz
b.	astar	690	1.79	3 GHz

1.13.4 [10] <1.8> If the execution time is reduced by an additional 10% without affecting to the CPI and with a clock rate of 4 GHz, determine the number of instructions.

1.13.5 [10] <1.8> Determine the clock rate required to give a further 10% reduction in CPU time while maintaining the number of instructions and with the CPI unchanged.

1.13.6 [10] <1.8> Determine the clock rate if the CPI is reduced by 15% and the CPU time by 20% while the number of instructions is unchanged.

Exercise 1.14

Section 1.8 cites as a pitfall the utilization of a subset of the performance equation as a performance metric. To illustrate this, consider the following data for the execution of a program in different processors.

	Processor	Clock Rate	CPI	No. Instr.
a.	P1	4 GHz	0.9	5.00E+06
	P2	3 GHz	0.75	1.00E+06
b.	P1	3 GHz	1.1	3.00E+06
	P2	2.5 GHz	1.0	0.50E+06

1.14.1 [5] <1.8> One usual fallacy is to consider the computer with the largest clock rate as having the largest performance. Check if this is true for P1 and P2.

1.14.2 [10] <1.8> Another fallacy is to consider that the processor executing the largest number of instructions will need a larger CPU time. Considering that processor P1 is executing a sequence of 10^6 instructions and that the CPI of processors P1 and P2 do not change, determine the number of instructions that P2 can execute in the same time that P1 needs to execute 10^6 instructions.

1.14.3 [10] <1.8> A common fallacy is to use MIPS (millions of instructions per second) to compare the performance of two different processors, and consider that the processor with the largest MIPS has the largest performance. Check if this is true for P1 and P2.

Another common performance figure is MFLOPS (million of floating-point operations per second), defined as

MFLOPS = No. FP operations / (execution time × 10^6)

but this figure has the same problems as MIPS. Consider the program in the following table, running on the two processors below.

	Processor	Instr. Count	No. Instructions			CPI			Clock Rate
			L/S	FP	Branch	L/S	FP	Branch	
a.	P1	1.00E+06	50%	40%	10%	0.75	1.0	1.5	4 GHz
	P2	5.00E+06	40%	40%	20%	1.25	0.8	1.25	3 GHz
b.	P1	5.00E+06	30%	30%	40%	1.5	1.0	2.0	4 GHz
	P2	2.00E+06	40%	30%	30%	1.25	1.0	2.5	3 GHz

1.14.4 [10] <1.8> Find the MFLOPS figures for the programs.

1.14.5 [10] <1.8> Find the MIPS figures for the programs.

1.14.6 [10] <1.8> Find the performance for the programs and compare it with MIPS and MFLOPS.

Exercise 1.15

Another pitfall cited in Section 1.8 is expecting to improve the overall performance of a computer by improving only one aspect of the computer. This might be true, but not always. Consider a computer running programs with CPU times shown in the following table.

	FP Instr.	INT Instr.	L/S Instr.	Branch Instr.	Total Time
a.	70 s	85 s	55 s	40 s	250 s
b.	40 s	90 s	60 s	20 s	210 s

1.15.1 [5] <1.8> How much is the total time reduced if the time for FP operations is reduced by 20%?

1.15.2 [5] <1.8> How much is the time for INT operations reduced if the total time is reduced by 20%?

1.15.3 [5] <1.8> Can the total time can be reduced by 20% by reducing only the time for branch instructions?

The following table shows the instruction type breakdown per processor of given applications executed in different numbers of processors.

	Processors	FP Instr.	INT Instr.	L/S Instr.	Branch Instr.	CPI (FP)	CPI (INT)	CPI (L/S)	CPI (Branch)
a.	2	280×10^6	1000×16^6	640×10^6	128×10^6	1	1	4	2
b.	16	50×10^6	110×10^6	80×10^6	16×10^6	1	1	4	2

Assume that each processor has a 2 GHz clock rate.

1.15.4 [10] <1.8> How much must we improve the CPI of FP instructions if we want the program to run two times faster?

1.15.5 [10] <1.8> How much must we improve the CPI of L/S instructions if we want the program to run two times faster?

1.15.6 [5] <1.8> How much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

Exercise 1.16

Another pitfall, related to the execution of programs in multiprocessor systems, is expecting improvement in performance by improving only the execution time of part of the routines. The following table shows the execution time of five routines of a program running on different numbers of processors.

	No. Processors	Routine A (ms)	Routine B (ms)	Routine C (ms)	Routine D (ms)	Routine E (ms)
a.	4	12	45	6	36	3
b.	32	2	7	1	6	2

1.16.1 [10] <1.8> Find the total execution time and by how much it is reduced if the time of routines A, C, and E is improved by 15%.

1.16.2 [10] <1.8> How much is the total time reduced if routine B is improved by 10%?

1.16.3 [10] <1.8> How much is the total time reduced if routine D is improved by 10%?

Execution time in a multiprocessor system can be split into computing time for the routines plus routing time spent sending data from one processor to another. Consider the execution time and routing time given in the following table. In this case, the routing time is an important component of the total time.

No. Processors	Routine A (ms)	Routine B (ms)	Routine C (ms)	Routine D (ms)	Routine E (ms)	Routing Time (ms)
2	40	78	9	70	4	11
4	29	60	4	36	2	13
8	15	45	3	19	3	17
16	7	35	1	11	2	22
32	4	23	1	6	1	23
64	2	12	0.5	3	1	26

1.16.4 [10] <1.8> For each doubling of the number of processors, determine the ratio of new to old computing time and the ratio of new to old routing time.

1.16.5 [5] <1.8> Using the geometric means of the ratios, extrapolate to find the computing time and routing time in a 128-processor system.

1.16.6 [10] <1.8> Find the computing time and routing time for a system with one processor.

Answers to Check Yourself

§1.1, page 9: Discussion questions: many answers are acceptable.
§1.3, page 25: Disk memory: nonvolatile, long access time (milliseconds), and cost \$0.20–\$2.00/GB. Semiconductor memory: volatile, short access time (nanoseconds), and cost \$20–\$75/GB.
§1.4, page 31: 1. a: both, b: latency, c: neither. 2. 7 seconds.
§1.4, page 38: b.
§1.7, page 50: 1, 3, and 4 are valid reasons. Answer 5 can be generally true because high volume can make the extra investment to reduce die size by, say, 10% a good economic decision, but it doesn't have to be true.
§1.8, page 53: a. Computer A has the higher MIPS rating. b. Computer B is faster.

2

Instructions: Language of the Computer

I speak Spanish to God, Italian to women, French to men, and German to my horse.

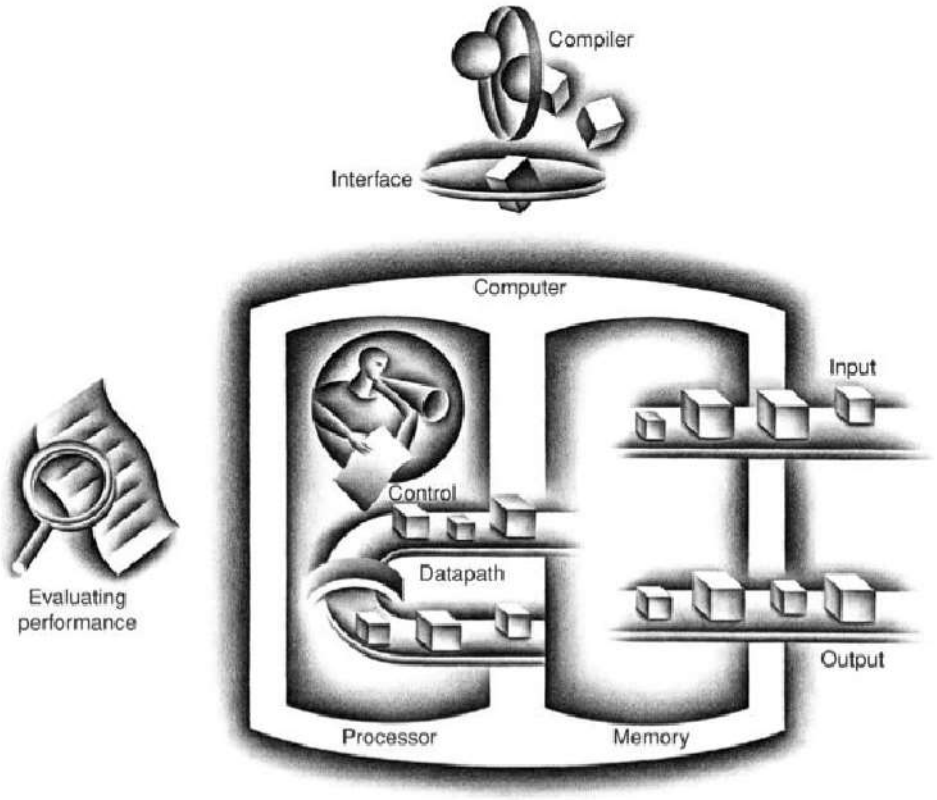
Charles V, Holy Roman Emperor
(1500–1558)

2.1 Introduction 76
2.2 Operations of the Computer Hardware 77
2.3 Operands of the Computer Hardware 80
2.4 Signed and Unsigned Numbers 87
2.5 Representing Instructions in the Computer 94
2.6 Logical Operations 102
2.7 Instructions for Making Decisions 105

Computer Organization and Design. DOI: 10.1016/B978-0-12-374750-1.00002-5
© 2012 Elsevier, Inc. All rights reserved.

2.8 **Supporting Procedures in Computer Hardware** 112
2.9 **Communicating with People** 122
2.10 **MIPS Addressing for 32-Bit Immediates and Addresses** 128
2.11 **Parallelism and Instructions: Synchronization** 137
2.12 **Translating and Starting a Program** 139
2.13 **A C Sort Example to Put It All Together** 149
2.14 **Arrays versus Pointers** 157
2.15 **Advanced Material: Compiling C and Interpreting Java** 161
2.16 **Real Stuff: ARM Instructions** 161
2.17 **Real Stuff: x86 Instructions** 165
2.18 **Fallacies and Pitfalls** 174
2.19 **Concluding Remarks** 176
2.20 **Historical Perspective and Further Reading** 179
2.21 **Exercises** 179

The Five Classic Components of a Computer



2.1 Introduction

instruction set The vocabulary of commands understood by a given architecture.

To command a computer's hardware, you must speak its language. The words of a computer's language are called *instructions*, and its vocabulary is called an **instruction set**. In this chapter, you will see the instruction set of a real computer, both in the form written by people and in the form read by the computer. We introduce instructions in a top-down fashion. Starting from a notation that looks like a restricted programming language, we refine it step-by-step until you see the real language of a real computer. Chapter 3 continues our downward descent, unveiling the hardware for arithmetic and the representation of floating-point numbers.

You might think that the languages of computers would be as diverse as those of people, but in reality computer languages are quite similar, more like regional dialects than like independent languages. Hence, once you learn one, it is easy to pick up others. This similarity occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all computers must provide. Moreover, computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and power. This goal is time honored; the following quote was written before you could buy a computer, and it is as true today as it was in 1947:

> *It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations. . . . The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.*
>
> Burks, Goldstine, and von Neumann, 1947

The "simplicity of the equipment" is as valuable a consideration for today's computers as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in hardware and the relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language; **Section 2.15** on the CD shows how these would change for an object-oriented language like Java.

By learning how to represent instructions, you will also discover the secret of computing: the **stored-program concept**. Moreover, you will exercise your "foreign language" skills by writing programs in the language of the computer and running them on the simulator that comes with this book. You will also see the impact of programming languages and compiler optimization on performance. We conclude with a look at the historical evolution of instruction sets and an overview of other computer dialects.

stored-program concept The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored-program computer.

The chosen instruction set comes from MIPS Technologies, which is an elegant example of the instruction sets designed since the 1980s. Later, we will take a quick look at two other popular instruction sets. ARM is quite similar to MIPS, and more than three billion ARM processors were shipped in embedded devices in 2008. The other example, the Intel x86, is inside almost all of the 330 million PCs made in 2008.

We reveal the MIPS instruction set a piece at a time, giving the rationale along with the computer structures. This top-down, step-by-step tutorial weaves the components with their explanations, making the computer's language more palatable. Figure 2.1 gives a sneak preview of the instruction set covered in this chapter.

2.2 Operations of the Computer Hardware

There must certainly be instructions for performing the fundamental arithmetic operations.

Burks, Goldstine, and von Neumann, 1947

Every computer must be able to perform arithmetic. The MIPS assembly language notation

```
add a, b, c
```

instructs a computer to add the two variables b and c and to put their sum in a.

This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables b, c, d, and e into variable a. (In this section we are being deliberately vague about what a "variable" is; in the next section we'll explain in detail.)

The following sequence of instructions adds the four variables:

```
add a, b, c      # The sum of b and c is placed in a.
add a, a, d      # The sum of b, c, and d is now in a.
add a, a, e      # The sum of b, c, d, and e is now in a.
```

Thus, it takes three instructions to sum the four variables.

The words to the right of the sharp symbol (#) on each line above are *comments* for the human reader, and the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.

MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register $zero always equals 0, and register $at is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 2.1 MIPS assembly language revealed in this chapter. This information is also found in Column 1 of the MIPS Reference Data Card at the front of this book.

The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum. Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number. This situation illustrates the first of four underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

We can now show, in the two examples that follow, the relationship of programs written in higher-level programming languages to programs in this more primitive notation.

Compiling Two C Assignment Statements into MIPS

EXAMPLE

This segment of a C program contains the five variables `a`, `b`, `c`, `d`, and `e`. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;
d = a - e;
```

The translation from C to MIPS assembly language instructions is performed by the *compiler*. Show the MIPS code produced by a compiler.

ANSWER

A MIPS instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two MIPS assembly language instructions:

```
add a, b, c
sub d, a, e
```

Compiling a Complex C Assignment into MIPS

EXAMPLE

A somewhat complex statement contains the five variables `f`, `g`, `h`, `i`, and `j`:

```
f = (g + h) - (i + j);
```

What might a C compiler produce?

ANSWER

The compiler must break this statement into several assembly instructions, since only one operation is performed per MIPS instruction. The first MIPS instruction calculates the sum of `g` and `h`. We must place the result somewhere, so the compiler creates a temporary variable, called `t0`:

```
add t0,g,h # temporary variable t0 contains g + h
```

Although the next operation is subtract, we need to calculate the sum of `i` and `j` before we can subtract. Thus, the second instruction places the sum of `i` and `j` in another temporary variable created by the compiler, called `t1`:

```
add t1,i,j  # temporary variable t1 contains i + j
```

Finally, the subtract instruction subtracts the second sum from the first and places the difference in the variable `f`, completing the compiled code:

```
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

Check Yourself For a given function, which programming language likely takes the most lines of code? Put the three representations below in order.

1. Java
2. C
3. MIPS assembly language

Elaboration: To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called *Java bytecodes* (see Section 2.15 on the CD), which is quite different from the MIPS instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like MIPS. Because this compilation is normally done much later than for C programs, such Java compilers are often called *Just In Time* (JIT) compilers. Section 2.12 shows how JITs are used later than C compilers in the start-up process, and Section 2.13 shows the performance consequences of compiling versus interpreting Java programs.

2.3 Operands of the Computer Hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called *registers*. Registers are primitives used in hardware design that

are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name **word** in the MIPS architecture.

word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS. (See Section 2.20 on the CD for the history of the number of registers.) Thus, continuing in our top-down, stepwise evolution of the symbolic representation of the MIPS language, in this section we have added the restriction that the three operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-bit registers.

The reason for the limit of 32 registers may be found in the second of our four underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Guidelines such as "smaller is faster" are not absolutes; 31 registers may not be faster than 32. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format, as Section 2.5 demonstrates.

Chapter 4 shows the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is critical to program performance.

Although we could simply write instructions using numbers for registers, from 0 to 31, the MIPS convention is to use two-character names following a dollar sign to represent a register. Section 2.8 will explain the reasons behind these names. For now, we will use `$s0`, `$s1`, ... for registers that correspond to variables in C and Java programs and `$t0`, `$t1`, ... for temporary registers needed to compile the program into MIPS instructions.

EXAMPLE

Compiling a C Assignment Using Registers

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS code?

ANSWER

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, `$t0` and `$t1`, which correspond to the temporary variables above:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Memory Operands

Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These complex data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access such large structures?

Recall the five components of a computer introduced in Chapter 1 and repeated on page 75. The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

data transfer instruction A command that moves data between memory and registers.

address A value used to delineate the location of a specific data element within a memory array.

As explained above, arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure 2.2, the address of the third data element is 2, and the value of Memory[2] is 10.

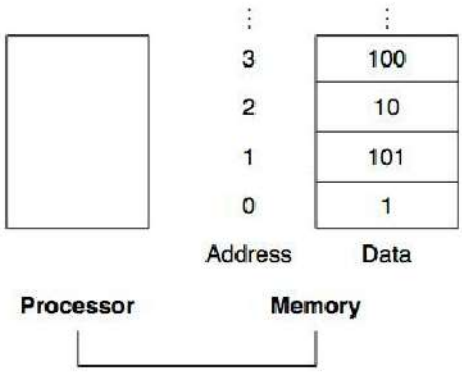


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.

The data transfer instruction that copies data from memory to a register is traditionally called *load*. The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The actual MIPS name for this instruction is `lw`, standing for *load word*.

Compiling an Assignment When an Operand Is in Memory

EXAMPLE

Let's assume that `A` is an array of 100 words and that the compiler has associated the variables `g` and `h` with the registers `$s1` and `$s2` as before. Let's also assume that the starting address, or *base address*, of the array is in `$s3`. Compile this C assignment statement:

```
g = h + A[8];
```

ANSWER

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer `A[8]` to a register. The address of this array element is the sum of the base of the array `A`, found in register `$s3`, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. Based on Figure 2.2, the first compiled instruction is

```
lw   $t0,8($s3) # Temporary reg $t0 gets A[8]
```

(On the next page we'll make a slight adjustment to this instruction, but we'll use this simplified version for now.) The following instruction can operate on the value in `$t0` (which equals `A[8]`) since it is in a register. The instruction must add `h` (contained in `$s2`) to `A[8]` (`$t0`) and put the sum in the register corresponding to `g` (associated with `$s1`):

```
add  $s1,$s2,$t0 # g = h + A[8]
```

The constant in a data transfer instruction (8) is called the *offset*, and the register added to form the address (`$s3`) is called the *base register*.

Hardware/ Software Interface

In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

Since 8-bit *bytes* are useful in many programs, most architectures address individual bytes. Therefore, the address of a word matches the address of one of the 4 bytes within the word, and addresses of sequential words differ by 4. For example, Figure 2.3 shows the actual MIPS addresses for the words in Figure 2.2; the byte address of the third word is 8.

alignment restriction A requirement that data be aligned in memory on natural boundaries.

In MIPS, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**, and many architectures have it. (Chapter 4 suggests why alignment leads to faster data transfers.)

Computers divide into those that use the address of the leftmost or "big end" byte as the word address versus those that use the rightmost or "little end" byte. MIPS is in the *big-endian* camp. (Appendix B, shows the two options to number bytes in a word.)

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register* `$s3` *must be* 4×8*, or 32,* so that the load address will select `A[8]` and not `A[8/4]`. (See the related pitfall on page 175 of Section 2.18.)

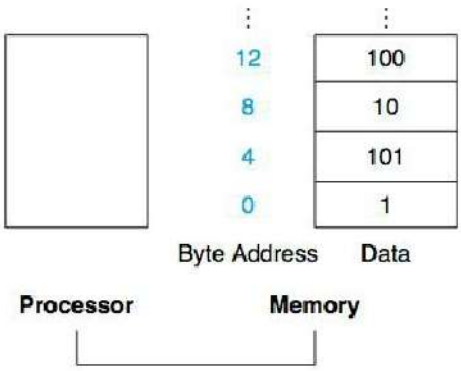


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

The instruction complementary to load is traditionally called *store;* it copies data from a register to memory. The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then offset to select the array element, and finally the base register. Once again, the MIPS address is specified in part by a constant and in part by the contents of a register. The actual MIPS name is `sw`, standing for *store word.*

Compiling Using Load and Store

EXAMPLE

Assume variable `h` is associated with register `$s2` and the base address of the array `A` is in `$s3`. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

ANSWER

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select `A[8]`, and the add instruction places the sum in `$t0`:

```
lw   $t0,32($s3)  # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into `A[12]`, using 48 (4×12) as the offset and register `$s3` as the base register.

```
sw   $t0,48($s3)  # Stores h + A[8] back into A[12]
```

Load word and store word are the instructions that copy words between memory and registers in the MIPS architecture. Other brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in Section 2.17.

Hardware/ Software Interface

Many programs have more variables than computers have registers. Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory. The process of putting less commonly used variables (or those needed later) into memory is called *spilling* registers.

The hardware principle relating size and speed suggests that memory must be slower than registers, since there are fewer registers. This is indeed the case; data accesses are faster if data is in registers instead of memory.

Moreover, data is more useful when in a register. A MIPS arithmetic instruction can read two registers, operate on them, and write the result. A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it.

Thus, registers take less time to access *and* have higher throughput than memory, making data in registers both faster to access and simpler to use. Accessing registers also uses less energy than accessing memory. To achieve highest performance and conserve energy, compilers must use registers efficiently.

Constant or Immediate Operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the MIPS arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks.

Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register `$s3`, we could use the code

```
lw $t0, AddrConstant4($s1)   # $t0 = constant 4
add $s3,$s3,$t0              # $s3 = $s3 + $t0 ($t0 == 4)
```

assuming that `$s1` + `AddrConstant4` is the memory address of the constant 4.

An alternative that avoids the load instruction is to offer versions of the arithmetic instructions in which one operand is a constant. This quick add instruction with one constant operand is called *add immediate* or `addi`. To add 4 to register `$s3`, we just write

```
addi    $s3,$s3,4            # $s3 = $s3 + 4
```

Immediate instructions illustrate the third hardware design principle, first mentioned in the Fallacies and Pitfalls of Chapter 1:

Design Principle 3: Make the common case fast.

Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, the move operation is just an add instruction where one operand is zero. Hence, MIPS dedicates a register `$zero` to be hardwired to the value zero. (As you might expect, it is register number 0.)

Check Yourself

Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

1. Very fast: They increase as fast as Moore's law, which predicts doubling the number of transistors on a chip every 18 months.
2. Very slow: Since programs are usually distributed in the language of the computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable.

Elaboration: Although the MIPS registers in this book are 32 bits wide, there is a 64-bit version of the MIPS instruction set with 32 64-bit registers. To keep them straight, they are officially called MIPS-32 and MIPS-64. In this chapter, we use a subset of MIPS-32. Appendix E shows the differences between MIPS-32 and MIPS-64.

The MIPS offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in Section 2.13.

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the *index register*. Today's memories are much larger and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

Since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

2.4 Signed and Unsigned Numbers

First, let's quickly review how a computer represents numbers. Humans are taught to think in base 10, but numbers may be represented in any base. For example, 123 base 10 = 1111011 base 2.

Numbers are kept in computer hardware as a series of high and low electronic signals, and so they are considered base 2 numbers. (Just as base 10 numbers are called *decimal* numbers, base 2 numbers are called *binary* numbers.)

A single digit of a binary number is thus the "atom" of computing, since all information is composed of **binary digits** or *bits*. This fundamental building block

binary digit Also called **binary bit**. One of the two numbers in base 2, 0 or 1, that are the components of information.

can be one of two values, which can be thought of as several alternatives: high or low, on or off, true or false, or 1 or 0.

Generalizing the point, in any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^i$$

where *i* starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*. For example,

$$1011_{two}$$

represents

$$\begin{aligned}
&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten} \\
&= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten} \\
&= 8 + 0 + 2 + 1_{ten} \\
&= 11_{ten}
\end{aligned}$$

We number the bits 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two}:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 0 1 1

(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase **least significant bit** is used to refer to the rightmost bit (bit 0 above) and **most significant bit** to the leftmost bit (bit 31).

least significant bit The rightmost bit in a MIPS word.

most significant bit The leftmost bit in a MIPS word.

The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4{,}294{,}967{,}295_{ten}$):

$$\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten} \\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten} \\
&\ldots \qquad\qquad\qquad\qquad \ldots \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten} \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten} \\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}
\end{aligned}$$

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here *xi* means the *i*th bit of *x*):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude.*

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude representation was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$$\begin{array}{ll}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} & = 0_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = 1_{ten} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} & = 2_{ten} \\
\ldots \qquad\qquad\qquad\qquad\qquad\qquad \ldots & \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} & = 2{,}147{,}483{,}645_{ten} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} & = 2{,}147{,}483{,}646_{ten} \\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} & = 2{,}147{,}483{,}647_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} & = -2{,}147{,}483{,}648_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = -2{,}147{,}483{,}647_{ten} \\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} & = -2{,}147{,}483{,}646_{ten} \\
\ldots \qquad\qquad\qquad\qquad\qquad\qquad \ldots & \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} & = -3_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} & = -2_{ten} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} & = -1_{ten}
\end{array}$$

The positive half of the numbers, from 0 to $2{,}147{,}483{,}647_{ten}$ ($2^{31} - 1$), use the same representation as before. The following bit pattern ($1000\ldots0000_{two}$) represents the most negative number $-2{,}147{,}483{,}648_{ten}$ (-2^{31}). It is followed by a declining set of negative numbers: $-2{,}147{,}483{,}647_{ten}$ ($1000\ldots0001_{two}$) down to -1_{ten} ($1111\ldots1111_{two}$).

Two's complement does have one negative number, $-2{,}147{,}483{,}648_{ten}$, that has no corresponding positive number. Such imbalance was also a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

The sign bit is multiplied by -2^{31}, and the rest of the bits are then multiplied by positive versions of their respective base values.

EXAMPLE

Binary to Decimal Conversion

What is the decimal value of this 32-bit two's complement number?

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$$

ANSWER

Substituting the number's bit values into the formula above:

$$\begin{aligned}
&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0}) \\
&= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0 \\
&= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten} \\
&= -4_{ten}
\end{aligned}$$

We'll see a shortcut to simplify conversion from negative to positive soon.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware/ Software Interface

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former *integers* (declared as `int` in the program) and the latter *unsigned integers* (`unsigned int`). Some C style guides even recommend declaring the former as `signed int` to keep the distinction clear.

Let's examine two useful shortcuts when working with two's complement numbers. The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111 \ldots 111_{two}$, which represents -1. Since $x + \overline{x} = -1$, therefore $x + \overline{x} + 1 = 0$ or $\overline{x} + 1 = -x$.

EXAMPLE

Negation Shortcut

Negate 2_{ten}, and then check the result by negating -2_{ten}.

ANSWER

$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

Negating this number by inverting the bits and adding one,

$$\begin{array}{ll} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

Our next shortcut tells us how to convert a binary number represented in n bits to a number represented with more than n bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32{,}768_{ten}$ (-2^{15}) to $32{,}767_{ten}$ ($2^{15} - 1$). To add the immediate field to a 32-bit register, the computer must convert that 16-bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension*.

EXAMPLE

Sign Extension Shortcut

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

ANSWER

The 16-bit binary version of the number 2 is

$$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000\ 0000\ 0000\ 0010_{two}$$

becomes

$$\begin{array}{l} 1111\ 1111\ 1111\ 1101_{two} \\ + \qquad\qquad\qquad\qquad 1_{two} \\ \hline = 1111\ 1111\ 1111\ 1110_{two} \end{array}$$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the overwhelming choice since 1965 has been two's complement.

Check Yourself

What is the decimal value of this 64-bit two's complement number?

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_{two}$$

1) -4_{ten}

2) -8_{ten}

3) -16_{ten}

4) $18{,}446{,}744{,}073{,}709{,}551{,}609_{ten}$

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its negative is 2^n; hence, the complement or negation of a two's complement number x is $2^n - x$.

one's complement
A notation that represents the most negative value by $10\ldots000_{two}$ and the most positive value by $01\ldots11_{two}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive ($00\ldots00_{two}$) and one negative ($11\ldots11_{two}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

biased notation
A notation that represents the most negative value by $00\ldots000_{two}$ and the most positive value by $11\ldots11_{two}$, with 0 typically having the value $10\ldots00_{two}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

A third alternative representation to two's complement and sign and magnitude is called one's complement. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, which helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00\ldots00_{two}$ is positive 0 and $11\ldots11_{two}$ is negative 0. The most negative number, $10\ldots000_{two}$, represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point in Chapter 3, is to represent the most negative value by $00\ldots000_{two}$ and the most positive value by $11\ldots11_{two}$, with 0 typically having the value $10\ldots00_{two}$. This is called a biased notation, since it biases the number such that the number plus the bias has a nonnegative representation.

Elaboration: For signed decimal numbers, we used "–" to represent negative because there are no limits to the size of a decimal number. Given a fixed word size, binary and hexadecimal (see Figure 2.4) bit strings can encode the sign; hence we do not normally use "+" or "–" with binary or hexadecimal notation.

2.5 Representing Instructions in the Computer

We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions.

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.

Since registers are referred to by almost all instructions, there must be a convention to map register names into numbers. In MIPS assembly language, registers `$s0` to `$s7` map onto registers 16 to 23, and registers `$t0` to `$t7` map onto registers 8 to 15. Hence, `$s0` means register 16, `$s1` means register 17, `$s2` means register 18, . . . , `$t0` means register 8, `$t1` means register 9, and so on. We'll describe the convention for the rest of the 32 registers in the following sections.

Translating a MIPS Assembly Instruction into a Machine Instruction

EXAMPLE

Let's do the next step in the refinement of the MIPS language as an example. We'll show the real MIPS language version of the instruction represented symbolically as

```
add $t0,$s1,$s2
```

first as a combination of decimal numbers and then of binary numbers.

ANSWER

The decimal representation is

0	17	18	8	0	32

Each of these segments of an instruction is called a *field*. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation (17 = `$s1`), and the third field gives the other source operand for the addition (18 = `$s2`). The fourth field contains the number of the register that is to receive the sum (8 = `$t0`). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register `$s1` to register `$s2` and places the sum in register `$t0`.

This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this MIPS instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all MIPS instructions are 32 bits long.

instruction format A form of representation of an instruction composed of fields of binary numbers.

To distinguish it from assembly language, we call the numeric version of instructions **machine language** and a sequence of such instructions *machine code*.

machine language Binary representation used for communication within a computer system.

It would appear that you would now be reading and writing long, tedious strings of binary numbers. We avoid that tedium by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, **hexadecimal** (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 2.4 converts between hexadecimal and binary.

hexadecimal Numbers in base 16.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

FIGURE 2.4 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Because we frequently deal with different number bases, to avoid confusion we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation 0x*nnnn* for hexadecimal numbers.

EXAMPLE

Binary to Hexadecimal and Back

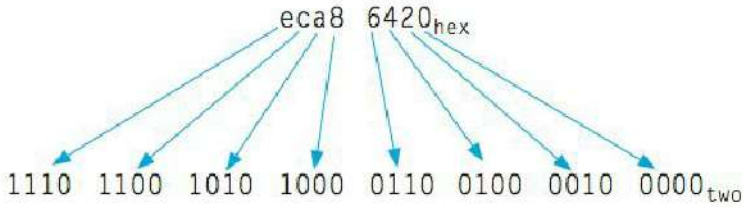
Convert the following hexadecimal and binary numbers into the other base:

$$\texttt{eca8 6420}_{hex}$$

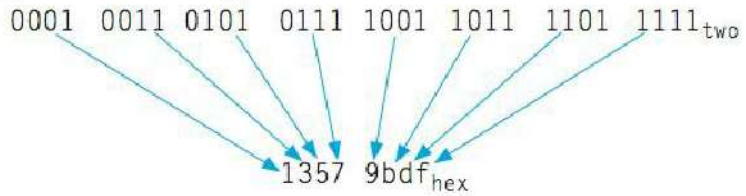
$$\texttt{0001 0011 0101 0111 1001 1011 1101 1111}_{two}$$

ANSWER

Using Figure 2.4, the answer is just a table lookup one way:



And then the other direction:



MIPS Fields

MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:

- *op:* Basic operation of the instruction, traditionally called the **opcode**.
- *rs:* The first register source operand.
- *rt:* The second register source operand.
- *rd:* The register destination operand. It gets the result of the operation.
- *shamt:* Shift amount. (Section 2.6 explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- *funct:* Function. This field, often called the *function code,* selects the specific variant of the operation in the op field.

opcode The field that denotes the operation and format of an instruction.

A problem occurs when an instruction needs longer fields than those shown above. For example, the load word instruction must specify two registers and a constant. If the address were to use one of the 5-bit fields in the format above, the constant within the load word instruction would be limited to only 2^5 or 32. This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 32. This 5-bit field is too small to be useful.

Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This leads us to the final hardware design principle:

Design Principle 4: Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$. We see that more than 32 registers would be difficult in this format, as the rs and rt fields would each need another bit, making it harder to fit everything in one word.

Let's look at the load word instruction from page 83:

```
lw   $t0,32($s3)    # Temporary reg $t0 gets A[8]
```

Here, 19 (for `$s3`) is placed in the rs field, 8 (for `$t0`) is placed in the rt field, and 32 is placed in the address field. Note that the meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the *destination* register, which receives the result of the load.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first three fields of the R-type and I-type formats are the same size and have the same names; the length of the fourth field in I-type is equal to the sum of the lengths of the last three fields of R-type.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (op) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type). Figure 2.5 shows the numbers used in each field for the MIPS instructions covered here.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{ten}	n.a.
`sub` (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
`add immediate`	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, "reg" means a register number between 0 and 31, "address" means a 16-bit address, and "n.a." (not applicable) means this field does not appear in this format. Note that `add` and `sub` instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

EXAMPLE

Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If `$t1` has the base of the array `A` and `$s2` corresponds to `h`, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw   $t0,1200($t1)# Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

ANSWER

For convenience, let's first represent the machine language instructions using decimal numbers. From Figure 2.5, we can determine the three machine language instructions:

op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

The `lw` instruction is identified by 35 (see Figure 2.5) in the first field (op). The base register 9 (`$t1`) is specified in the second field (rs), and the destination register 8 (`$t0`) is specified in the third field (rt). The offset to select `A[300]` ($1200 = 300 \times 4$) is found in the final field (address).

The `add` instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to `$s2`, `$t0`, and `$t0`.

The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

Since $1200_{ten} = 0000\ 0100\ 1011\ 0000_{two}$, the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Note the similarity of the binary representations of the first and last instructions. The only difference is in the third bit from the left, which is highlighted here.

Figure 2.6 summarizes the portions of MIPS machine language described in this section. As we shall see in Chapter 4, the similarity of the binary representations of related instructions simplifies hardware design. These similarities are another example of regularity in the MIPS architecture.

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

The BIG Picture

Today's computers are built on two key principles:

1. Instructions are represented as numbers.
2. Programs are stored in memory to be read or written, just like numbers.

These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 2.7 shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

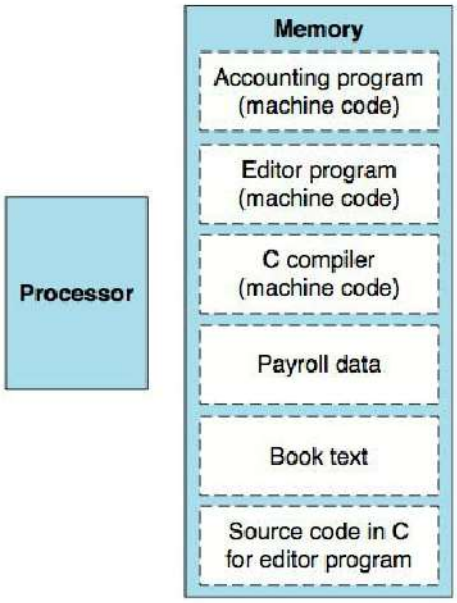


FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

Check Yourself

What MIPS instruction does this represent? Chose from one of the four options below.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. `sub $t0, $t1, $t2`
2. `add $t2, $t0, $t1`
3. `sub $t2, $t1, $t0`
4. `sub $t2, $t0, $t1`

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll, *Alice's Adventures in Wonderland*, 1865

2.6 Logical Operations

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see Section 2.9). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.8 shows logical operations in C, Java, and MIPS.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit NOT	~	~	`nor`

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

The first class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register `$s0` contained

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

and the instruction to shift left by 4 was executed, the new value would be:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (`sll`) and *shift right logical* (`srl`). The following

instruction performs the operation above, assuming that the original value was in register `$s0` and the result should go in register `$t2`:

```
sll  $t2,$s0,4   # reg $t2 = reg $s0 << 4 bits
```

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of `sll` is 0 in both the op and funct fields, rd contains 10 (register `$t2`), rt contains 16 (register `$s0`), and shamt contains 4. The rs field is unused and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by i bits gives the same result as multiplying by 2^i, just as shifting a decimal number by i digits is equivalent to multiplying by 10^i. For example, the above `sll` shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and $9 \times 16 = 144$, the value of the second bit pattern.

Another useful operation that isolates fields is **AND**. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register `$t2` contains

AND A logical bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in *both* operands.

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 1100\ 0000_{two}$$

and register `$t1` contains

$$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{two}$$

then, after executing the MIPS instruction

```
and $t0,$t1,$t2     # reg $t0 = reg $t1 & reg $t2
```

the value of register `$t0` would be

$$0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{two}$$

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask "conceals" some bits.

OR A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers `$t1` and `$t2` are unchanged from the preceding example, the result of the MIPS instruction

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

is this value in register `$t0`:

$$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 1100\ 0000_{two}$$

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

NOR A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in *both* operands.

The final logical operation is a contrarian. **NOT** takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. In keeping with the three-operand format, the designers of MIPS decided to include the instruction **NOR** (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT: A NOR 0 = NOT (A OR 0) = NOT (A).

If the register `$t1` is unchanged from the preceding example and register `$t3` has the value 0, the result of the MIPS instruction

```
nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

is this value in register `$t0`:

$$1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_{two}$$

Figure 2.8 above shows the relationship between the C and Java operators and the MIPS instructions. Constants are useful in AND and OR logical operations as well as in arithmetic operations, so MIPS also provides the instructions *and immediate* (`andi`) and *or immediate* (`ori`). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version.

Elaboration: The full MIPS instruction set also includes exclusive or (XOR), which sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same. C allows *bit fields* or *fields* to be defined within words, both allowing objects to be

packed within a word and to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: `and`, `or`, `sll`, and `srl`.

Which operations can isolate a field in a word?

1. AND
2. A shift left followed by a shift right

Check Yourself

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation. ... This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Burks, Goldstine, and von Neumann, 1947

2.7 Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
beq register1, register2, L1
```

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`. The mnemonic `beq` stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled `L1` if the value in `register1` does *not* equal the value in `register2`. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

conditional branch
An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

Compiling *if-then-else* into Conditional Branches

EXAMPLE

In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C *if* statement?

```
if (i == j) f = g + h; else f = g - h;
```

ANSWER

Figure 2.9 is a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (beq). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label Else is defined below) and so we use the branch if registers are *not* equal instruction (bne):

```
bne $s3,$s4,Else    # go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2     # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as j (the label Exit is defined below).

```
j Exit      # go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label Else to this instruction. We also show the label Exit that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```

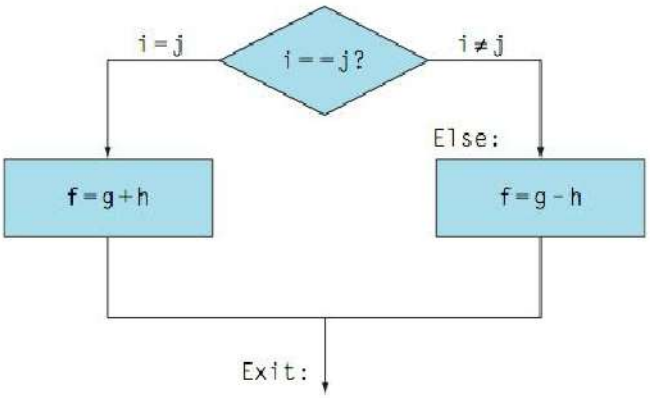


FIGURE 2.9 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.12).

Hardware/Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

EXAMPLE

Compiling a *while* Loop in C

Here is a traditional loop in C:

```
while (save[i] == k)
        i += 1;
```

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

ANSWER

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 103 in the prior section). We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll  $t1,$s3,2     # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of `save` in `$s6`:

```
      add $t1,$t1,$s6     # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
      lw  $t0,0($t1)      # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i]` ≠ `k`:

```
      bne  $t0,$s5, Exit  # go to Exit if save[i] ≠ k
```

The next instruction adds `1` to `i`:

```
      addi  $s3,$s3,1     # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
      j     Loop          # go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Hardware/Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two

registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than*, or `slt`. For example,

```
slt     $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
slti    $t0,$s2,10       # $t0 = 1 if $s2 < 10
```

Hardware/Software Interface

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of `0` (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

Hardware/Software Interface

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

EXAMPLE

Signed versus Unsigned Comparison

Suppose register `$s0` has the binary number

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$$

and that register `$s1` has the binary number

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$$

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

ANSWER

The value in register `$s0` represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1_{ten} in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \le x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

EXAMPLE

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$s1` ≥ `$t2` or if `$s1` is negative.

ANSWER

The checking code just uses `sltu` to do both checks:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction, which is described in the next section.

jump address table Also called **jump table**. A table of addresses of alternative instruction sequences.

Hardware/Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

Elaboration: If you have heard about *delayed branches*, covered in Chapter 4, don't worry: the MIPS assembler makes them invisible to the assembly language programmer.

Check Yourself

I. C has many statements for decisions and loops, while MIPS has few. Which of the following do or do not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while MIPS doesn't?

1. Logical operations AND and OR implement & and |, while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.8 Supporting Procedures in Computer Hardware

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

A **procedure** or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results. We describe the equivalent to procedures in Java in Section 2.15 on the CD, but Java needs everything from a computer that C needs.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a "need to know" basis, so the spy can't make assumptions about his employer.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- `$a0`–`$a3`: four argument registers in which to pass parameters
- `$v0`–`$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register `$ra`. The **jump-and-link instruction** (`jal`) is simply written

```
jal ProcedureAddress
```

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link," stored in register `$ra` (register 31), is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

To support such situations, computers like MIPS use *jump register* instruction (`jr`), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:

```
jr  $ra
```

Jump register instruction jumps to the address stored in register `$ra`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `$a0`–`$a3` and uses `jal X` to jump to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in `$v0` and `$v1`, and returns control to the caller using `jr $ra`.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *PC* in the MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves PC + 4 in register `$ra` to link to the following instruction to set up the procedure return.

jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

program counter (PC) The register containing the address of the instruction in the program being executed.

Using More Registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the *Hardware/Software Interface* section.

stack A data structure for spilling registers organized as a last-in-first-out queue.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register `$sp`.

push Add element to stack.

pop Remove element from stack.

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The **stack pointer** is adjusted by one word for each register that is saved or restored. MIPS software reserves register 29 for the stack pointer, giving it the obvious name `$sp`. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a **push**, and removing data from the stack is called a **pop**.

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a C Procedure That Doesn't Call Another Procedure

EXAMPLE

Let's turn the example on page 79 from Section 2.2 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

What is the compiled MIPS assembly code?

ANSWER

The parameter variables `g`, `h`, `i`, and `j` correspond to the argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` corresponds to `$s0`. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 79, which uses two temporary registers. Thus, we need to save three registers: `$s0`, `$t0`, and `$t1`. We "push" the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw   $t1, 8($sp)      # save register $t1 for use afterwards
sw   $t0, 4($sp)      # save register $t0 for use afterwards
sw   $s0, 0($sp)      # save register $s0 for use afterwards
```

Figure 2.10 shows the stack before, during, and after the procedure call.

The next three statements correspond to the body of the procedure, which follows the example on page 79:

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

To return the value of `f`, we copy it into a return value register:

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
lw   $s0, 0($sp)  # restore register $s0 for caller
lw   $t0, 4($sp)  # restore register $t0 for caller
lw   $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12   # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr   $ra     # jump back to calling routine
```

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- `$t0`–`$t9`: ten temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0`–`$s7`: eight saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers `$t0` and `$t1` to be preserved across a procedure call,

we can drop two stores and two loads from the code. We still must save and restore `$s0`, since the callee must assume that the caller needs its value.

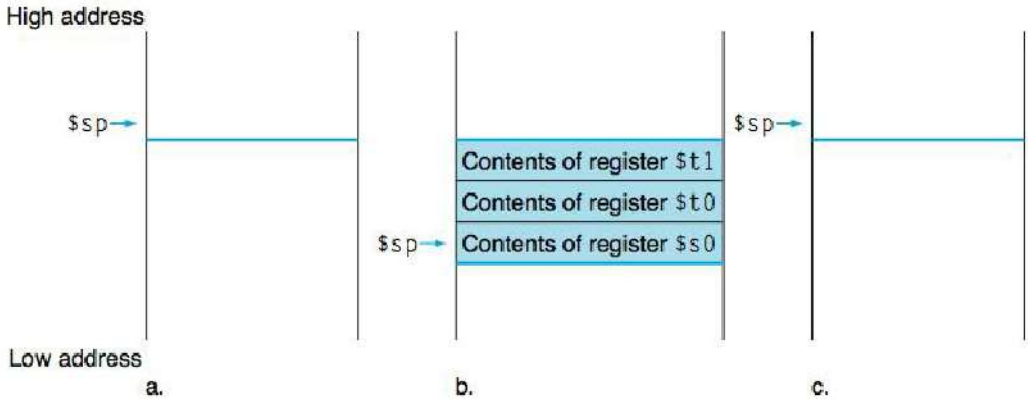


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Nested Procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register `$a0` and then using `jal A`. Then suppose that procedure A calls procedure B via `jal B` with an argument of 7, also placed in `$a0`. Since A hasn't finished its task yet, there is a conflict over the use of register `$a0`. Similarly, there is a conflict over the return address in register `$ra`, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved onto the stack, just as we did with the saved registers. The caller pushes any argument registers (`$a0–$a3`) or temporary registers (`$t0–$t9`) that are needed after the call. The callee pushes the return address register `$ra` and any saved registers (`$s0–$s7`) used by the callee. The stack pointer `$sp` is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

EXAMPLE

Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
    if (n < 1) return (1);
        else return (n * fact(n - 1));
}
```

What is the MIPS assembly code?

ANSWER

The parameter variable `n` corresponds to the argument register `$a0`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `$a0`:

```
fact:
    addi  $sp, $sp, -8  # adjust stack for 2 items
    sw    $ra, 4($sp)   # save the return address
    sw    $a0, 0($sp)   # save the argument n
```

The first time `fact` is called, `sw` saves an address in the program that called `fact`. The next two instructions test whether `n` is less than 1, going to `L1` if $n \geq 1$.

```
    slti  $t0,$a0,1       # test for n < 1
    beq   $t0,$zero,L1    # if n >= 1, go to L1
```

If `n` is less than 1, `fact` returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in `$v0`. It then pops the two saved values off the stack and jumps to the return address:

```
    addi  $v0,$zero,1 # return 1
    addi  $sp,$sp,8   # pop 2 items off stack
    jr    $ra         # return to caller
```

Before popping two items off the stack, we could have loaded `$a0` and `$ra`. Since `$a0` and `$ra` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: addi $a0,$a0,-1   # n >= 1: argument gets (n - 1)
    jal  fact         # call fact with (n - 1)
```

The next instruction is where `fact` returns. Now the old return address and old argument are restored, along with the stack pointer:

```
lw   $a0, 0($sp)    # return from jal: restore argument n
lw   $ra, 4($sp)    # restore the return address
addi $sp, $sp, 8    # adjust stack pointer to pop 2 items
```

Next, the value register `$v0` gets the product of old argument `$a0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul  $v0,$a0,$v0   # return n * fact (n - 1)
```

Finally, `fact` jumps again to the return address:

```
jr   $ra           # return to the caller
```

Hardware/Software Interface

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Examples include integers and characters (see Section 2.9). C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*. The rest are automatic. To simplify access to static data, MIPS software reserves another register, called the **global pointer**, or `$gp`.

global pointer The register that is reserved to point to the static area.

Figure 2.11 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above `$sp` is preserved simply by making sure the callee does not write above `$sp`; `$sp` is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Preserved	Not preserved
Saved registers: `$s0`–`$s7`	Temporary registers: `$t0`–`$t9`
Stack pointer register: `$sp`	Argument registers: `$a0`–`$a3`
Return address register: `$ra`	Return value registers: `$v0`–`$v1`
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.11 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.12 shows the state of the stack before, during, and after the procedure call.

procedure frame Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

Some MIPS software uses a **frame pointer** (`$fp`) to point to the first word of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory-references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We've been avoiding using `$fp` by avoiding changes to `$sp` within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.

frame pointer A value denoting the location of the saved registers and local variables for a given procedure.

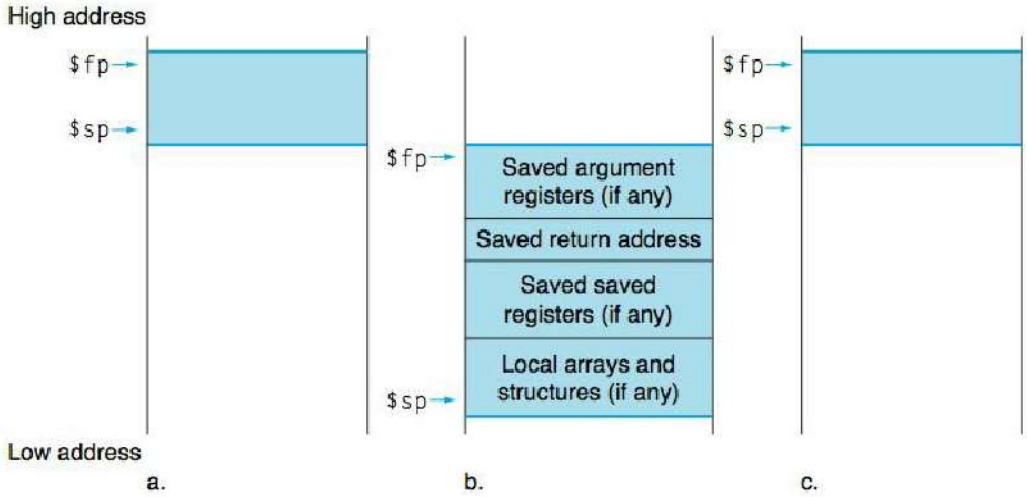


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `$sp` on a call, and `$sp` is restored using `$fp`. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Allocating Space for New Data on the Heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. Figure 2.13 shows the MIPS convention for allocation of memory. The stack starts in the high end of memory and grows down. The first part of the low end of memory is reserved, followed by the home of the MIPS machine code, traditionally called the **text segment**. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap*, and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

text segment The segment of a UNIX object file that contains the machine language code for routines in the source file.

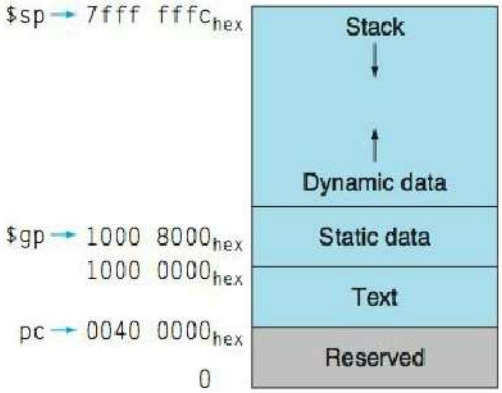


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 0000$_{hex}$. The static data starts at 1000 0000$_{hex}$. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, $gp, is set to an address to make it easy to access data. It is initialized to 1000 8000$_{hex}$ so that it can access from 1000 0000$_{hex}$ to 1000 ffff$_{hex}$ using the positive and negative 16-bit offsets from $gp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

C allocates and frees space on the heap with explicit functions. malloc() allocates space on the heap and returns a pointer to it, and free() releases space on the heap to which the pointer points. Memory allocation is controlled by programs in C, and it is the source of many common and difficult bugs. Forgetting to free space leads to a "memory leak," which eventually uses up so much memory that the operating system may crash. Freeing space too early leads to "dangling pointers," which can cause pointers to point to things that the program never intended. Java uses automatic memory allocation and garbage collection just to avoid such bugs.

Figure 2.14 summarizes the register conventions for the MIPS assembly language.

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called $at, is reserved for the assembler (see Section 2.12), and registers 26–27, called $k0–$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

Elaboration: What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first four parameters to be in registers $a0 through $a3 and the rest in memory, addressable via the frame pointer.

As mentioned in the caption of Figure 2.12, the frame pointer is convenient because all references to variables in the stack within a procedure will have the same offset. The frame pointer is not necessary, however. The GNU MIPS C compiler uses a frame pointer, but the C compiler from MIPS does not; it treats register 30 as another save register ($s8).

Elaboration: Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with procedure calls. For example, consider a procedure used to accumulate a sum:

```
int sum (int n, int acc) {
  if (n > 0)
      return sum(n - 1, acc + n);
  else
      return acc;
}
```

Consider the procedure call sum(3,0). This will result in recursive calls to sum(2,3), sum(1,5), and sum(0,6), and then the result 6 will be returned four times. This recursive call of sum is referred to as a *tail call*, and this example use of tail recursion can be implemented very efficiently (assume $a0 = n and $a1 = acc):

```
sum: slti $t0, $a0, 1          # test if n <= 0
     bne $t0, $zero, sum_exit  # go to sum_exit if n <= 0
     add$a1, $a1, $a0          # add n to acc
```

```
        addi$a0, $a0, -1        # subtract 1 from n
        j sum                   # go to sum
sum_exit:
        add$v0, $a1, $zero      # return value acc
        jr $ra                  # return to caller
```

Check Yourself Which of the following statements about C and Java are generally true?

1. C programmers manage data explicitly, while it's automatic in Java.
2. C leads to more pointer bugs and memory leak bugs than does Java.

!(@|=>(wow open tab at bar is great)

Fourth line of the keyboard poem "Hatless Atlas," 1991 (some give names to ASCII characters: "!" is "wow," "(" is open, "|" is bar, and so on).

2.9 Communicating with People

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today offer 8-bit bytes to represent characters, with the American Standard Code for Information Interchange (ASCII) being the representation that nearly everyone follows. Figure 2.15 summarizes ASCII.

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.

Hardware/ Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted to all binary, converting to base 10 only for the relatively infrequent input/output events.

ASCII versus Binary Numbers

EXAMPLE

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

ANSWER

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be $(10 \times 8)/32$ or 2.5. In addition to the expansion in storage, the hardware to add, subtract, multiply, and divide such decimal numbers is difficult. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

A series of instructions can extract a byte from a word, so load word and store word are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, MIPS provides instructions to move bytes. Load byte (`lb`) loads a byte from memory, placing it in the rightmost 8 bits of a register. Store byte (`sb`) takes a byte from the rightmost 8 bits of a register and writes it to memory. Thus, we copy a byte with the sequence

```
lb $t0,0($sp)          # Read byte from source
sb $t0,0($gp)          # Write byte to destination
```

Hardware/ Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (`lb`) treats the byte as a signed number and thus sign-extends to fill the 24 leftmost bits of the register, while *load byte unsigned* (`lbu`) works with unsigned integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `lbu` is used practically exclusively for byte loads.

Characters are normally combined into strings, which have a variable number of characters. There are three choices for representing a string: (1) the first position of the string is reserved to give the length of a string, (2) an accompanying variable has the length of the string (as in a structure), or (3) the last position of a string is indicated by a character used to mark the end of a string. C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII). Thus, the string "Cal" is represented in C by the following 4 bytes, shown as decimal numbers: 67, 97, 108, 0. (As we shall see, Java uses the first option.)

EXAMPLE

Compiling a String Copy Procedure, Showing How to Use C Strings

The procedure `strcpy` copies string `y` to string `x` using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
     i += 1;
}
```

What is the MIPS assembly code?

ANSWER

Below is the basic MIPS assembly code segment. Assume that base addresses for arrays x and y are found in $a0 and $a1, while i is in $s0. strcpy adjusts the stack pointer and then saves the saved register $s0 on the stack:

```
strcpy:
    addi   $sp,$sp,-4    # adjust stack for 1 more item
    sw     $s0, 0($sp)   # save $s0
```

To initialize i to 0, the next instruction sets $s0 to 0 by adding 0 to 0 and placing that sum in $s0:

```
    add    $s0,$zero,$zero # i = 0 + 0
```

This is the beginning of the loop. The address of y[i] is first formed by adding i to y[]:

```
L1: add    $t1,$s0,$a1   # address of y[i] in $t1
```

Note that we don't have to multiply i by 4 since y is an array of *bytes* and not of words, as in prior examples.

To load the character in y[i], we use load byte unsigned, which puts the character into $t2:

```
    lbu    $t2, 0($t1)   # $t2 = y[i]
```

A similar address calculation puts the address of x[i] in $t3, and then the character in $t2 is stored at that address.

```
    add    $t3,$s0,$a0   # address of x[i] in $t3
    sb     $t2, 0($t3)   # x[i] = y[i]
```

Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:

```
    beq    $t2,$zero,L2 # if y[i] == 0, go to L2
```

If not, we increment i and loop back:

```
    addi   $s0, $s0,1    # i = i + 1
    j      L1            # go to L1
```

If we don't loop back, it was the last character of the string; we restore $s0 and the stack pointer, and then return.

```
L2: lw    $s0, 0($sp) # y[i] == 0: end of string. Re-
store old $s0

    addi  $sp,$sp,4   # pop 1 word off stack
    jr    $ra         # return
```

String copies usually use pointers instead of arrays in C to avoid the operations on i in the code above. See Section 2.14 for an explanation of arrays versus pointers.

Since the procedure strcpy above is a leaf procedure, the compiler could allocate i to a temporary register and avoid saving and restoring $s0. Hence, instead of thinking of the $t registers as being just for temporaries, we can think of them as registers that the callee should use whenever convenient. When a compiler finds a leaf procedure, it exhausts all temporary registers before using registers it must save.

Characters and Strings in Java

Unicode is a universal encoding of the alphabets of most human languages. Figure 2.16 is a list of Unicode alphabets; there are almost as many *alphabets* in Unicode as there are useful *symbols* in ASCII. To be more inclusive, Java uses Unicode for characters. By default, it uses 16 bits to represent a character.

The MIPS instruction set has explicit instructions to load and store such 16-bit quantities, called *halfwords*. Load half (lh) loads a halfword from memory, placing it in the rightmost 16 bits of a register. Like load byte, *load half* (lh) treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register, while *load halfword unsigned* (lhu) works with unsigned integers. Thus, lhu is the more popular of the two. Store half (sh) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence

```
lhu $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

Strings are a standard Java class with special built-in support and predefined methods for concatenation, comparison, and conversion. Unlike C, Java includes a word that gives the length of the string, similar to Java arrays.

Elaboration: MIPS software tries to keep the stack aligned to word addresses, allowing the program to always use lw and sw (which must be aligned) to access the stack. This convention means that a char variable allocated on the stack occupies 4 bytes, even though it needs less. However, a C string variable or an array of bytes *will* pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURE 2.16 Example alphabets in Unicode. Unicode version 4.0 has more than 160 "blocks," which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16–32 bits for the other characters. UTF-32 uses 32 bits per character. To learn more, see *www.unicode.org*.

Check Yourself

I. Which of the following statements about characters and strings in C and Java are true?

1. A string in C takes about half the memory as the same string in Java.
2. Strings are just an informal name for single-dimension arrays of characters in C and Java.
3. Strings in C and Java use null (0) to mark the end of a string.
4. Operations on strings, like length, are faster in C than in Java.

II. Which type of variable that can contain $1{,}000{,}000{,}000_{ten}$ takes the most memory space?

1. `int` in C
2. `string` in C
3. `string` in Java

2.10 MIPS Addressing for 32-Bit Immediates and Addresses

Although keeping all MIPS instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches and jumps.

32-Bit Immediate Operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (`lui`) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. Figure 2.17 shows the operation of `lui`.

EXAMPLE

Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

ANSWER

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61   # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `$s0` afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to insert the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register `$s0` is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

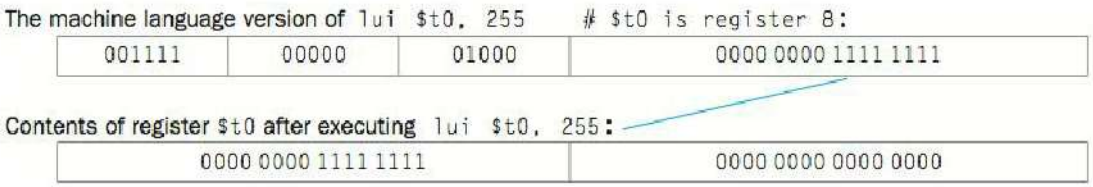


FIGURE 2.17 The effect of the `lui` instruction. The instruction `lui` transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

Hardware/Software Interface

Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions. If this job falls to the assembler, as it does for MIPS software, then the assembler must have a temporary register available in which to create the long values. This is a reason for the register `$at`, which is reserved for the assembler.

Hence, the symbolic representation of the MIPS machine language is no longer limited by the hardware, but by whatever the creator of an assembler chooses to include (see Section 2.12). We stick close to the hardware to explain the architecture of the computer, noting when we use the enhanced language of the assembler that is not found in the processor.

Elaboration: Creating 32-bit constants needs care. The instruction `addi` copies the leftmost bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. *Logical or immediate* from Section 2.6 loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with `lui` to create 32-bit constants.

Addressing in Branches and Jumps

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
j   10000   # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

2	10000
6 bits	26 bits

where the value of the jump opcode is 2 and the jump address is 10000.

Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

```
bne  $s0,$s1,Exit   # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^{16}, which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch address}$$

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the program counter (PC) contains the address of the current instruction, we can branch within $\pm 2^{15}$ words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if* statements are much smaller than 2^{16} words, so the PC is the ideal choice.

PC-relative addressing An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

This form of branch addressing is called **PC-relative addressing**. As we shall see in Chapter 4, it is convenient for the hardware to increment the PC early to point to the next instruction. Hence, the MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to the current instruction (PC).

Like most recent computers, MIPS uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch. On the other hand, jump-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 16-bit field can branch four

times as far by interpreting the field as a relative word address rather than as a relative byte address. Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.

Elaboration: Since the PC is 32 bits, 4 bits must come from somewhere else for jumps. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker (Section 2.12) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

Showing Branch Offset in Machine Language

EXAMPLE

The *while* loop on page 107–108 was compiled into this MIPS assembler code:

```
Loop:sll    $t1,$s3,2   # Temp reg $t1 = 4 * i
     add $t1,$t1,$s6    # $t1 = address of save[i]
     lw  $t0,0($t1)     # Temp reg $t0 = save[i]
     bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
     addi $s3,$s3,1     # i = i + 1
     j   Loop           # go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

ANSWER

The assembled instructions and their addresses are:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					