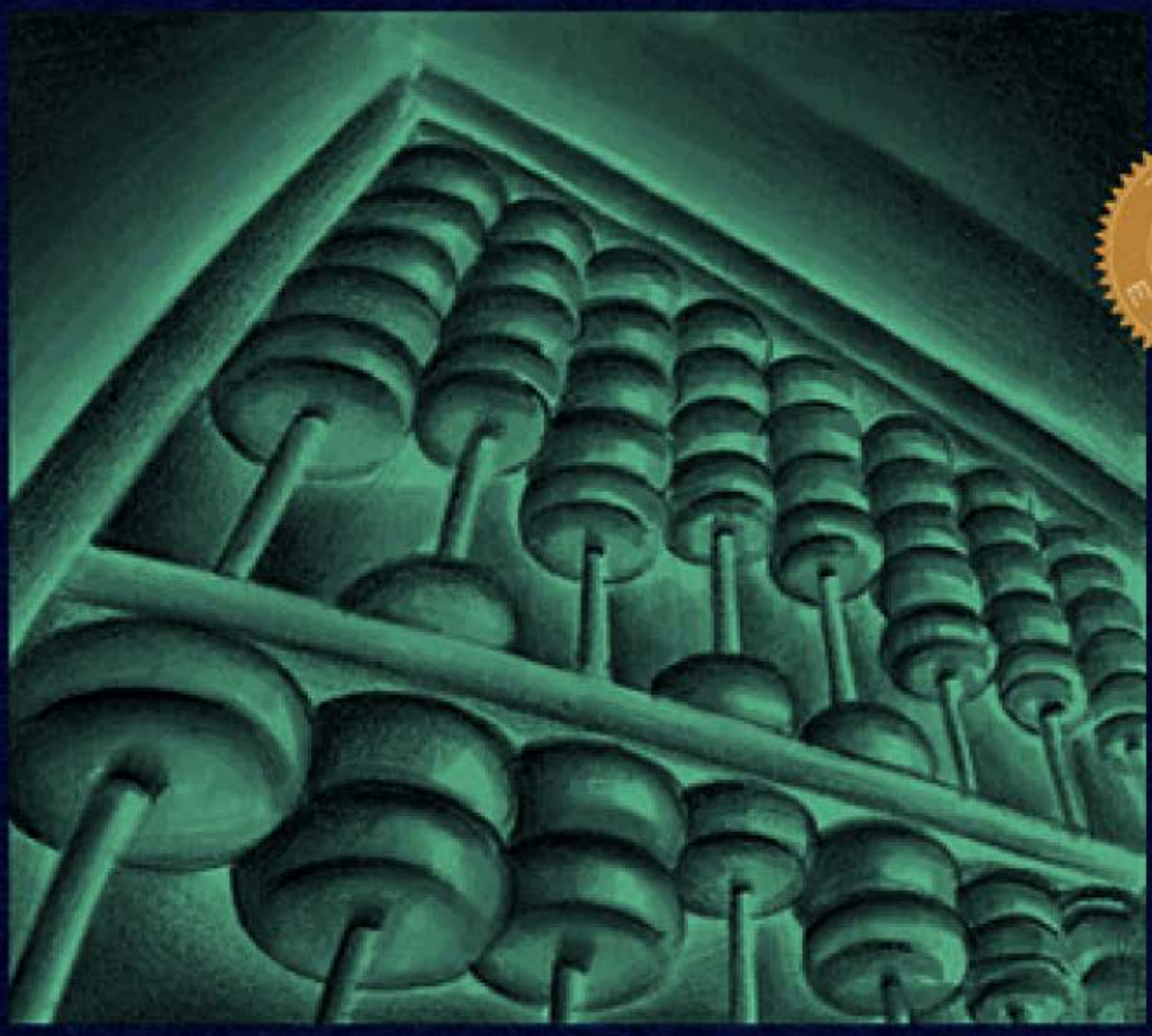
ELSEVIER

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE



THIRD EDITION 3

DAVID A. PATTERSON
JOHN L. HENNESSY

MK MORGAN KAUFMANN

INTEL-1018
9,135,190

THIRD EDITION

Computer Organization Design

THE HARDWARE/SOFTWARE INTERFACE

ACKNOWLEDGEMENTS

Figures 1.9, 1.15 Courtesy of Intel.

Figure 1.11 Courtesy of Storage Technology Corp.

Figures 1.7.1, 1.7.2, 6.13.2 Courtesy of the Charles Babbage Institute, University of Minnesota Libraries, Minneapolis.

Figures 1.7.3, 6.13.1, 6.13.3, 7.9.3, 8.11.2 Courtesy of IBM.

Figure 1.7.4 Courtesy of Cray Inc.

Figure 1.7.5 Courtesy of Apple Computer, Inc.

Figure 1.7.6 Courtesy of the Computer History Museum.

Figure 7.33 Courtesy of AMD.

Figures 7.9.1, 7.9.2 Courtesy of Museum of Science, Boston.

Figure 7.9.4 Courtesy of MIPS Technologies, Inc.

Figure 8.3 ©Peg Skorpinski.

Figure 8.11.1 Courtesy of the Computer Museum of America.

Figure 8.11.3 Courtesy of the Commercial Computing Museum.

Figures 9.11.2, 9.11.3 Courtesy of NASA Ames Research Center.

Figure 9.11.4 Courtesy of Lawrence Livermore National Laboratory.

Computers in the Real World:

Photo of "A Laotian villager," courtesy of David Sanger.

Photo of an "Indian villager," property of Encore Software, Ltd., India.

Photos of "Block and students" and "a pop-up archival satellite tag," courtesy of Professor Barbara Block. Photos by Scott Taylor.

Photos of "Professor Dawson and student" and "the Mica micromote," courtesy of AP/World Wide Photos.

Photos of "images of pottery fragments" and "a computer reconstruction," courtesy of Andrew Willis and David B. Cooper, Brown University, Division of Engineering.

Photo of "the Eurostar TGV train," by Jos van der Kolk.

Photo of "the interior of a Eurostar TGV cab," by Andy Veitch.

Photo of "firefighter Ken Whitten," courtesy of World Economic Forum.

Graphic of an "artificial retina," © The San Francisco Chronicle. Reprinted by permission.

Image of "A laser scan of Michelangelo's statue of David," courtesy of Marc Levoy and Dr. Franca Falletti, director of the Galleria dell'Accademia, Italy.

"An image from the Sistine Chapel," courtesy of Luca Pezzati. IR image recorded using the scanner for IR reflectography of the INOA (National Institute for Applied Optics, http://arte.ino.it) at the Opificio delle Pietre Dure in Florence.

THIRD EDITION

Computer Organization and Design

THE HARDWARE/SOFTWARE INTERFACE

David A. Patterson
University of California, Berkeley

John L. Hennessy
Stanford University

With a contribution by

Peter J. Ashenden
Ashenden Designs Pty Ltd

James R. Larus
Microsoft Research

Daniel J. Sorin
Duke University



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 PUBLISHERS

Senior Editor	Denise E. M. Penrose
Publishing Services Manager	Simon Crump
Editorial Assistant	Summer Block
Cover Design	Ross Caron Design
Cover and Chapter Illustration	Chris Asimoudis
Text Design	GGS Book Services
Composition	Nancy Logan and Dartmouth Publishing, Inc.
Technical Illustration	Dartmouth Publishing, Inc.
Copyeditor	Ken DellaPenta
Proofreader	Jacqui Brownstein
Indexer	Linda Buskus
Interior printer	Courier
Cover printer	Courier

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

© 2005 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (http://elsevier.com) by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Application submitted

ISBN: 1-55860-604-1

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com.

Printed in the United States of America
04 05 06 07 08 5 4 3 2 1

Contents

Preface ix

CHAPTERS

1 **Computer Abstractions and Technology 2**

1.1 Introduction 3
1.2 Below Your Program 11
1.3 Under the Covers 15
1.4 Real Stuff: Manufacturing Pentium 4 Chips 28
1.5 Fallacies and Pitfalls 33
1.6 Concluding Remarks 35
1.7 Historical Perspective and Further Reading 36
1.8 Exercises 36

COMPUTERS IN THE REAL WORLD
Information Technology for the 4 Billion without IT 44

2 **Instructions: Language of the Computer 46**

2.1 Introduction 48
2.2 Operations of the Computer Hardware 49
2.3 Operands of the Computer Hardware 52
2.4 Representing Instructions in the Computer 60
2.5 Logical Operations 68
2.6 Instructions for Making Decisions 72
2.7 Supporting Procedures in Computer Hardware 79
2.8 Communicating with People 90
2.9 MIPS Addressing for 32-Bit Immediates and Addresses 95
2.10 Translating and Starting a Program 106
2.11 How Compilers Optimize 116
2.12 How Compilers Work: An Introduction 121

2.13 A C Sort Example to Put It All Together 121
2.14 Implementing an Object-Oriented Language 130
2.15 Arrays versus Pointers 130
2.16 Real Stuff: IA-32 Instructions 134
2.17 Fallacies and Pitfalls 143
2.18 Concluding Remarks 145
2.19 Historical Perspective and Further Reading 147
2.20 Exercises 147

COMPUTERS IN THE REAL WORLD
Helping Save Our Environment with Data 156

3 Arithmetic for Computers 158

3.1 Introduction 160
3.2 Signed and Unsigned Numbers 160
3.3 Addition and Subtraction 170
3.4 Multiplication 176
3.5 Division 183
3.6 Floating Point 189
3.7 Real Stuff: Floating Point in the IA-32 217
3.8 Fallacies and Pitfalls 220
3.9 Concluding Remarks 225
3.10 Historical Perspective and Further Reading 229
3.11 Exercises 229

COMPUTERS IN THE REAL WORLD
Reconstructing the Ancient World 236

4 Assessing and Understanding Performance 238

4.1 Introduction 240
4.2 CPU Performance and Its Factors 246
4.3 Evaluating Performance 254
4.4 Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors 259
4.5 Fallacies and Pitfalls 266
4.6 Concluding Remarks 270
4.7 Historical Perspective and Further Reading 272
4.8 Exercises 272

COMPUTERS IN THE REAL WORLD
Moving People Faster and More Safely 280

5 The Processor: Datapath and Control 282

5.1 Introduction 284
5.2 Logic Design Conventions 289
5.3 Building a Datapath 292
5.4 A Simple Implementation Scheme 300
5.5 A Multicycle Implementation 318
5.6 Exceptions 340
5.7 Microprogramming: Simplifying Control Design 346
5.8 An Introduction to Digital Design Using a Hardware Design Language 346
5.9 Real Stuff: The Organization of Recent Pentium Implementations 347
5.10 Fallacies and Pitfalls 350
5.11 Concluding Remarks 352
5.12 Historical Perspective and Further Reading 353
5.13 Exercises 354

COMPUTERS IN THE REAL WORLD
Empowering the Disabled 366

6 Enhancing Performance with Pipelining 368

6.1 An Overview of Pipelining 370
6.2 A Pipelined Datapath 384
6.3 Pipelined Control 399
6.4 Data Hazards and Forwarding 402
6.5 Data Hazards and Stalls 413
6.6 Branch Hazards 416
6.7 Using a Hardware Description Language to Describe and Model a Pipeline 426
6.8 Exceptions 427
6.9 Advanced Pipelining: Extracting More Performance 432
6.10 Real Stuff: The Pentium 4 Pipeline 448
6.11 Fallacies and Pitfalls 451
6.12 Concluding Remarks 452
6.13 Historical Perspective and Further Reading 454
6.14 Exercises 454

COMPUTERS IN THE REAL WORLD
Mass Communication without Gatekeepers 464



Large and Fast: Exploiting Memory Hierarchy 466

7.1 Introduction 468
7.2 The Basics of Caches 473
7.3 Measuring and Improving Cache Performance 492
7.4 Virtual Memory 511
7.5 A Common Framework for Memory Hierarchies 538
7.6 Real Stuff: The Pentium P4 and the AMD Opteron Memory Hierarchies 546
7.7 Fallacies and Pitfalls 550
7.8 Concluding Remarks 552
7.9 Historical Perspective and Further Reading 555
7.10 Exercises 555

COMPUTERS IN THE REAL WORLD
Saving the World's Art Treasures 562



Storage, Networks, and Other Peripherals 564

8.1 Introduction 566
8.2 Disk Storage and Dependability 569
8.3 Networks 580
8.4 Buses and Other Connections between Processors, Memory, and I/O Devices 581
8.5 Interfacing I/O Devices to the Processor, Memory, and Operating System 588
8.6 I/O Performance Measures: Examples from Disk and File Systems 597
8.7 Designing an I/O System 600
8.8 Real Stuff: A Digital Camera 603
8.9 Fallacies and Pitfalls 606
8.10 Concluding Remarks 609
8.11 Historical Perspective and Further Reading 611
8.12 Exercises 611

COMPUTERS IN THE REAL WORLD
Saving Lives through Better Diagnosis 622



Multiprocessors and Clusters 9-2

9.1 Introduction 9-4
9.2 Programming Multiprocessors 9-8
9.3 Multiprocessors Connected by a Single Bus 9-11

9.4 Multiprocessors Connected by a Network 9-20
9.5 Clusters 9-25
9.6 Network Topologies 9-27
9.7 Multiprocessors Inside a Chip and Multithreading 9-30
9.8 Real Stuff: The Google Cluster of PCs 9-34
9.9 Fallacies and Pitfalls 9-39
9.10 Concluding Remarks 9-42
9.11 Historical Perspective and Further Reading 9-47
9.12 Exercises 9-55

APPENDICES

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

A.1 Introduction A-3
A.2 Assemblers A-10
A.3 Linkers A-18
A.4 Loading A-19
A.5 Memory Usage A-20
A.6 Procedure Call Convention A-22
A.7 Exceptions and Interrupts A-33
A.8 Input and Output A-38
A.9 SPIM A-40
A.10 MIPS R2000 Assembly Language A-45
A.11 Concluding Remarks A-81
A.12 Exercises A-82

B The Basics of Logic Design B-2

B.1 Introduction B-3
B.2 Gates, Truth Tables, and Logic Equations B-4
B.3 Combinational Logic B-8
B.4 Using a Hardware Description Language B-20
B.5 Constructing a Basic Arithmetic Logic Unit B-26
B.6 Faster Addition: Carry Lookahead B-38
B.7 Clocks B-47
B.8 Memory Elements: Flip-flops, Latches, and Registers B-49
B.9 Memory Elements: SRAMs and DRAMs B-57
B.10 Finite State Machines B-67
B.11 Timing Methodologies B-72

B.12 Field Programmable Devices B-77
B.13 Concluding Remarks B-78
B.14 Exercises B-79



Mapping Control to Hardware C-2

C.1 Introduction C-3
C.2 Implementing Combinational Control Units C-4
C.3 Implementing Finite State Machine Control C-8
C.4 Implementing the Next-State Function with a Sequencer C-21
C.5 Translating a Microprogram to Hardware C-27
C.6 Concluding Remarks C-31
C.7 Exercises C-32



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

D.1 Introduction D-3
D.2 Addressing Modes and Instruction Formats D-5
D.3 Instructions: The MIPS Core Subset D-9
D.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs D-16
D.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs D-19
D.6 Instructions: Common Extensions to MIPS Core D-20
D.7 Instructions Unique to MIPS64 D-25
D.8 Instructions Unique to Alpha D-27
D.9 Instructions Unique to SPARC v.9 D-29
D.10 Instructions Unique to PowerPC D-32
D.11 Instructions Unique to PA-RISC 2.0 D-34
D.12 Instructions Unique to ARM D-36
D.13 Instructions Unique to Thumb D-38
D.14 Instructions Unique to SuperH D-39
D.15 Instructions Unique to M32R D-40
D.16 Instructions Unique to MIPS16 D-41
D.17 Concluding Remarks D-43
D.18 Acknowledgments D-46
D.19 References D-47

Index I-1
Glossary G-1
Further Reading FR-1

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 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 called Patterson and Hennessy.) Our motivation in writing that book was to describe the principles of computer architecture using solid engineering funda-

mentals and quantitative cost/performance trade-offs. 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 is intended for the serious computing professional who wants a detailed understanding of computers.

A majority of the readers for this book do not plan to become computer architects. The performance 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 Third Edition

We had six major goals for the third edition of *Computer Organization and Design:* make the book work equally well for readers with a software focus or with a hardware focus; improve pedagogy in general; enhance understanding of program performance; update the technical content to reflect changes in the industry since the publication of the second edition in 1998; tie the ideas from the book more closely to the real world *outside* the computing industry; and reduce the size of this book.

First, the table on the next page shows the hardware and software paths through the material. Chapters 1, 4, and 7 are found on both paths, no matter what the experience or the focus. Chapters 2 and 3 are likely to be review material for the hardware-oriented, but are essential reading for the software-oriented, especially for those readers interested in learning more about compilers and object-oriented programming languages. The first sections of Chapters 5 and 6 give overviews for those with a software focus. Those with a hardware focus, however, will find that these chapters present core material; they may also, depending on background, want to read Appendix B on logic design first and the sections on microprogramming and how to use hardware description languages to specify control. Chapter 8 on input/output is key to readers with a software focus and should be read if time permits by others. The last chapter on multiprocessors and clusters is again a question of time for the reader. Even the history sections show this balanced focus; they include short histories of programming languages, compilers, numerical software, operating systems, networking protocols, and databases.

Chapter or Appendix	*Sections*	*Software Focus*	*Hardware Focus*
1. Computer Abstractions and Technology	1.1 to 1.6		
	1.7 (History)		
2. Instructions: Language of the Computer	2.1 to 2.11		
	2.12 (Compilers)		
	2.13 (C sort)		
	2.14 (Java)		
	2.15 to 2.18		
	2.19 (History)		
3. Arithmetic for Computers	3.1 to 3.11		
	3.12 (History)		
D. RISC instruction set architectures	D.1 to D.19		
4. Assessing and Understanding Performance	4.1 to 4.6		
	4.7 (History)		
B. The Basics of Logic Design	B.1 to B.13		
5. The Processor: Datapath and Control	5.1 (Overview)		
	5.2 to 5.7		
	5.8 (Microcode)		
	5.9 (Verilog)		
	5.10 to 5.12		
	5.13 (History)		
C. Mapping Control to Hardware	C.1 to C.6		
6. Enhancing Performance with Pipelining	6.1 (Overview)		
	6.2 to 6.6		
	6.7 (verilog)		
	6.8 to 6.9		
	6.10 to 6.12		
	6.13 (History)		
7. Large and Fast: Exploiting Memory Hierarchy	7.1 to 7.8		
	7.9 (History)		
8. *Storage, Networks, and Other Peripherals*	8.1 to 8.2		
	8.3 (Networks)		
	8.4 to 8.10		
	8.13 (History)		
9. Multiprocessors and Clusters	9.1 to 9.10		
	9.11 (History)		
A. Assemblers, Linkers, and the SPIM Simulator	A.1 to A.12		
Computers in the Real World	Between Chapters		

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

The next goal was to improve the exposition of the ideas in the book, based on difficulties mentioned by readers of the second edition. We added five new book elements to help. To make the book work better as a reference, we placed definitions of new terms in the margins at their first occurrence. We hope this will help readers find the sections when they want to refer back to material they have already read. Another change was the insertion of the "Check Yourself" sections, which we added to help readers to check their comprehension of the material on the first time through it. A third change is that added extra exercises in the "For More Practice" section. Fourth, we added the answers to the "Check Yourself" sections and to the For More Practice exercises to help readers see for themselves if they understand the material by comparing their answers to the book. The final new book element was inspired by the "Green Card" of the IBM System/360. We believe that you will find that the MIPS Reference Data Card will be a handy reference when writing MIPS assembly language programs. Our idea is that you will remove the card from the front of the book, fold it in half, and keep it in your pocket, just as IBM S/360 programmers did in the 1960s.

Third, computers are so complex today that understanding the performance of a program involves understanding a good deal about the underlying principles and the organization of a given computer. Our goal is that readers of this book should be able to understand the performance of their progams and how to improve it. To aid in that goal, we added a new book element called "Understanding Program Performance" in several chapters. These sections often give concrete examples of how ideas in the chapter affect performance of real programs.

Fourth, in the interval since the second edition of this book, Moore's law has marched onward so that we now have processors with 200 million transistors, DRAM chips with a billion transistors, and clock rates of multiple gigahertz. The "Real Stuff" examples have been updated to describe such chips. This edition also includes AMD64/IA-32e, the 64-bit address version of the long-lived 80x86 architecture, which appears to be the nemesis of the more recent IA-64. It also reflects the transition from parallel buses to serial networks and switches. Later chapters describe Google, which was born after the second edition, in terms of its cluster technology and in novel uses of search.

Fifth, although many computer science and engineering students enjoy information technology for technology's sake, some have more altruistic interests. This latter group tends to have more women and underrepresented minorities. Consequently, we have added a new book element, "Computers in the Real World," two-page layouts found between each chapter. Our perspective is that information technology is more valuable for humanity than most other topics you could study—whether it is preserving our art heritage, helping the Third World, saving our environment, or even changing political systems—and so we demonstrate our view with concrete examples of nontraditional applications. We think readers of these segments will have a greater appreciation of the computing culture beyond

the inherently interesting technology, much like those who read the history sections at the end of each chapter

Finally, books are like people: they usually get larger as they get older. By using technology, we have managed to do all the above and yet shrink the page count by hundreds of pages. As the table illustrates, the core portion of the book for hardware and software readers is on paper, but sections that some readers would value more than others are found on the companion CD. This technology also allows your authors to provide longer histories and more extensive exercises without concerns about lengthening the book. Once we added the CD to the book, we could then include a great deal of free software and tutorials that many instructors have told us they would like to use in their courses. This hybrid paper-CD publication weighs about 30% less than it did six years ago—an impressive goal for books as well as for people.

Instructor Support

We have collected a great deal of material to help instructors teach courses using this book. Solutions to exercises, 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:

www.mkp.com/companions/1558606041

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 *cod3bugs@mkp.com* or by low-tech mail using the address found on the copyright page. The first person to report a technical error will be awarded a $1.00 bounty upon its implementation in future printings of the book!

This book is truly collaborative, despite one of us running a major university. Together we brainstormed about the ideas and method of presentation, then individually wrote about one-half of the chapters and acted as reviewer for every draft of the other half. The page count suggests we again wrote almost exactly the same number of pages. Thus, we equally share the blame for what you are about to read.

Acknowledgments for the Third Edition

We'd like to again express our appreciation to **Jim Larus** 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. Our

exercise editor **Dan Sorin** took on the Herculean task of adding new exercises and answers. **Peter Ashenden** worked similarly hard to collect and organize the companion CD.

We are 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: Michael Anderson (University of Hartford), David Bader (University of New Mexico), Rusty Baldwin (Air Force Institute of Technology), John Barr (Ithaca College), Jack Briner (Charleston Southern University), Mats Brorsson (KTH, Sweden), Colin Brown (Franklin University), Lori Carter (Point Loma Nazarene University), John Casey (Northeastern University), Gene Chase (Messiah College), George Cheney (University of Massachusetts, Lowell), Daniel Citron (Jerusalem College of Technology, Israel), Albert Cohen (INRIA, France), Lloyd Dickman (PathScale), Jose Duato (Universidad Politécnica de Valencia, Spain), Ben Dugan (University of Washington), Derek Eager (University of Saskatchewan, Canada), Magnus Ekman (Chalmers University of Technology, Sweden), Ata Elahi (Southern Connecticut State University), Soundararajan Ezekiel (Indiana University of Pennsylvania), Ernest Ferguson (Northwest Missouri State University), Michael Fry (Lebanon Valley College, Pennsylvania), R. Gaede (University of Arkansas at Little Rock), Jean-Luc Gaudiot (University of California, Irvine), Thomas Gendreau (University of Wisconsin, La Crosse), George Georgiou (California State University, San Bernardino), Paul Gillard (Memorial University of Newfoundland, Canada), Joe Grimes (California Polytechnic State University, SLO), Max Hailperin (Gustavus Adolphus College), Jayantha Herath (St. Cloud State University, Minnesota), Mark Hill (University of Wisconsin, Madison), Michael Hsaio (Virginia Tech), Richard Hughey (University of California, Santa Cruz), Tony Jebara (Columbia University), Elizabeth Johnson (Xavier University), Peter Kogge (University of Notre Dame), Morris Lancaster (BAH), Doug Lawrence (University of Montana), David Lilja (University of Minnesota), Nam Ling (Santa Clara University, California), Paul Lum (Agilent Technologies), Stephen Mann (University of Waterloo, Canada), Diana Marculescu (Carnegie Mellon University), Margaret McMahon (U.S. Naval Academy Computer Science), Uwe Meyer-Baese (Florida State University), Chris Milner (University of Virginia), Tom Pittman (Southwest Baptist University), Jalel Rejeb (San Jose State University, California), Bill Siever (University of Missouri, Rolla), Kevin Skadron (University of Virginia), Pam Smallwood (Regis University, Colorado), K. Stuart Smith (Rocky Mountain College), William J. Taffe (Plymouth State University), Michael E. Thomodakis (Texas A&M University), Ruppa K. Thulasiram (University of Manitoba, Canada), Ye Tung (University of South Alabama), Steve VanderLeest (Calvin College), Neal R. Wagner (University of Texas at San Antonio), and Kent Wilken (University of California, Davis).

We are grateful too to those who carefully read our draft manuscripts; some read successive drafts to help ensure new errors didn't creep in as we revised. They include Krste Asanovic (Massachusetts Institute of Technology), Jean-Loup Baer (University of Washington), David Brooks (Harvard University), Doug Clark (Princeton University), Dan Connors (University of Colorado at Boulder), Matt Farrens (University of California, Davis), Manoj Franklin (University of Maryland College Park), John Greiner (Rice University), David Harris (Harvey Mudd College), Paul Hilfinger (University of California, Berkeley), Norm Jouppi (Hewlett-Packard), David Kaeli (Northeastern University), David Oppenheimer (University of California, Berkeley), Timothy Pinkston (University of Southern California), Mark Smotherman (Clemson University), and David Wood (University of Wisconsin, Madison).

To help us meet our goal of creating 70% new exercises and solutions for this edition, we recruited several graduate students recommended to us by their professors. We are grateful for their creativity and persistence: Michael Black (University of Maryland), Lei Chen (University of Rochester), Nirav Dave (Massachusetts Institute of Technology), Wael El Essawy (University of Rochester), Nikil Mehta (Brown University), Nicholas Nelson (University of Rochester), Aaron Smith (University of Texas, Austin), and Charlie Wang (Duke University).

We would like to especially thank **Mark Smotherman** for making a careful final pass to find technical and writing glitches that significantly improved the quality of this edition.

We wish to thank the extended Morgan Kaufmann family for agreeing to publish this book again under the able leadership of **Denise Penrose**. She developed the vision of the hybrid paper-CD book and recruited the many people above who played important roles in developing the book.

Simon Crump managed the book production process, and **Summer Block** coordinated the surveying of users and their responses. We thank also the many freelance vendors who contributed to this volume, especially **Nancy Logan** and **Dartmouth Publishing, Inc.**, our compositors.

The contributions of the nearly 100 people we mentioned here have made this third edition our best book yet. Enjoy!

David A. Patterson **John L. Hennessy**

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** 11
1.3 **Under the Covers** 15
1.4 **Real Stuff: Manufacturing Pentium 4 Chips** 28
1.5 **Fallacies and Pitfalls** 33
1.6 **Concluding Remarks** 35
1.7 **Historical Perspective and Further Reading** 36
1.8 **Exercises** 36

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. This unusual industry embraces innovation at a breathtaking rate. Since 1985 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 coast to coast 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 also 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, physics, . . .

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."

- *Automatic teller machines:* A computer placed in the wall of banks to distribute and collect cash would have been a ridiculous concept in the 1950s, when the cheapest computer cost at least $500,000 and was the size of a car.
- *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 and 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.
- *Laptop computers:* Who would have dreamed that advances in computer systems would lead to laptop computers, allowing students to bring computers to coffeehouses and on airplanes?
- *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.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the World Wide Web has transformed our society. Among its uses are distributing news, sending flowers, buying from online catalogues, taking electronic tours to help pick vacation spots, finding others who share your esoteric interests, and even more mundane topics like finding the lecture notes of the authors of your textbooks.

Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, and explain why computers are omnipresent. Tomorrow's science fiction computer applications are the cashless society, automated intelligent highways, and genuinely ubiquitous computing: no one carries computers because they are available everywhere.

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (discussed in Sections 1.3 and 1.4) 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 most readers of this book have probably used extensively. Desktop computers emphasize delivering good performance to a single user at low cost and usually are used to execute third-party software, also called shrink-wrap software. Desktop computing is one of the largest markets for computers, and the evolution of many computing technologies is driven by this class of computing, which is only about 30 years old!

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

Servers are the modern form of what was 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 often 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. As we will see in the Chapter 4, the performance of a server can be measured in several different ways, depending on the application of interest. 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.

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

Servers span the widest range in cost and capability. At the low end, a server may be little more than a desktop machine without a screen or keyboard and with a cost of a thousand dollars. These low-end servers are typically used for file storage, small business applications, or simple web serving. At the other extreme are **supercomputers**, which at the present consist of hundreds to thousands of processors, and usually gigabytes to **terabytes** of memory and terabytes to 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 are a relatively small fraction of the servers and a relatively small fraction of the overall computer market in terms of total revenue.

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.

Embedded computers are the largest class of computers and span the widest range of applications and performance. Embedded computers include the microprocessors found in your washing machine and car, the computers in a cell phone

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

or personal digital assistant, the computers in a video game or digital 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, which is 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!

Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. For example, consider a cell phone: 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 the least 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 car 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, which are used in servers, are often employed. Although this book focuses on general-purpose computers, most of the concepts apply directly, or with slight modifications, to embedded computers. In several places, we will touch on some of the unique aspects of embedded computers.

Figure 1.1 shows that during the last several years, the growth in the number of embedded computers has been much faster (40% compounded annual growth rate) than the growth rate among desktop computers and servers (9% annually). Note that the embedded computers include cell phones, video games, digital TVs and set-top boxes, personal digital assistants, and a variety of such consumer devices. Note that this data does not include low-end embedded control devices that use 8-bit and 16-bit processors.

Elaboration: Elaborations are short sections used throughout the text to provide more detail on a particular subject, which 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. The core allows a designer to integrate other application-specific hardware with the processor core for fabrication on a single chip. The availability of synthesis tools that can generate a chip from a Verilog specification, together with the capacity of modern silicon chips, has made such special-purpose processors highly attractive. Since the core can be synthesized for different semiconductor manufacturing lines, using a core provides flexibility in choosing a manufacturer as well. In the last few years, the use of cores has

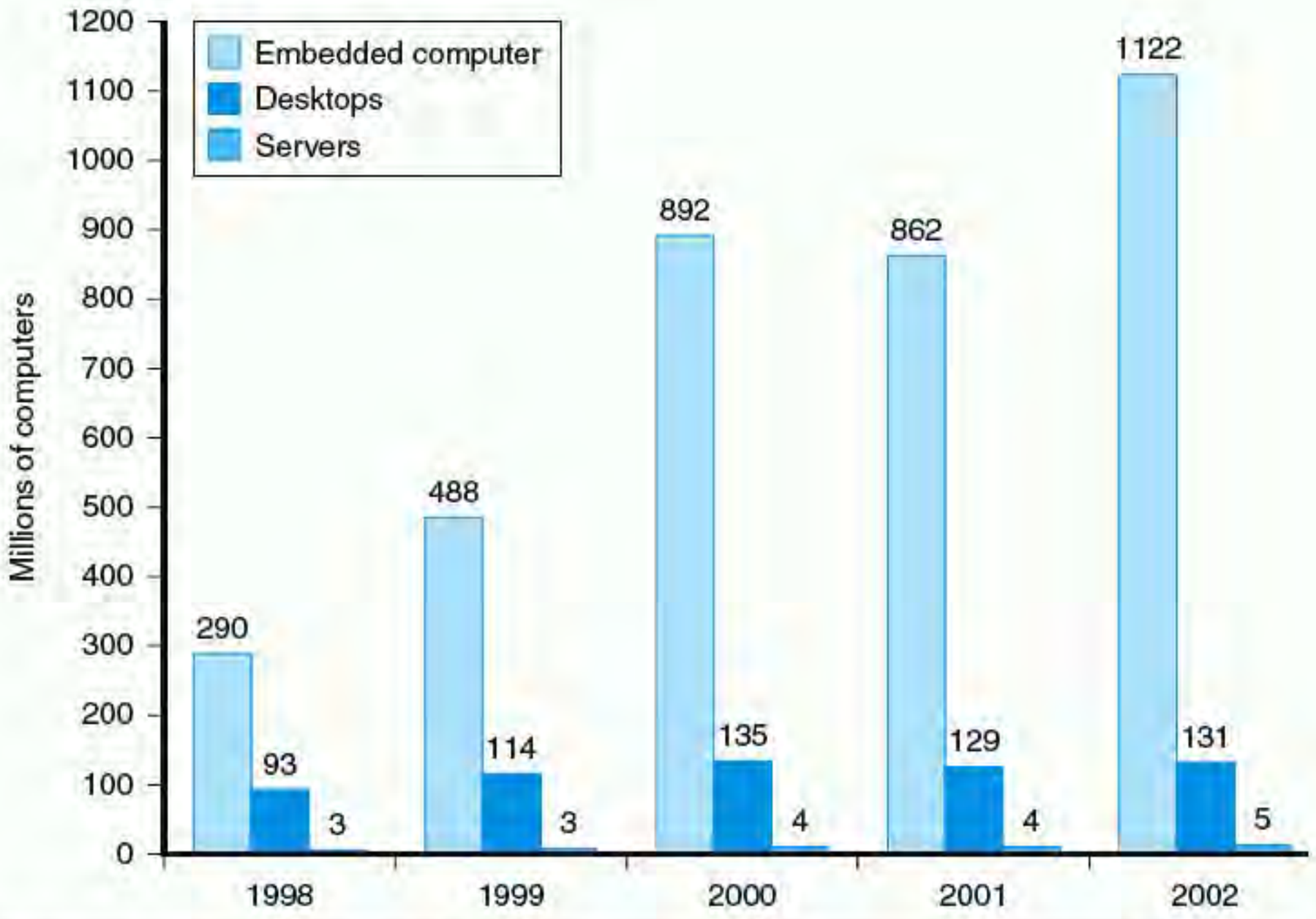


FIGURE 1.1 The number of distinct processors sold between 1998 and 2002. These counts are obtained somewhat differently, so some caution is required in interpreting the results. For example, the totals for desktops and servers count complete computer systems, because some fraction of these include multiple processors, the number of processors sold is somewhat higher, but probably by only 10–20% in total (since the servers, which may average more than one processor per system, are only about 3% of the desktop sales, which are predominantly single-processor systems). The totals for embedded computers actually count processors, many of which are not even visible, and in some cases there may be multiple processors per device.

been growing very fast. For example, in 1998 only 31% of the embedded processors were cores. By 2002, 56% of the embedded processors were cores. Furthermore, while the overall growth rate in the embedded market has been 40% per year, this growth has been primarily driven by cores, where the compounded annual growth rate has been 63%!

Figure 1.2 shows the major architectures sold in these markets with counts for each architecture, across all three types of products (embedded, desktop, and server). Only 32-bit and 64-bit processors are included, although 32-bit processors are the vast majority for most of the architectures.

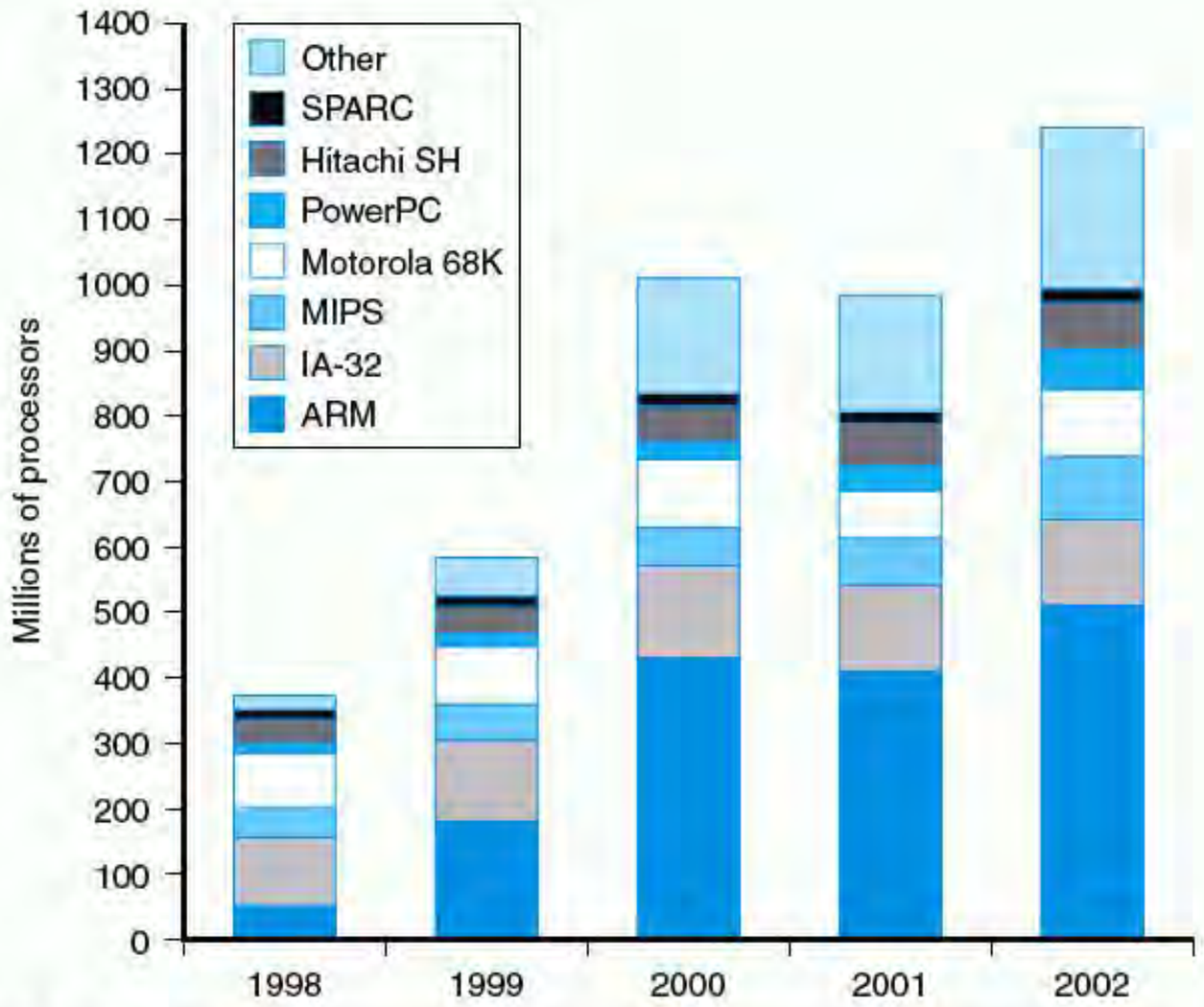


FIGURE 1.2 Sales of microprocessors between 1998 and 2002 by instruction set architecture combining all uses. The "other" category refers to processors that are either application-specific or customized architectures. In the case of ARM, roughly 80% of the sales are for cell phones, where an ARM core is used in conjunction with application-specific logic on a 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 hierarchical nature of memories and the parallel nature of processors. 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, *A Computer Architecture: A Quantitative Approach*.

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, and introduces integrated circuits, the technology that fuels the computer revolution. In this chapter, and later ones, you will likely see a lot of 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, the first time it appears in the text. After a short time of working with the terminology, you will be fluent, and your friends

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

will be impressed as you correctly use words such as BIOS, DIMM, CPU, cache, DRAM, ATA, PCI, and many others.

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, with the first one appearing below. These elements summarize important insights into program performance.

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 I/O operations. The following 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 machine instructions for each source-level statement	Chapters 2 and 3
Processor and memory system	Determines how fast instructions can be executed	Chapters 5, 6, and 7
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed	Chapter 8

Check Yourself

"Check Yourself" sections are designed to help readers assess whether they have comprehended 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

1.2 Below Your Program

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

A typical application, such as a word processor or a large database system, may consist of hundreds of thousands to 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.

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, as shown in Figure 1.3.

systems software Software that provides services that are commonly useful, including operating systems, compilers, 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 machine.

- handling basic input and output operations
- allocating storage and memory
- providing for sharing the computer among multiple applications using it simultaneously

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

Compilers perform another vital function: the translation of a program written in a high-level language, such as C or Java, into instructions that the hardware

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

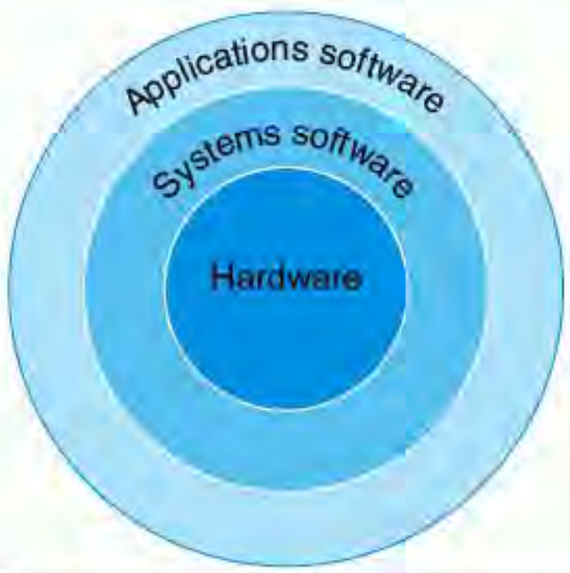


FIGURE 1.3 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications there are often multiple layers of application software as well. For example, a database system may run on top of the systems software hosting an application, which in turn runs on top of the database.

can execute. Given the sophistication of modern programming languages and the simple instructions executed by the hardware, the translation from a high-level language program to hardware instructions is complex. We will give a brief overview of the process and return to the subject in Chapter 2.

From a High-Level Language to the Language of Hardware

To actually speak to an electronic machine, you need to send electrical signals. The easiest signals for machines to understand are *on* and *off*, and so the machine 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 machine 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, 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
```

tell one computer to add two numbers. Chapter 3 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 machine to help program the machine, 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**.

assembly language A symbolic representation of machine instructions.

Although a tremendous improvement, assembly language is still far from the notation 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 machine will follow, forcing the programmer to think like the machine.

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.

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

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
```

The assembler would translate this statement into the binary instruction that tells the computer to add the two numbers `A` and `B`:

```
1000110010100000
```

Figure 1.4 shows the relationships among these programs and languages.

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 1.4). Moreover, they allow languages to

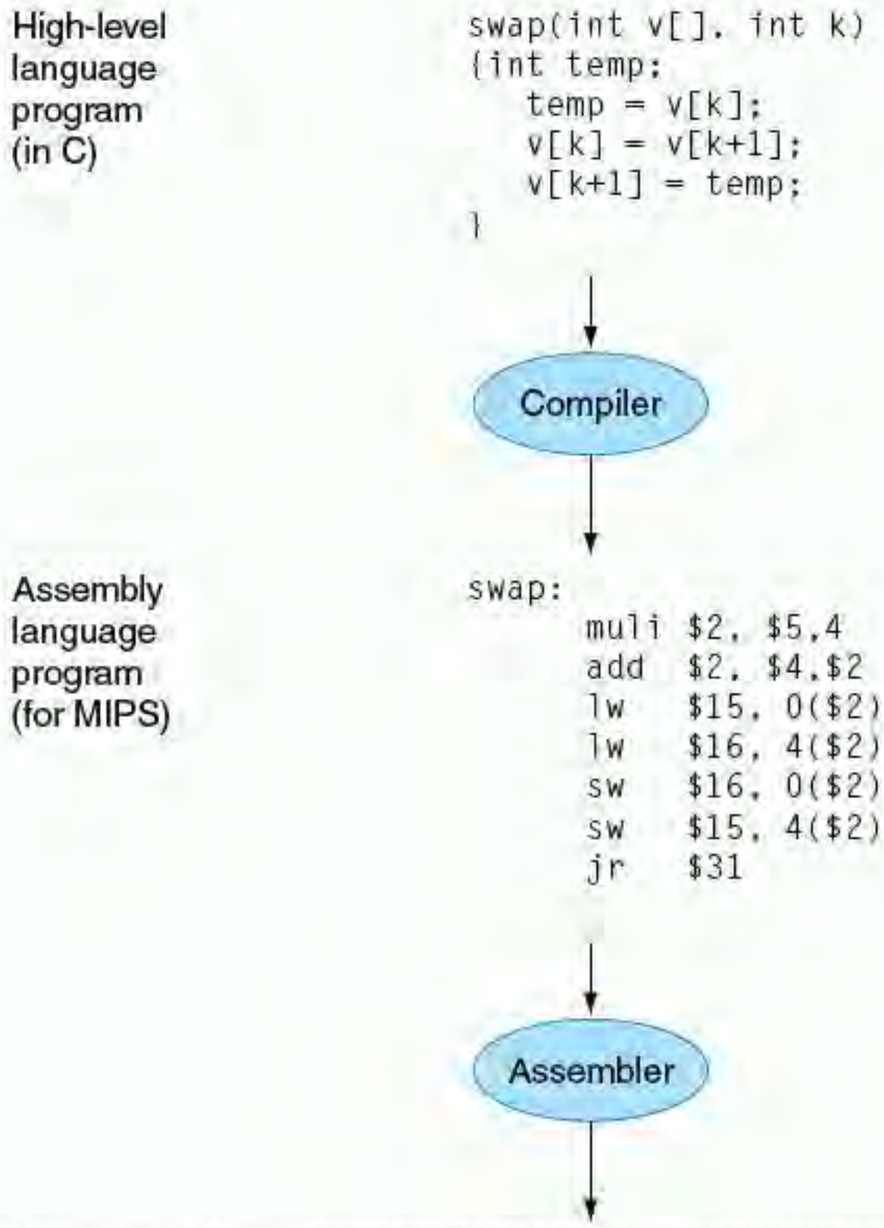


FIGURE 1.4 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in Chapter 2.

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.

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 machine. 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 the 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, with 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 1.5 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 the following chapters, with the portion of interest to that chapter highlighted.

Figure 1.6 shows a typical desktop computer with keyboard, mouse, screen, and a box containing even more hardware. What is not visible in the photograph is a network that connects the computer to other computers. 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.

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.

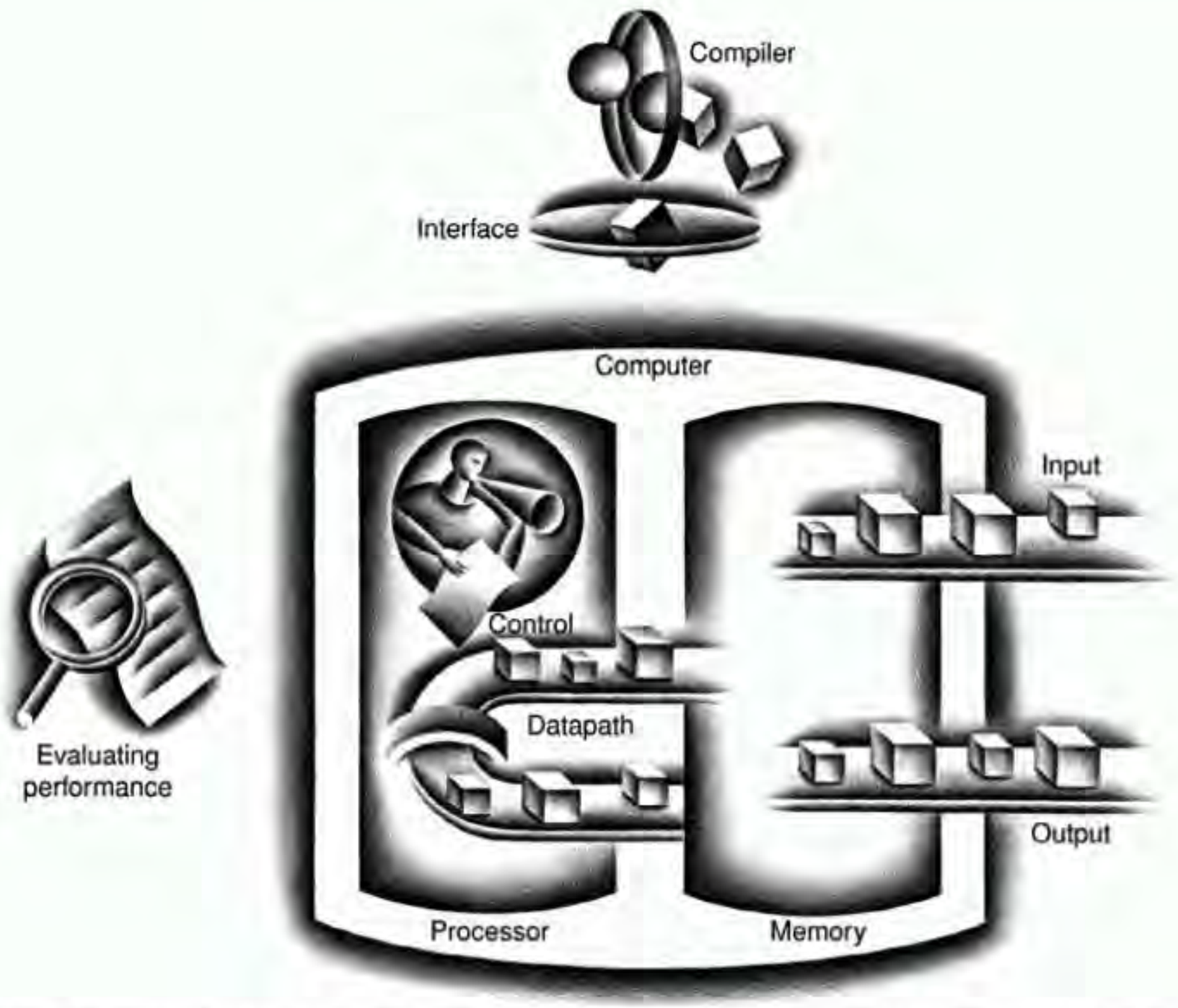


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Chapter 8 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.

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 Engelbart using a research prototype in 1967. The Alto, which was the inspiration for all workstations as well as for the Macintosh, 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.



FIGURE 1.6 A desktop computer. The liquid crystal display (LCD) screen is the primary output device, and the keyboard and mouse are the primary input devices. The box contains the processor as well as additional I/O devices. This system is a Dell Optiplex GX260.

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.

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, quoted in "Computer Software for Graphics," *Scientific American*, 1984

Through the Looking Glass

The most fascinating I/O device is probably the graphics display. Based on television technology, a **cathode ray tube (CRT) display** scans an image one line at a time, 30 to 75 times per second. At this *refresh rate*, people don't notice a flicker on the screen.

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 512×340 to 1920×1280 pixels in 2003. The simplest display has 1 bit per pixel, allowing it to be black or white. For displays that support 256 different shades of black and white, sometimes called *gray-scale* displays, 8 bits per pixel are required. 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.

All laptop and handheld computers, calculators, cellular phones, and many desktop computers use **flat-panel** or **liquid crystal displays (LCDs)** instead of CRTs to get a thin, low-power display. The main difference is that the LCD pixel 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. As in a CRT, a red-green-blue mask associated with each pixel 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 pixel.

No matter what the display, 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 on-screen is stored in the frame buffer, and the bit pattern per pixel is read out to the graphics display at the refresh rate. Figure 1.7 shows a frame buffer with 4 bits per pixel.

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. For example, when the screen is being updated, the eye can detect the inconsistency between the portion of the screen that has changed and that which hasn't.

cathode ray tube (CRT) display A display, such as a television set, that displays an image using an electron beam scanned across a screen.

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

flat panel display, 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.

Opening the Box

If we open the box containing the computer, we see a fascinating board of thin green plastic, covered with dozens of small gray or black rectangles. Figure 1.8

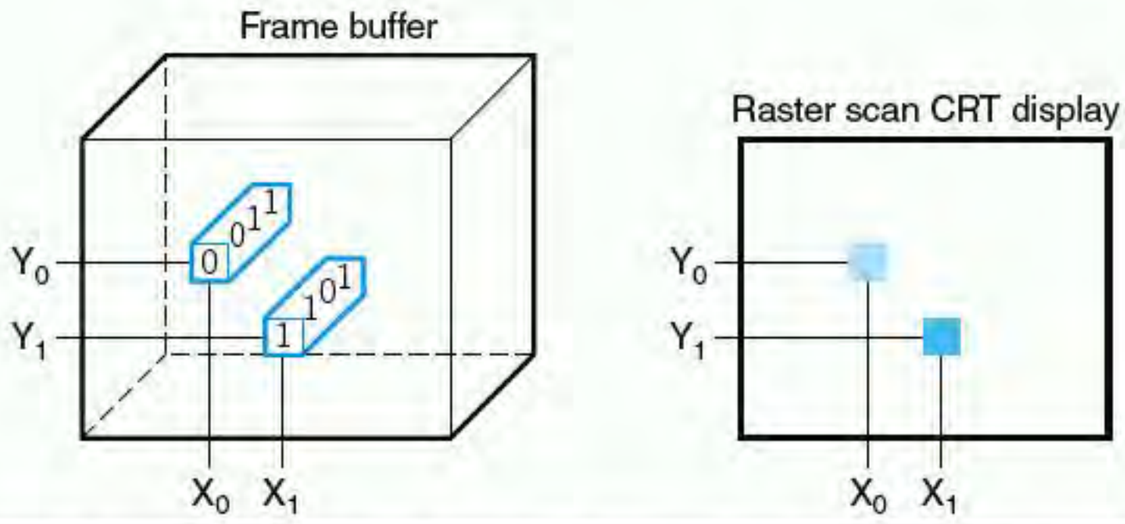


FIGURE 1.7 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade of gray on the screen than the bit pattern 1101 in pixel (X_1, Y_1).

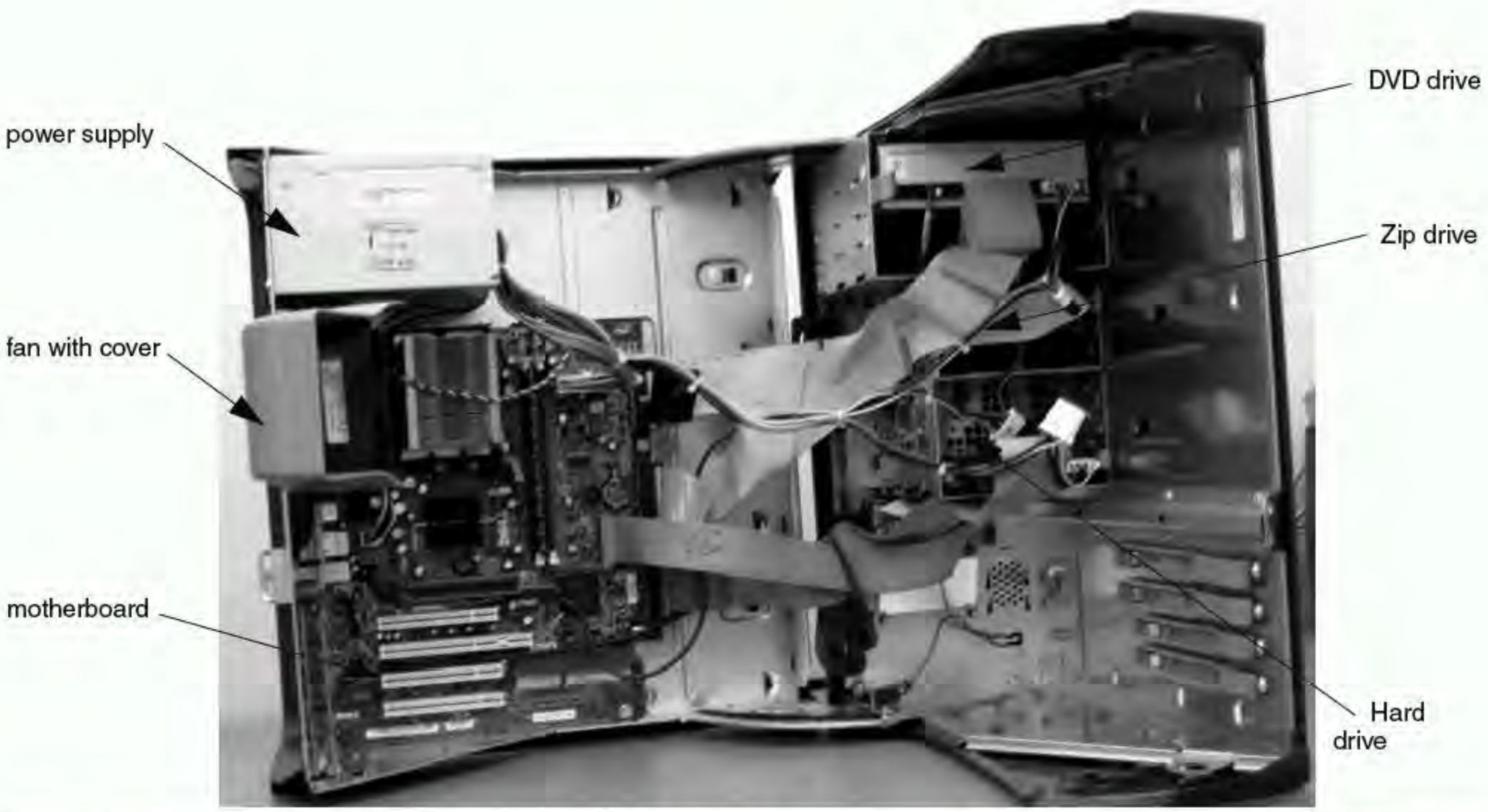


FIGURE 1.8 Inside the personal computer of Figure 1.6 on page 17. This packaging is sometimes called a clamshell because of the way it opens with hinges on one side. To see what's inside, let's start on the top left-hand side. The shiny metal box on the top far left side is the power supply. Just below that on the far left is the fan, with its cover pulled back. To the right and below the fan is a printed circuit board (PC board), called the *motherboard* in a PC, that contains most of the electronics of the computer; Figure 1.10 is a close-up of that board. The processor is the large raised rectangle just to the right of the fan. On the right side we see the bays designed to hold types of disk drives. The top bay contains a DVD drive, the middle bay a Zip drive, and the bottom bay contains a hard disk.

shows the contents of the desktop computer in Figure 1.6. This **motherboard** is shown vertically on the left with the power supply. Three disk drives—a DVD drive, Zip drive, and hard drive—appear on the right.

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.

The small rectangles on the motherboard contain the devices that drive our advancing technology, **integrated circuits** or **chips**. The board is composed of three pieces: the piece connecting to the I/O devices mentioned earlier, the memory, and the processor. The I/O devices are connected via the two large boards attached perpendicularly to the motherboard toward the middle on the right-hand side.

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

The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. In Figure 1.8, memory is found on the two small boards that are attached perpendicularly toward the middle of the motherboard. Each small memory board contains eight integrated circuits.

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

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 the large square below the memory boards in the lower-right corner of Figure 1.8. 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 the processor in Figure 1.8. The processor 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 5 explains the datapath and control for a straightforward implementation, and Chapter 6 describes the changes needed 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.

Descending into the depths of any component of the hardware reveals insights into the machine. The memory in Figure 1.10 is built from DRAM chips. *DRAM* 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 the same amount of time no matter what portion of the memory is read. 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.

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

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

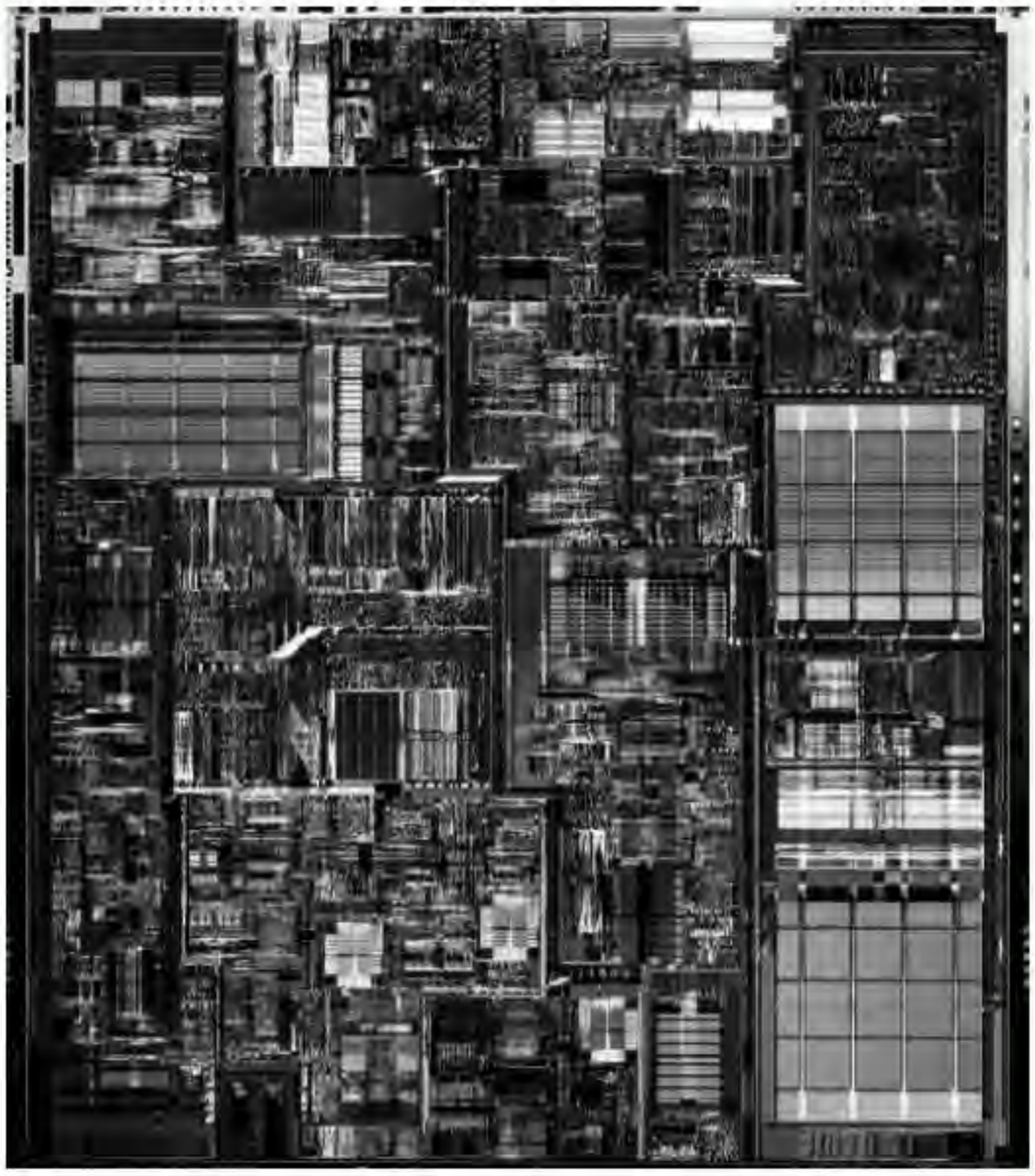


FIGURE 1.9 Inside the processor chip used on the board shown in Figure 1.8. The left-hand side is a microphotograph of the Pentium 4 processor chip, and the right-hand side shows the major blocks in the processor.

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.

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



FIGURE 1.10 Close-up of PC motherboard. This board uses the Intel Pentium 4 processor, which is located on the left-upper quadrant of the board. It is covered by a set of metal fins, which look like a radiator. This structure is the *heat sink*, used to help cool the chip. The main memory is contained on one or more small boards that are perpendicular to the motherboard near the middle. The DRAM chips are mounted on these boards (called DIMMs, for dual inline memory modules) and then plugged into the connectors. Much of the rest of the board comprises connectors for external I/O devices: audio/MIDI and parallel/serial at the right edge, two PCI card slots near the bottom, and an ATA connector used for attaching hard disks.

DIMM (dual inline memory module) A small board that contains DRAM chips on both sides. SIMMs have DRAMs on only one side. Both DIMMs and SIMMs are meant to be plugged into memory slots, usually on a motherboard.

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

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.

name: the **instruction set architecture**, or simply **architecture**, of a machine. 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.

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 cassette tape for a stereo doesn't forget the recorded music when you turn off the power because the tape is magnetic and is thus a **nonvolatile memory** technology.

To distinguish between the memory used to hold programs while they are running and this nonvolatile memory used to store programs between runs, the term **primary memory** or **main memory** is used for the former, and **secondary memory** for the latter. DRAMs have dominated main memory since 1975, but **magnetic disks** have dominated secondary memory since 1965. In embedded applications, FLASH, a nonvolatile semiconductor memory is also used.

Today the primary nonvolatile storage used on all desktop and server computers is the magnetic hard disk. As Figure 1.11 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 video tape. 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.

Diameters of hard disks vary by more than a factor of 3 today, from less than 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, the smallest disks have the lowest unit cost, and the best cost per **megabyte** is usually a disk in between. Although most hard drives appear inside computers (as in Figure 1.8), hard drives can also be attached using external interfaces such as Firewire or USB.

The use of mechanical components means that access times for magnetic disks are much slower than for DRAMs: disks typically take 5–15 milliseconds, while DRAMs take 40–80 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 2004, the cost per megabyte of disk is about 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 megabyte because they have very high storage capacity at a modest cost.

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

volatile memory Storage, such as DRAM, that only 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 and DRAM is not.

primary memory Also called main memory. Volatile 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.

megabyte Traditionally 1,048,576 (2^{20}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000 (10^6) bytes.

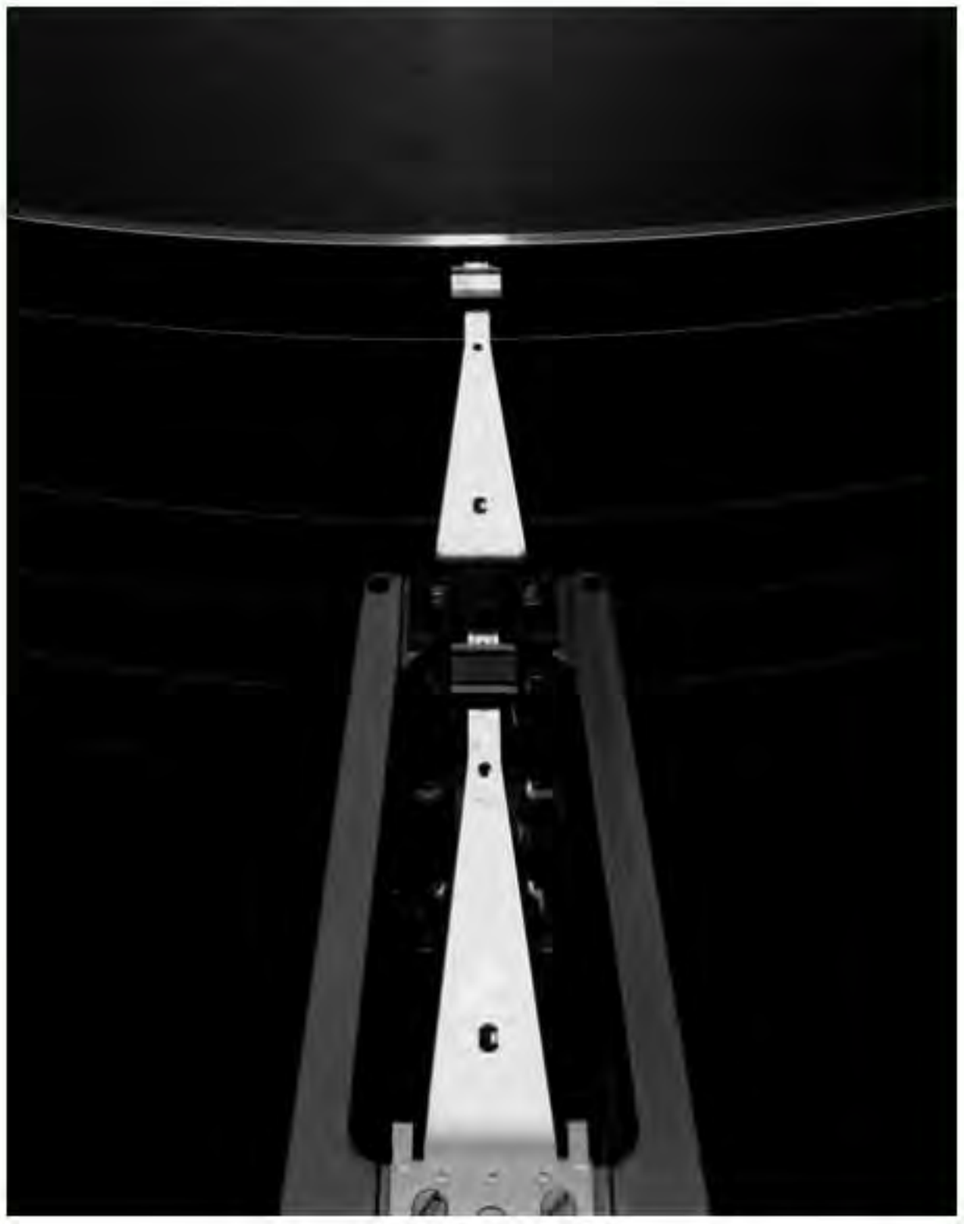


FIGURE 1.11 A disk showing 10 disk platters and the read/write heads.

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.

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.
- Magnetic tape provides only slow serial access and has been used to back up disks, in a role now often replaced by duplicate hard drives.
- FLASH-based removable memory cards typically attach by a USB (Universal Serial Bus) connection and are often used to transfer files.
- Floppy drives and Zip drives are a version of magnetic disk technology with removable flexible disks. **Floppy disks** were the original primary storage for personal computers, but have now largely vanished.

floppy disk A portable form of secondary memory composed of a rotating mylar platter coated with a magnetic recording material.

Optical disk technology works in a completely different way 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 be focused on, and the size of each bit is much smaller, which together yield a significant increase in capacity.

CD and DVD 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 tens of minutes (for a full CD) to close to an hour (for a full DVD). Thus, for large quantities a different technique called pressing is used, which costs only pennies per CD or DVD.

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 1.5 on page 16 is connected to memory and I/O devices, networks connect 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 machine 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 machine 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 the *Ethernet*. Its length is limited to about a kilometer, and the most popular version in 2004 takes about a tenth of a second to send 1 million bytes of data. 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 A network extended over hundreds of kilometers which can span a continent.

Networks have changed the face of computing in the last 25 years both by becoming much more ubiquitous and by 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. In the 1970s, 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, if not a hundred, computers. Today, local area network technology offers a capacity of from 100 million bits per second to a gigabit per second, usually shared by at most a few computers. Furthermore, 10-gigabit technology is in development! 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 the increases in capacity have made network technology central to the information revolution of the last 25 years.

Recently, another innovation in networking is reshaping the way computers communicate. Wireless technology has become widely deployed, and most 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 less than 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

1. 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 Memories

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 1.12 shows the technologies that have been used over time, with an estimate of the relative performance per unit cost for each technology. This section 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.

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

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

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 1.12 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2005 extrapolated by the authors.

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 which uses an electron beam to transfer data.

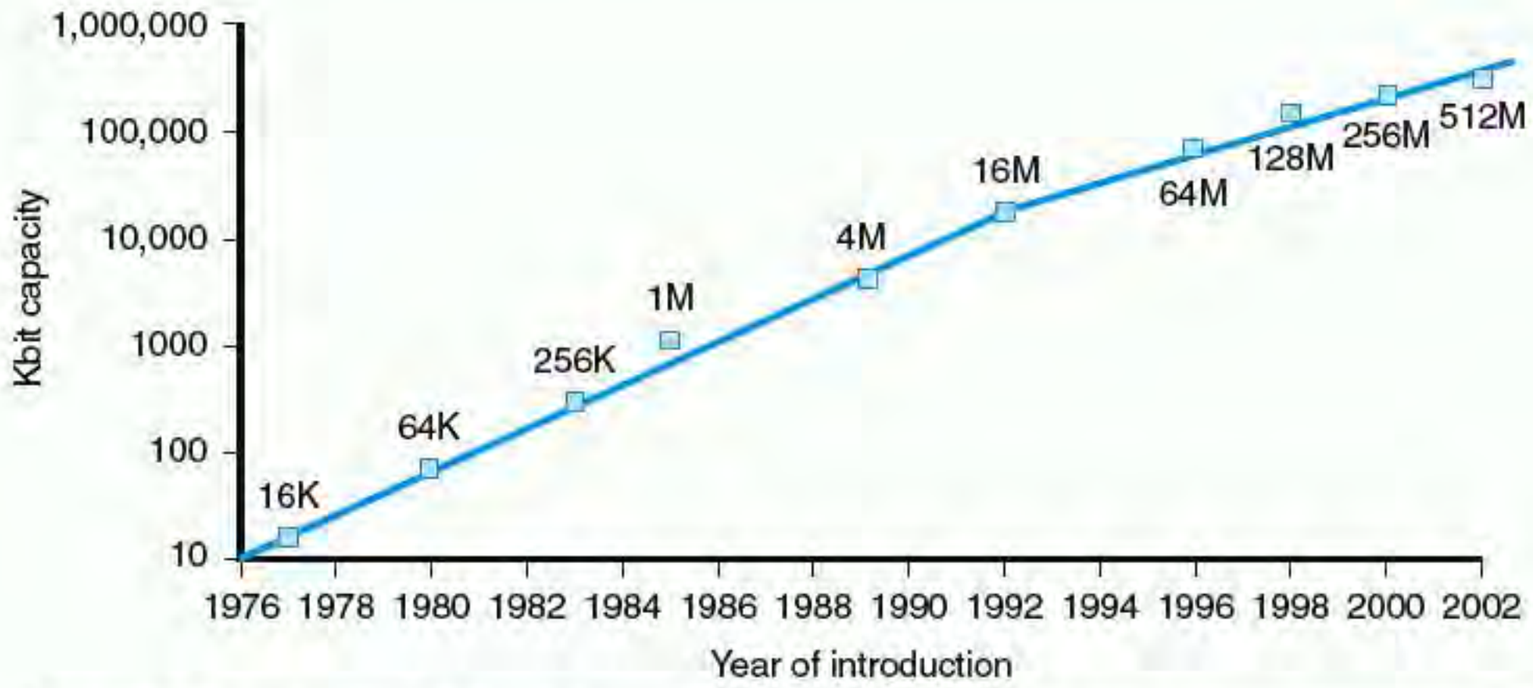


FIGURE 1.13 Growth of capacity per DRAM chip over time. The *y*-axis is measured in Kbits, where K = 1024 (2^{10}). The DRAM industry quadrupled capacity almost every 3 years, a 60% increase per year, for 20 years. This "four times every three years" estimate was called the *DRAM growth rule*. In recent years, the rate has slowed down somewhat and is somewhat closer to doubling every two years or four times every four years.

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

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 1.13 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 the manufacturing techniques. In Section 1.4, we discuss how integrated circuits are manufactured.

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

1.4 Real Stuff: Manufacturing Pentium 4 Chips

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 the IBM-compatible PC, the Apple Macintosh, a common server, or an embedded computer. For this first "Real Stuff" section, we look at how integrated circuits are manufactured, with the Pentium 4 as an 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)

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.

silicon A natural element which is a semiconductor.

semiconductor A substance that does not conduct electricity well.

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.14 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

silicon crystal ingot A rod composed of a silicon crystal that is between 6 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.

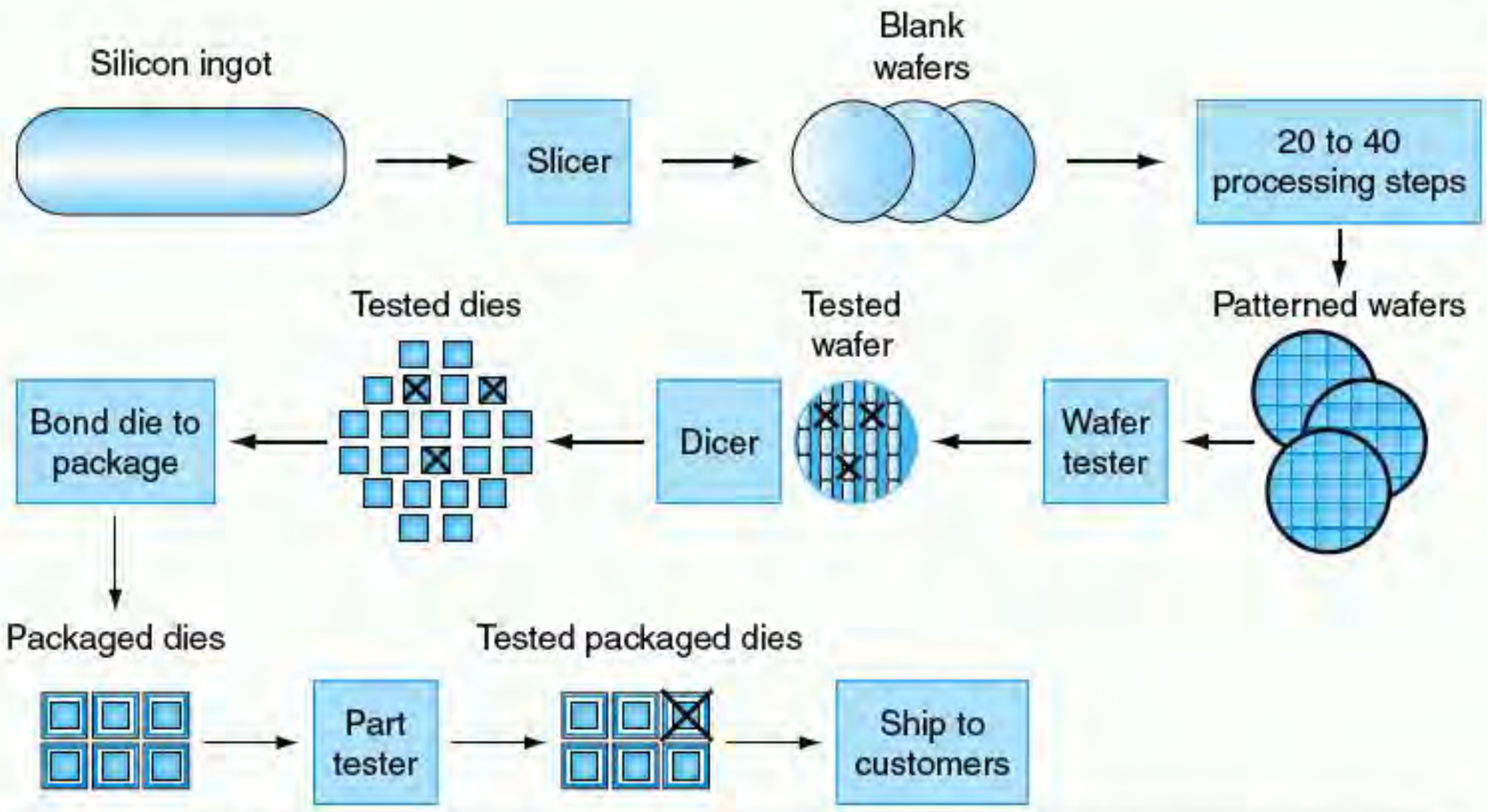


FIGURE 1.14 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.15 on page 31). These patterned wafers are then tested with a wafer tester and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9 on page 21). 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.

on 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.

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 1.15 is a photograph of a wafer containing Pentium 4 microprocessors before they have been diced; earlier, Figure 1.9 on page 21 showed an individual die of the Pentium 4 and its major components.

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.

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.

yield 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 large 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. (An **Integrated Circuit Cost** section on the CD probes these issues further.)

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.

Another 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 Intel Pentium 4 at 3.06 GHz burns 82 watts, which must be removed from a chip whose surface area is just over 1 cm^2! Figure 1.16 shows a 3.06 GHz Pentium 4 mounted on top of its heat sink, which in turn sits right next to the fan in the box shown in Figure 1.8 (on page 19)!

What determines the power consumed by an integrated circuit? Ignoring technology and circuit specifics, the power is proportional to the product of the number of transistors switched times the frequency they are switched. Thus, in general, higher clock rates or higher transistor counts lead to higher power. For example, the Intel Itanium 2 has four times the transistors of the Intel Pentium 4; although its clock rate is only one-half that of the Pentium 4, the Itanium burns 130 watts

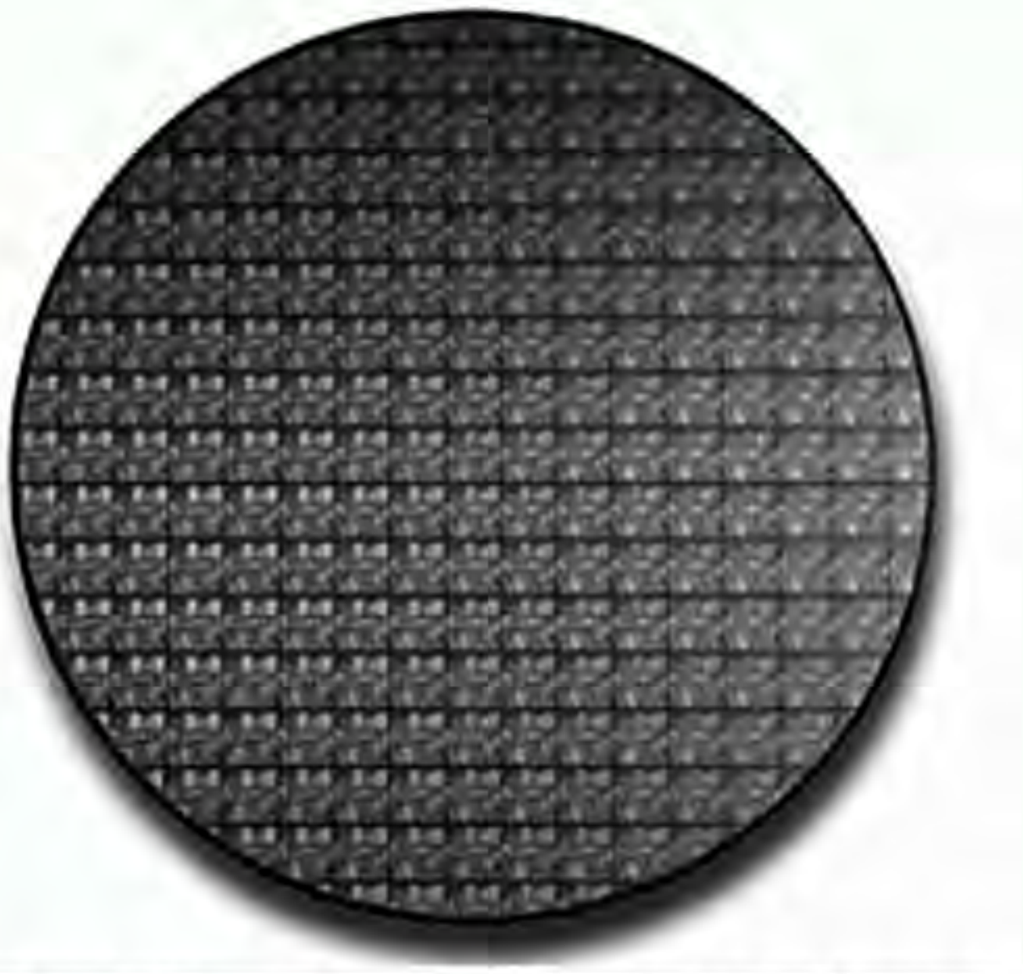


FIGURE 1.15 An 8-inch (200-mm) diameter wafer containing Intel Pentium 4 processors. The number of Pentium dies per wafer at 100% yield is 165. Figure 1.9 on page 21 is a photomicrograph of one of these Pentium 4 dies. The die area is 250 mm^2, and it contains about 55 million transistors.This die uses a 0.18 micron technology, which means that the smallest transistors are approximately 0.18 microns 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 Pentium 4 is also made using a more advanced 0.13 micron technology. 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.

compared to the 82 watts consumed by the Pentium 4. As we will see in later chapters, both performance and power consumption vary widely.

Elaboration: In CMOS (Complementary Metal Oxide Semiconductor), which is the dominant technology for integrated circuits, the primary source of power dissipation is so-called "dynamic power"—that is, power that is consumed during switching. CMOS technology, unlike earlier technologies, does not directly consume power when it is idle—hence the use of low clock rates to allow a processor to "sleep" and conserve power. The dynamic power dissipation depends on the capacitive loading of each transistor, the voltage applied, and the frequency that the transistor is switched:

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



FIGURE 1.16 An Intel Pentium 4 (3.06 GHz) mounted on top of its heat sink, which is designed to remove the 82 watts generated within the die.

Power can be reduced by lowering the voltage, which typically occurs with a new generation of technology; in 20 years, voltages have gone from 5V to 1.5V, significantly reducing power. 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.

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. In 2004, leakage is probably responsible for 20–30% of the power consumption. Thus, increasing the number of transistors increases power dissipation, even if the transistors are always off. A variety of design techniques and technology innovations have been deployed to control leakage.

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.

1.5 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 machines you may design or use.

Fallacy: Computers have been built in the same, old-fashioned way for far too long, and this antiquated model of computation is running out of steam.

For an antiquated model of computation, it surely is improving quickly. Figure 1.17 plots the top performance per year of workstations between 1987 and 2003. (Chapter 4 explains the proper way to measure performance.) The graph shows a line indicating an improvement of 1.54 per year, or doubling performance approximately every 18 months. In contrast to the statement above, computers are improving in performance faster today than at any time in their history, with over a thousandfold improvement between 1987 and 2003!

Pitfall: Ignoring the inexorable progress of hardware when planning a new machine.

Suppose you plan to introduce a machine in three years, and you claim the machine will be a terrific seller because it's three times as fast as anything available today. Unfortunately, the machine will probably sell poorly because the average

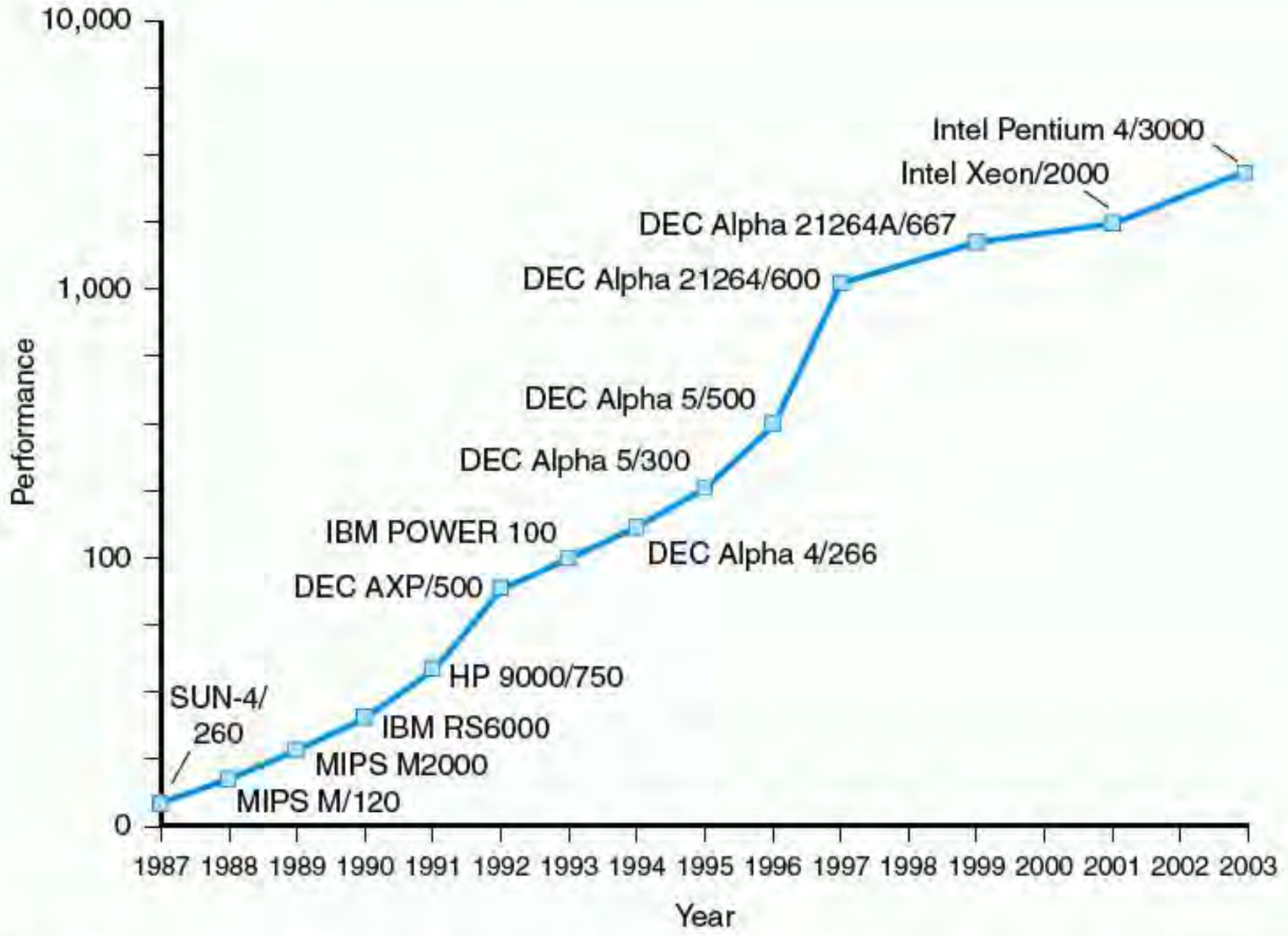


FIGURE 1.17 Performance increase of workstations, 1987–2003. Here performance is given as approximately the number of times faster than the VAX-11/780, which was a commonly used yardstick. The rate of performance improvement is between 1.5 and 1.6 times per year. These performance numbers are based on SPECint performance (see Chapter 2) and scaled over time to deal with changing benchmark sets. For processors listed with *x*/*y* after their name, *x* is the model number and *y* is the speed in megahertz.

performance growth rate for the industry will yield machines with the same performance. For example, assuming a 50% yearly growth rate in performance, a machine with performance x today can be expected to have performance $1.5^3x = 3.4x$ in three years. Your machine would have no performance advantage! Many projects within computer companies are canceled, either because they ignore this rule or because the project is completed late and the performance of the delayed machine is below the industry average. This phenomenon may occur in any industry, but rapid improvements in cost/performance make it a major concern in the computer industry.

1.6 Concluding Remarks

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 1 1/2 tons.

Popular Mechanics, March 1949

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 technology. 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.

Key technologies for modern processors are compilers and silicon. Clearly, to participate you must understand some of the characteristics of both. 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 processor, typically via pipelining, and exploiting locality of accesses to a memory hierarchy, typically via caches.

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 back to Figure 1.5). These five components also serve as the framework for the rest of the chapters in this book:

- *Datapath:* Chapters 3, 5, and 6
- *Control:* Chapters 5 and 6
- *Memory:* Chapter 7
- *Input:* Chapter 8
- *Output:* Chapter 8

Chapter 6 describes how processor pipelining exploits parallelism, and Chapter 7 describes how the memory hierarchy exploits locality. The remaining chapters provide the introduction and the conclusion to this material. Chapter 2 describes instruction sets—the interface between compilers and the machine—and emphasizes the role of compilers and programming languages in using the features of the instruction set. Chapter 3 describes how computers perform arithmetic operations and handle arithmetic data. Chapter 4 covers performance and thus describes how to evaluate the whole computer. Chapter 9 describes multiprocessors and is included on the CD. Appendix B, also on the CD, discusses 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

1.7 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 machines 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 this Section 1.7 is on the CD.

1.8 Exercises

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, <§1.2> means you should have read Section 1.3, "Under the Covers," to help you solve this exercise. If the solution to an exercise depends on others, they will be listed in curly brackets; for example, {Ex.1.1} means that you should answer Exercise 1.1 before trying this exercise.

In More Depth Exercises introduce a new topic or explore a topic in more detail. Such exercises include sufficient background to understand the concepts, as

well as exercises to explore its implication or use. The In More Depth sections appear on the CD associated with the specific chapter.

Starting in Chapter 2, you will also find For More Practice Exercises. For More Practice Exercises include additional problems intended to give the interested reader more practice in dealing with a subject. These exercises have been collected primarily from earlier editions of this book as well as exercises developed by other instructors. The For More Practice sections appear on the CD associated with the specific chapter.

Exercises 1.1 through 1.28 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	abstraction	15	embedded system
2	assembler	16	instruction
3	bit	17	instruction set architecture
4	cache	18	local area network (LAN)
5	central processor unit (CPU)	19	memory
6	chip	20	operating system
7	compiler	21	semiconductor
8	computer family	22	server
9	control	23	supercomputer
10	datapath	24	transistor
11	desktop or personal computer	25	VLSI (very large scale integrated circuit)
12	Digital Video Disk (DVD)	26	wafer
13	defect	27	wide area network (WAN)
14	DRAM (dynamic random access memory)	28	yield

1.1 [2] Active part of the computer, following the instructions of the programs to the letter. It adds numbers, tests numbers, controls other components, and so on.

1.2 [2] Approach to the design of hardware or software. The system consists of hierarchical layers, with each lower layer hiding details from the level above.

1.3 [2] Binary digit.

1.4 [2] Collection of implementations of the same instruction set architecture. They are usually made by the same company and vary in price and performance.

1.5 [2] Component of the computer where all running programs and associated data reside.

1.6 [2] Component of the processor that performs arithmetic operations.

1.7 [2] Component of the processor that tells the datapath, memory, and I/O devices what to do according to the instructions of the program.

1.8 [2] Computer designed for use by an individual, usually incorporating a graphics display, keyboard, and mouse.

1.9 [2] Computer inside another device used for running one predetermined application or collection of software.

1.10 [2] Computer used for running larger programs for multiple users often simultaneously and typically accessed only by a network.

1.11 [2] Computer network that connects a group of computers by a common transmission cable or wireless link within a small geographic area (for example, within the same floor of a building).

1.12 [2] Computer networks that connect computers spanning great distances, the backbone of the Internet.

1.13 [2] High-performance machine, costing more than $1 million.

1.14 [2] Integrated circuit commonly used to construct main memory.

1.15 [2] Microscopic flaw in a wafer.

1.16 [2] Nickname for a die or integrated circuit.

1.17 [2] On/off switch controlled by electricity.

1.18 [2] Optical storage medium with a storage capacity of more than 4.7 GB. It was initially marketed for entertainment and later for computer users.

1.19 [2] Percentage of good dies from the total number of dies on the wafer.

1.20 [2] Program that converts a symbolic version of an instruction into the binary version.

1.21 [2] Program that manages the resources of a computer for the benefit of the programs that run on that machine.

1.22 [2] Program that translates from a higher-level notation to assembly language.

1.23 [2]Technology in which single chip that contains hundreds of thousands to millions of transistors.

1.24 [2] Single software command to a processor.

1.25 [2] Small, fast memory that acts as a buffer for the main memory.

1.26 [2] Specific interface that the hardware provides the low-level software.

1.27 [2] Substance that does not conduct electricity well but is the foundation of integrated circuits.

1.28 [2] Thin disk sliced from a silicon crystal ingot, which will be later divided into dies.

Exercises 1.29 through 1.45 Using the categories in the list below, classify the following examples. Use the letters to the left of the words in the answer. Unlike the previous exercises, answers in this group may be used more than once.

a	applications software	f	personal computer
b	high-level programming language	g	semiconductor
c	input device	h	supercomputer
d	integrated circuit	i	systems software
e	output device		

1.29 [1] Assembler

1.30 [1] C++

1.31 [1] Liquid crystal display (LCD)

1.32 [1] Compiler

1.33 [1] Cray-1

1.34 [1] DRAM

1.35 [1] IBM PC

1.36 [1] Java

1.37 [1] Scanner

1.38 [1] Macintosh

1.39 [1] Microprocessor

1.40 [1] Microsoft Word

1.41 [1] Mouse

1.42 [1] Operating system

1.43 [1] Printer

1.44 [1] Silicon

1.45 [1] Spreadsheet

1.46 [15] <§1.3> In a magnetic disk, the disks containing the data are constantly rotating. On average it should take half a revolution for the desired data on the disk to spin under the read/write head. Assuming that the disk is rotating at 7200 revolutions per minute (RPM), what is the average time for the data to rotate under the disk head? What is the average time if the disk is spinning at 10,000 revolutions per minute?

1.47 [5] <§1.3> A DVD drive, however, works in the Constant Linear Velocity (CLV) mode. The read head must interact with the concentric circles at a constant rate, whether it is accessing data from the inner or outermost portions of the disc. This is affected by varying the rotation speed of the disc, from 1600 RPM at the center, to 570 RPM at the outside. Assuming that the DVD drive reads 1.35 MB of user data per second, how many bytes can the center circle store? How many bytes can the outside circle store?

1.48 [5] <§1.3> If a computer issues 30 network requests per second and each request is on average 64 KB, will a 100 Mbit Ethernet link be sufficient?

1.49 [5] <§1.3>What kinds of networks do you use on a regular basis? What kinds of media do they use? How much bandwidth do they provide?

1.50 [15] <§1.3> End-to-end delay is an important performance metric for networks. It is the time between the point when the source starts to send data and the point when the data is completely delivered to the destination. Consider two hosts A and B, connected by a single link of rate R bps. Suppose the two hosts are separated by m meters, and suppose the propagation speed along the link is s m/sec. Host A is sending a file of size L bits to host B.

a. Obtain an expression for the end-to-end delay in terms of R, L, m, and s.

b. Suppose there is a router between A and B, and the data from A must be forwarded to B by the router. If the forwarding process takes t sec, then what is the end-to-end delay?

c. Suppose the router is configured to provide QoS (Quality of Service) control for different kinds of data. If the data is a multimedia stream, such as video conference data, it will forward it at a shorter delay of $t/2$ sec. For other kinds of data, the delay is t sec. If host A is sending a multimedia stream of size $2L$, what is the end-to-end delay?

1.51 [15] <§§1.4, 1.5> Assume you are in a company that will market a certain IC chip. The fixed costs, including R&D, fabrication and equipments, and so on, add up to $500,000. The cost per wafer is $6000, and each wafer can be diced into 1500 dies. The die yield is 50%. Finally, the dies are packaged and tested, with a cost of $10 per chip. The test yield is 90%; only those that pass the test will be sold to customers. If the retail price is 40% more than the cost, at least how many chips have to be sold to break even?

1.52 [8] <§1.6> In this exercise, you will evaluate the performance difference between two CPU architectures, CISC (Complex Instruction Set Computing) and RISC (Reduced Instruction Set Computing). Generally speaking, CISC CPUs have more complex instructions than RISC CPUs and therefore need fewer instructions to perform the same tasks. However, typically one CISC instruction, since it is more complex, takes more time to complete than a RISC instruction. Assume that a certain task needs P CISC instructions and $2P$ RISC instructions, and that one CISC instruction takes $8T$ ns to complete, and one RISC instruction takes $2T$ ns. Under this assumption, which one has the better performance?

1.53 [15] <§§1.3, 1.6> Suppose there are five computers connected together to form a local area network. The maximum data transport rate (bandwidth) that the network cable can provide is 10 Mbps. If we use a low-end device (Hub) to connect them, all the computers in the network share the 10 Mbps bandwidth. If we use a high-end device (Switch), then any two of the computers can communicate with each other without disturbing the other computers. If you want to download a 10 MB file from a remote server, which is located outside your local network, how long will it take if using a Hub? How long will it take if using a Switch? Assume the other four computers only communicate with each other, and each has a constant data rate of 2 Mbps.

1.54 [8] <§1.6> Sometimes software optimization can dramatically improve the performance of a computer system. Assume that a CPU can perform a multiplication operation in 10 ns, and a subtraction operation in 1 ns. How long will it take for the CPU to calculate the result of $d = a \times b - a \times c$? Could you optimize the equation so that it will take less time?

1.55 [8] <§§1.1–1.5> This book covers abstractions for computer systems at many different levels of detail. Pick another system with which you are familiar and write one or two paragraphs describing some of the many different levels of abstraction inherent in that system. Some possibilities include automobiles, homes, airplanes, geometry, the economy, a city, and the government. Be sure to identify both high-level and low-level abstractions.

1.56 [15] <§§1.1–1.5> A less technically inclined friend has asked you to explain how computers work. Write a detailed, one-page description for your friend.

1.57 [10] <§§1.1–1.5> In what ways do you lack a clear understanding of how computers work? Are there levels of abstraction with which you are particularly unfamiliar? Are there levels of abstraction with which you are familiar but still have specific questions about? Write at least one paragraph addressing each of these questions.

1.58 [15] <§1.3> In this exercise, you will learn more about interfaces or abstractions. For example, we can provide an abstraction for a disk like this:

Performance characteristics:

- Capacity (how much data can it store?)
- Bandwidth (how fast can data be transferred between the computer and disk?)
- Latency (how long does it take to find a specific position for access?)

Functions the interface provides:

- Read/write data
- Seek to a specific position
- Status report (is the disk ready to read/write, etc.?)

Following this pattern, please provide an abstraction for a network card.

1.59 [5] <§§1.4, 1.5> **In More Depth:** Integrated Circuit Cost

1.60 [15] <§§1.4, 1.5> **In More Depth:** Integrated Circuit Cost

1.61 [10] <§§1.4, 1.5> **In More Depth:** Integrated Circuit Cost

1.62 [5] <§§1.4, 1.5> **In More Depth:** Integrated Circuit Cost

1.63 [10] <§§1.4, 1.5> **In More Depth:** Integrated Circuit Cost

1.64 [10] <§§1.4, 1.5> **In More Depth:** Integrated Circuit Cost

Answers to Check Yourself

§1.1, page 10: Discussion questions: lots of answers are acceptable.
§1.3, page 27: Disk memory: nonvolatile, long access time (milliseconds), and cost $2–4/GB. Semiconductor memory: volatile, short access time (nanoseconds), and cost $200–400/GB.
§1.4, page 33: 1, 3, and 4 are valid reasons.

Computers in the Real World

Information Technology for the 4 Billion without IT

Throughout this book you will see sections entitled "Computers in the Real World." These sections describe compelling uses for computers outside of their typical functions in office automation and data processing. The goal of these sections is to illustrate the diversity of uses for information technology.

Problem to solve: Make information technology available to the rest of humanity, such as farmers in rural villages, beyond a multilingual character set like Unicode.

Solution: Develop a computer, software, and a communication system for a rural farming village. However, there is no electricity, no telephone, no technical support, and the villagers do not read English.

The Jhai Foundation took on this challenge for five villages in the Hin Heup district of Laos. This American-Lao foundation was founded to raise the standard of living for rural Laos by developing an export economy. It also built schools, installed wells, and started a weaving cooperative. When asked what they wanted next, villagers said they wanted access to the Internet! First, they wanted to learn the prices before taking their crops to the nearest market, which is 35 kilometers away. They could also learn about the market abroad to make better decisions on what crops to grow and to increase their bargaining power when it was time to sell them. Second, they wanted to use Internet telephony to talk to relatives in Laos and beyond.

The goal was "a rugged computer and printer assembled from off-the-shelf components that draws less than 20 watts in normal use—less than 70 watts when the printer is printing—and that can survive dirt, heat, and immersion in water."

The resulting Jhai PC design uses flash memory instead of a disk drive, thereby eliminating moving parts from the PC to make it more rugged and easier to maintain. Rather than use a power-hungry cathode ray tube, it has a liquid-crystal display. To lower costs and power, it uses an 80486 microprocessor. The power is supplied by a car battery, which can be charged by a turning bicycle crank. An old



A Laotian villager who wanted access to the Internet.

dot matrix printer completes the hardware, bringing the cost to about $400. The operating system is Linux, and the applications are accounting, email, and letter writing, which expatriates are tailoring to the Lao language.

The communication solution is to adapt the WiFi (IEEE 802.11b) wireless network (see Chapter 8). The plan is to boost the signal using larger antennas and then place repeater stations on the hilltops between the village and the market city. These repeaters get their power from solar cells. The local phone system ties to it at the far end, which completes the connection to the Internet. Twenty-five volunteers in Silicon Valley are developing this Jhai PC network.

An alternative attempt is the *simputer*, which stands for "simple, inexpensive, multilingual computer." Indian computer scientists designed this personal digital assistant, which is similar to the Palm Pilot, to meet the needs of villagers in third world countries. Input is through a touch screen and speech recognition so that people need not be able to write to use it. It uses three AAA batteries, which last 3 to 4 hours. The cost is $250, and there is no special solution for communication. It's unclear whether villagers in the developing world would spend $250 on a PDA, where even batteries are a luxury.

To learn more see these references on the ◎ library

"Making the Web world-wide," *The Economist*, September 26, 2002, www.jhai.org/economist

The Jhai Foundation, www.jhai.org/

"Computers for the Third World," *Scientific American*, October 2002



Indian villager using the Simputer.

2

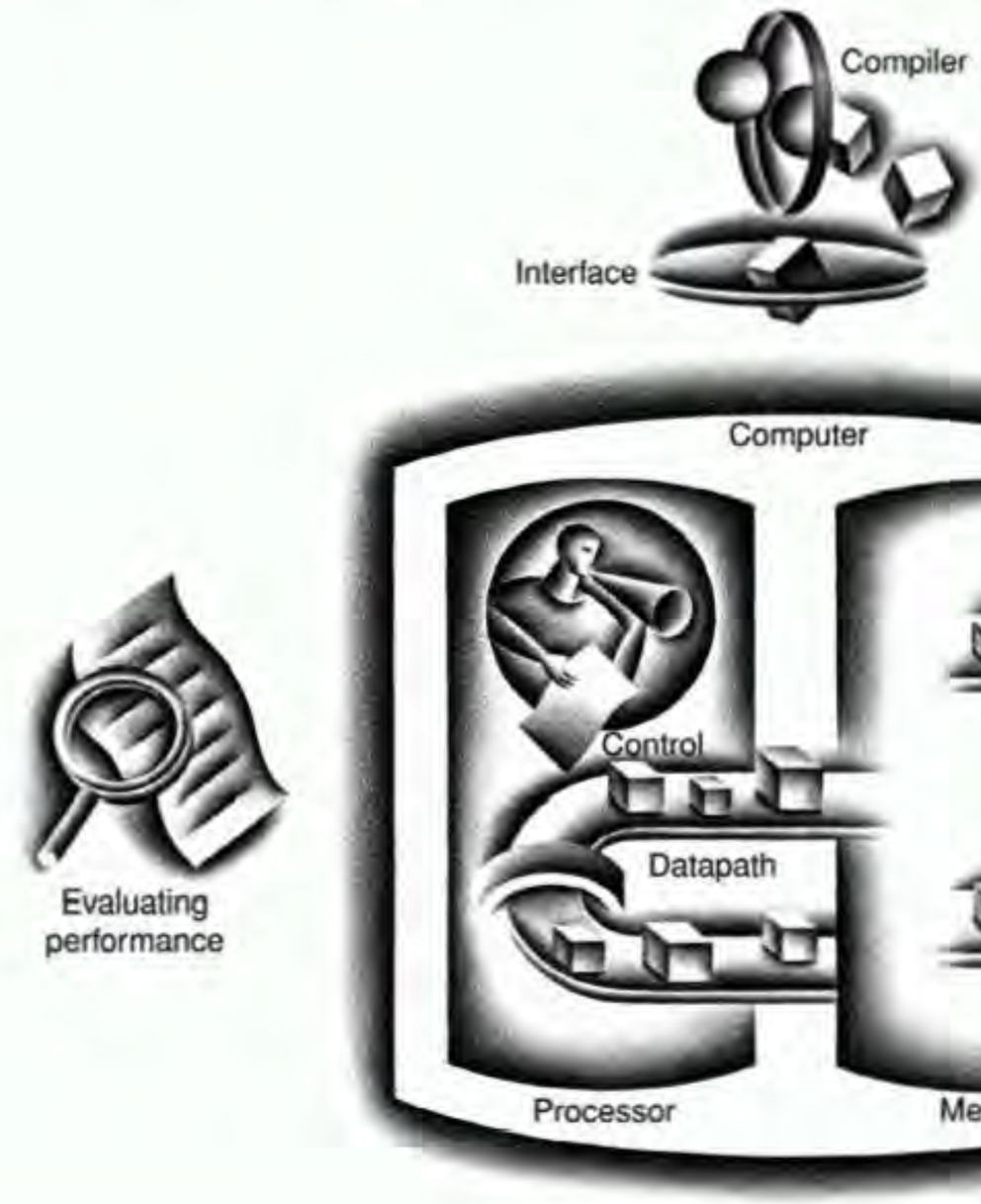
Instructions: Language of the Computer

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

Charles V, King of France
1337–1380

2.1 **Introduction** 48
2.2 **Operations of the Computer Hardware** 49
2.3 **Operands of the Computer Hardware** 52
2.4 **Representing Instructions in the Computer** 60
2.5 **Logical Operations** 68
2.6 **Instructions for Making Decisions** 72
2.7 **Supporting Procedures in Computer Hardware** 79
2.8 **Communicating with People** 90
2.9 **MIPS Addressing for 32-Bit Immediates and Addresses** 95
2.10 **Translating and Starting a Program** 106
2.11 **How Compilers Optimize** 116
2.12 **How Compilers Work: An Introduction** 121
2.13 **A C Sort Example to Put It All Together** 121
2.14 **Implementing an Object-Oriented Language** 130
2.15 **Arrays versus Pointers** 130
2.16 **Real Stuff: IA-32 Instructions** 134
2.17 **Fallacies and Pitfalls** 143
2.18 **Concluding Remarks** 145
2.19 **Historical Perspective and Further Reading** 147
2.20 **Exercises** 147

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 humans 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 representation of integer and floating-point numbers and the hardware that operates on them.

You might think that the languages of computers would be as diverse as those of humans, 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. 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 computers of the 2000s 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 2.14 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, which is typical of instruction sets designed since the 1980s. Almost 100 million of these popular microprocessors were manufactured in 2002, and they are found in products from ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Texas Instruments, and Toshiba, among others.

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 assembly language more palatable. To keep the overall picture in mind, each section ends with a figure summarizing the MIPS instruction set revealed thus far, highlighting the portions presented in that section.

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 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 take the sum of 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.

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.

EXAMPLE

Compiling Two C Assignment Statements into MIPS

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 `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)
```

Figure 2.1 summarizes the portions of MIPS assembly language described in this section. These instructions are symbolic representations of what the MIPS processor actually understands. In the next few sections, we will evolve this symbolic representation into the real language of MIPS, with each step making the symbolic representation more concrete.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add a,b,c`	a = b + c	Always three operands
	subtract	`sub a,b,c`	a = b - c	Always three operands

FIGURE 2.1 MIPS architecture revealed in Section 2.2. The real computer operands will be unveiled in the next section. Highlighted portions in such summaries show MIPS assembly language structures introduced in this section; for this first figure, all is new.

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*, 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 2.10 shows how JITs are used later than C compilers in the startup process, and Section 2.13 shows the performance consequences of compiling versus interpreting Java programs. The Java examples in this chapter skip the Java bytecode step and just show the MIPS code that are produced by a compiler.

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 the bricks of computer construction: registers are primitives used in hardware design that are also visible to the programmer when the computer is completed. 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. MIPS has 32 registers. (See Section 2.19 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 2.4 demonstrates.

Chapters 5 and 6 show the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is key 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 2.7 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.

Compiling a C Assignment Using Registers

EXAMPLE

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 depicted on page 47. The processor can keep only a small amount of data in registers, but computer memory contains millions of data elements. Hence, data structures (arrays and structures) are kept in memory.

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.

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.

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*.

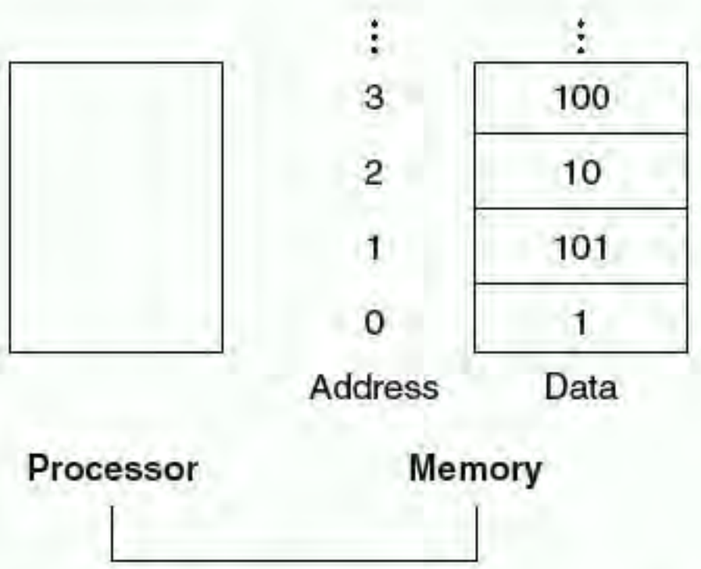


FIGURE 2.2 Memory addresses and contents of memory at those locations. This is a simplification of the MIPS addressing; Figure 2.3 shows the actual MIPS addressing for sequential word addresses in memory.

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 is called the *offset*, and the register added to form the address 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. Hence, addresses of sequential words differ by 4. For example, Figure 2.3 shows the actual MIPS addresses for Figure 2.2; the byte address of the third word is 8.

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 5 suggests why alignment leads to faster data transfers.)

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

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 A, page A-43, 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 of page 144 of Section 2.17.)

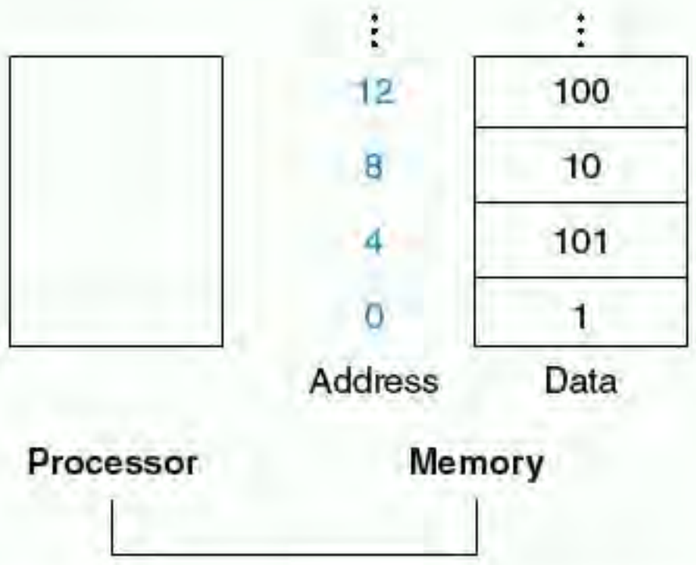


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of four: there are four bytes in a word.

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 as the offset and register `$s3` as the base register.

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

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 SPEC2000 benchmarks.

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 registers are smaller. 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, MIPS registers take both less time to access *and* have higher throughput than memory—a rare combination—making data in registers both faster to access and simpler to use. To achieve highest performance, compilers must use registers efficiently.

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 `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, they are much faster than if constants were loaded from memory.

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, . . . , $t0, $t1, . . .`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. 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 operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three operands; data in registers
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory

FIGURE 2.4 MIPS architecture revealed through Section 2.3. Highlighted portions show MIPS assembly language structures introduced in Section 2.3.

Figure 2.4 summarizes the portions of the symbolic representation of the MIPS instruction set described in this section. Load word and store word are the instructions that copy words between memory and registers in the MIPS architecture. Other brands of computers use instructions along with load and store to transfer data. An architecture with such alternatives is the Intel IA-32, described in Section 2.16.

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 D 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.

Section 2.4 explains that since MIPS supports negative constants, there is no need for subtract immediate in MIPS.

2.4 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. 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 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.

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

Instructions are also 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 part of 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

To distinguish it from assembly language, we call the numeric version of instructions **machine language** and a sequence of such instructions *machine code*.

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.

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 con-

machine language Binary representation used for communication within a computer system.

instruction format A form of representation of an instruction composed of fields of binary numbers.

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 2.5 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.

hexadecimal Numbers in base 16.

verts 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 2.5 converts hexadecimal to binary, and vice versa.

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

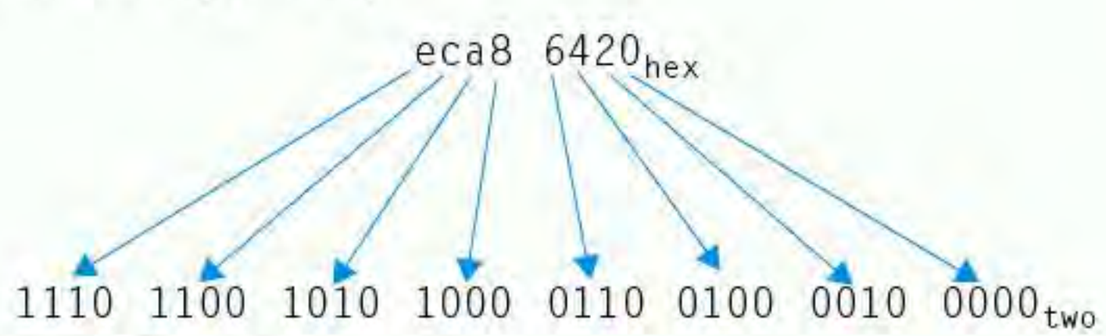
Binary-to-Hexadecimal and Back

Convert the following hexadecimal and binary numbers into the other base:
$\texttt{eca8 6420}_{hex}$

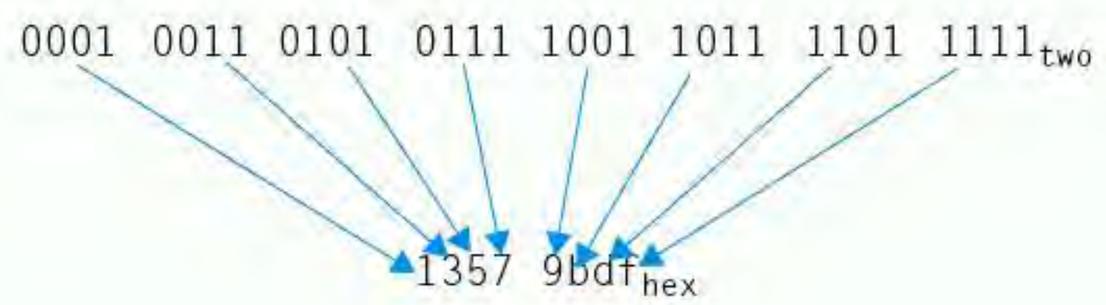
ANSWER

$$\texttt{0001 0011 0101 0111 1001 1011 1101 1111}_{two}$$

Just a table lookup one way:



And then the other direction too:



MIPS Fields

MIPS fields are given names to make them easier to discuss:



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 2.5 explains shift instructions and this term; it will not be used until then, and hence the field contains zero.)
- *funct:* Function. This field selects the specific variant of the operation in the op field and is sometimes called the *function code.*

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



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 $\pm 2^{15}$. (Chapter 3 explains how to represent negative numbers.) 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 57:

```
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 fourth field in I-type is equal to the length 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 2.6 shows the numbers used in each field for the MIPS instructions covered through Section 2.3.

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 2.6 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).

Translating MIPS Assembly Language into Machine Language

EXAMPLE

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 2.6, 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 2.6) 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 × 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.

The binary equivalent to the decimal form is the following (1200 in base 10 is 0000 0100 1011 0000 base 2):

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.

Figure 2.7 summarizes the portions of MIPS assembly language described in this section. As we shall see in Chapters 5 and 6, the similarity of the binary representations of related instructions simplifies hardware design. These instructions are another example of regularity in the MIPS architecture.

Check Yourself

Why doesn't MIPS have a subtract immediate instruction?

1. Negative constants appear much less frequently in C and Java, so they are not the common case and do not merit special support.
2. Since the immediate field holds both negative and positive constants, add immediate with a negative number is equivalent to subtract immediate with a positive number, so subtract immediate is superfluous.

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 2.8 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.

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, ..., $s7` `$t0, $t1, ..., $t7`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers `$s0-$s7` map to 16–23 and `$t0-$t7` map to 8–15.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions in MIPS. 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 operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory

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 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.7 MIPS architecture revealed through Section 2.4. Highlighted portions show MIPS machine language structures introduced in Section 2.4. 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; *shamt* field, which Section 2.5 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format keeps the last 16 bits as a single *address* field.

Elaboration: Representing decimal numbers in base 2 gives an easy way to represent positive integers in computer words. Chapter 3 explains how to represent negative numbers, but for now take it on faith that a 32-bit word can represent integers between -2^{31} and $+2^{31} - 1$ or –2,147,483,648 to +2,147,483,647, and the 16-bit constant field really holds -2^{15} to $+2^{15} - 1$ or –32,768 to 32,767. Such integers are called *two's complement* numbers. Chapter 3 shows how we would encode `addi $t0,$t0,-1` or `lw $t0, -4($s0)`, which require negative numbers in the constant field of the immediate format.

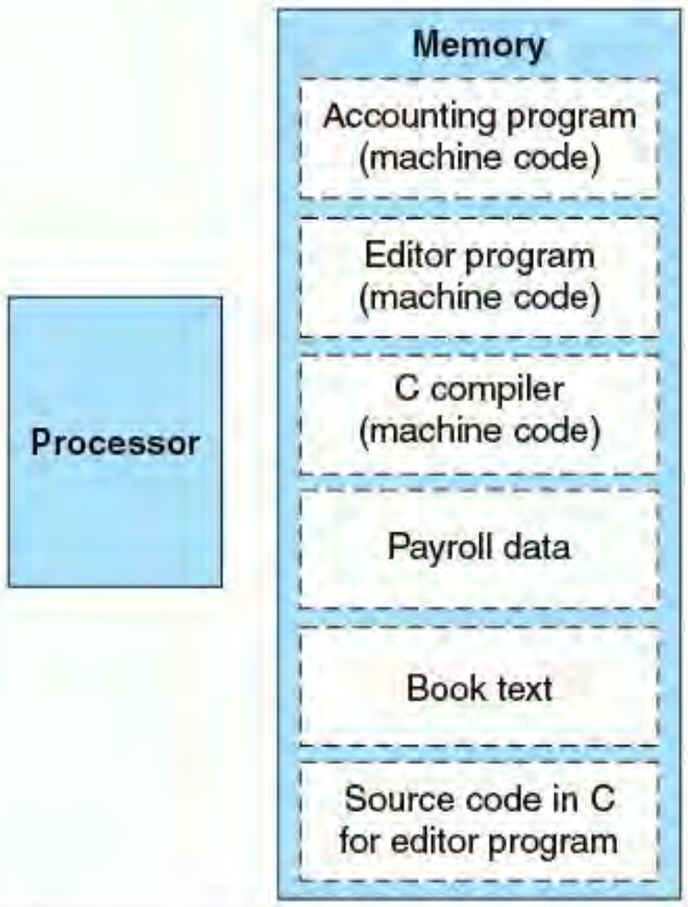


FIGURE 2.8 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given 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.

"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

2.5 Logical Operations

Although the first computers concentrated 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 are stored as 8 bits, is one example of such an operation. It follows that instructions were added to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.9 shows logical operations in C and Java.

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 2.9 C and Java logical operators and their corresponding MIPS instructions.

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 look like this:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

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 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. It stands for *shift amount* and is used in shift instructions. 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 `$t2`, rt contains `$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 (Chapter 3 explains why). 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` still contains

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0000\ 0000_{two}$$

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.

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`:

$$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 0000\ 0000_{two}$$

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.

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 two-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. For example, 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 2.9 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 hardware has no immediate version. Figure 2.10, which summarizes the MIPS instructions seen thus far, highlights the logical instructions.

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, ..., $s7` `$t0, $t1, ..., $t7`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers `$s0-$s7` map to 16–23 and `$t0-$t7` map to 8–15.
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 operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow detected
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 $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	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
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory

FIGURE 2.10 MIPS architecture revealed thus far. Color indicates the portions introduced since Figure 2.7 on page 67. The back endpapers of this book also list the MIPS machine language.

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. When the iteration is completed a different sequence of [instructions] is to be followed, so we must, in most cases, give two parallel trains of [instructions] preceded by an instruction as to which routine is to be followed. 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

2.6 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**.

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 2.11 is a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want `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):.

```
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)
```

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.

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:
```

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 2.10).

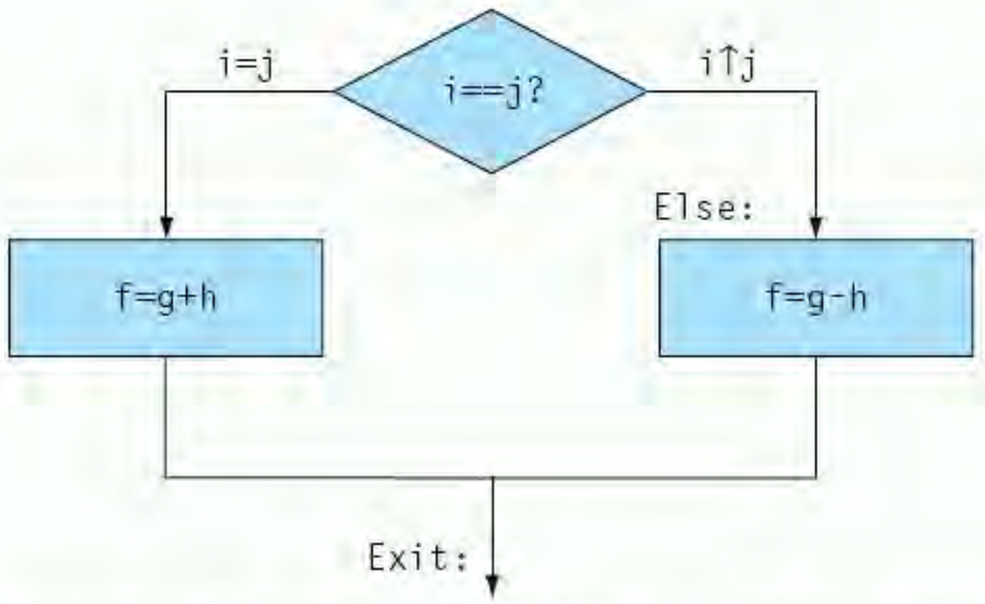


FIGURE 2.11 Illustration of the options in the *if* statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

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 4 (see page 69 in Section 2.5). 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 = 4 * i
```

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`:

```
add  $s3,$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 Exercise 2.33 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
```

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. Since register `$zero` always has 0, we can already compare to 0. To compare to other values, there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
slti    $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

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

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal. (As you might expect, register `$zero` maps to register 0.)

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.

jump address table Also called jump table. A table of addresses of alternative instruction sequences.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address 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. See the In More Depth exercises in Section 2.20 for more details on jump address tables.

To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. The program loads the appropriate entry from the jump table into a register, and then it jumps to the proper address using a jump register. This instruction is described in Section 2.7.

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 next lower level is the conditional branch.

Figure 2.12 summarizes the portions of MIPS assembly language described in this section, and Figure 2.13 summarizes the corresponding MIPS machine language. This step along the evolution of the MIPS language has added branches and jumps to our symbolic representation, and fixes the useful value 0 permanently in a register.

Elaboration: If you have heard about *delayed branches*, covered in Chapter 6, don't worry: The MIPS assembler makes them invisible to the assembly language programmer.

MIPS operands

Name	Example	Comments
32 registers	`$s0, $s1, . . . , $s7` `$t0, $t1, . . . , $t7,` `$zero`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers `$s0`–`$s7` map to 16–23 and `$t0`–`$t7` map to 8–15. MIPS register $zero always equals 0.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. 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 operands; data in registers
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three operands; data in registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory
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 $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	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,L`	`if ($s1 == $s2) go to L`	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	`if ($s1 != $s2) go to L`	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	`if ($s2 < $s3) $s1 = 1;` `else $s1 = 0`	Compare less than; used with `beq`, `bne`
	set on less than immediate	`slt $s1,$s2,100`	`if ($s2 < 100) $s1 = 1;` `else $s1 = 0`	Compare less than immediate; used with `beq`, `bne`
Unconditional jump	jump	`j L`	go to L	Jump to target address

FIGURE 2.12 MIPS architecture revealed through Section 2.6. Highlighted portions show MIPS structures introduced in Section 2.6.

Check Yourself

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.

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
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 2.13 MIPS machine language revealed through Section 2.6. Highlighted portions show MIPS structures introduced in Section 2.6. The J-format, used for jump instructions, is explained in Section 2.9. Section 2.9 also explains the proper values in address fields of branch instructions.

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.

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.

2.7 Supporting Procedures in Computer Hardware

A **procedure** or function is one tool C or Java 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, with parameters acting as a barrier between the procedure and the rest of the program and data, allowing it to be passed values and return results. We describe the equivalent in Java at the end of this section, but Java needs everything from a computer that C needs.

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his 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. Place 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. Place 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 in allocating its 32 registers for procedure calling:

- `$a0–$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

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).

```
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`, is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

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`.

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 saves PC + 4 in register `$ra` to link to the following instruction to set up the procedure return.

program counter (PC) The register containing the address of the instruction in the program being executed

To support such situations, computers like MIPS use a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register:

```
jr  $ra
```

The 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`–`$v1`, and returns control to the caller using `jr $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.

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 on page 58.

stack A data structure for spilling registers organized as a last-in-first-out queue.

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. 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*.

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.

MIPS software allocates another register just for the stack: the stack pointer (`$sp`), used to save the registers needed by the callee. 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 51 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 51, 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 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 2.14 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 51:

```
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 example above 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`: 10 temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0`–`$s7`: 8 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 (procedure doing the calling) 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.

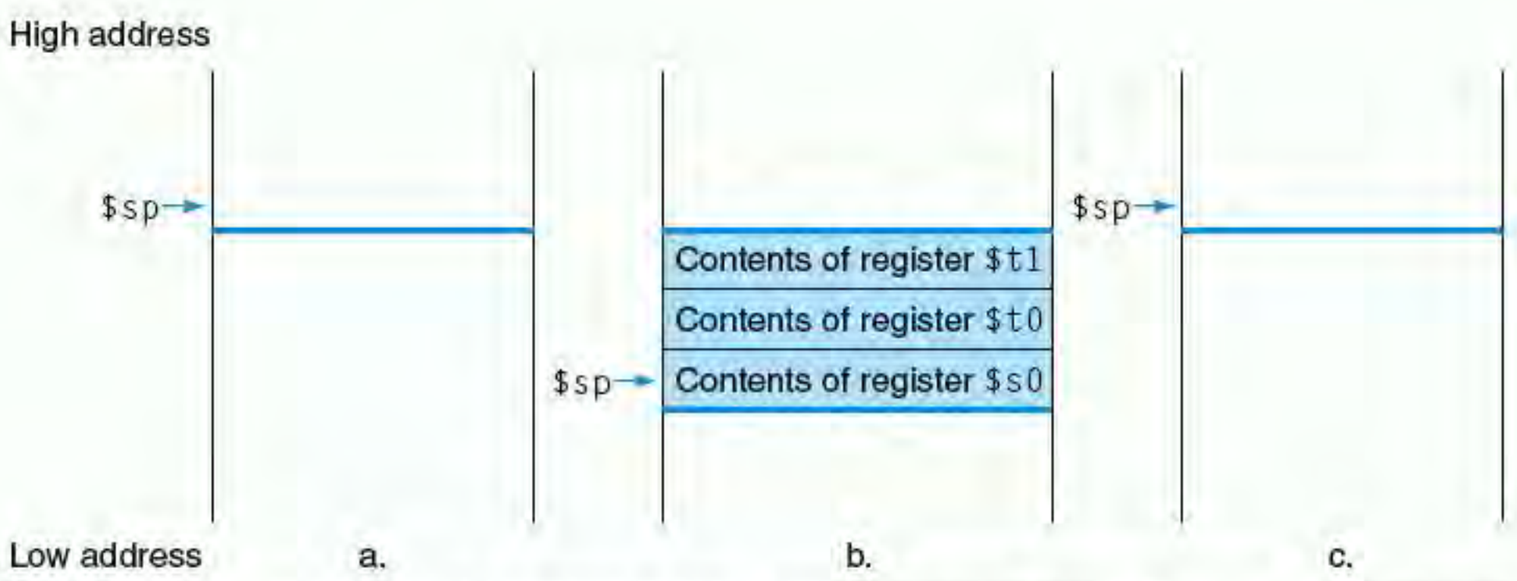


FIGURE 2.14 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

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 if `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 after jal
```

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)
    jalfact        # 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 a location in storage, and its interpretation depends both on its *type* and *storage class*. Types are discussed in detail in Chapter 3, but examples include integers and characters. 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 static data.

Figure 2.15 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack. The stack above `$sp` is preserved simply by making sure the callee does not write above `$sp`; `$sp` is itself preserved by the callee adding exactly the same amount that was subtracted from it, and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there. These actions also guarantee that the caller will get the same data back on a load from the stack as it put into the stack on a store because the callee promises to preserve `$sp` and because the callee also promises not to modify the caller's portion of the stack, that is, the area above the `$sp` at the time of the call.

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 2.15 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 sections, 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 that 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 2.16 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.

frame pointer A value denoting the location of the saved registers and local variables for a given procedure.

Some MIPS software uses a **frame pointer** (`$fp`) to point to the first word of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We've been avoiding `$fp` by avoiding changes to `$sp` within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.

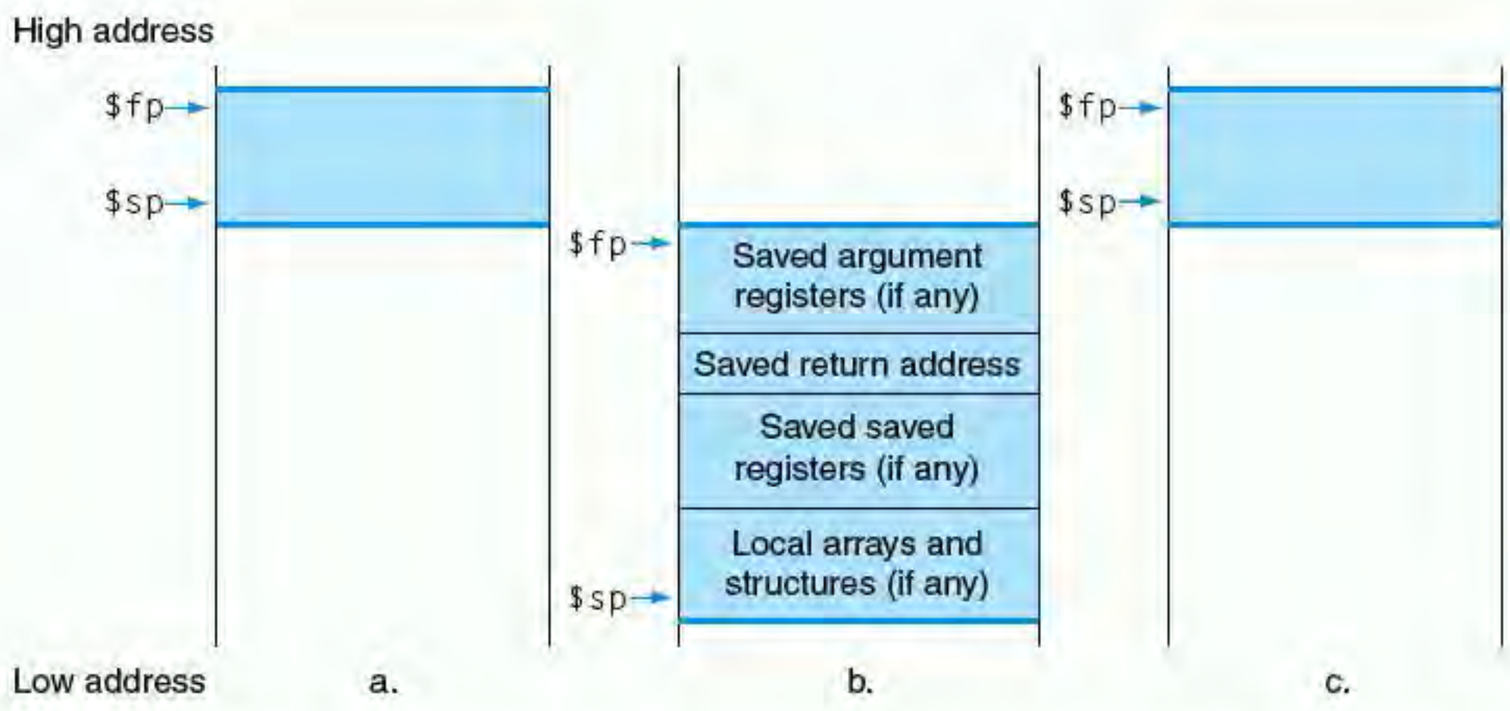


FIGURE 2.16 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack. The stack is adjusted to make room 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`.

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 2.17 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 to 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.

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 stack 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

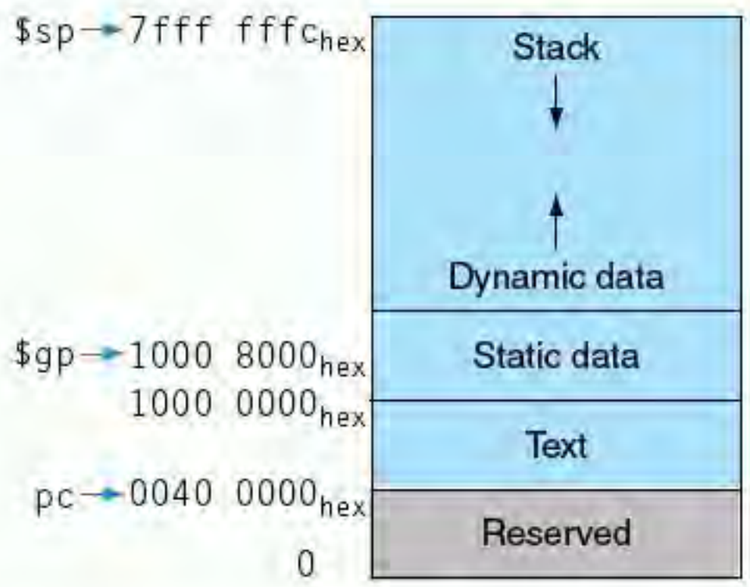


FIGURE 2.17 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. Starting top down, the stack pointer is initialized to $7fff\ fffc_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at $0040\ 0000_{hex}$. The static data starts at $1000\ 0000_{hex}$. Dynamic data, allocated by `malloc` in C and via `new` in Java, is next and 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` (see two's complement addressing in Chapter 3).

leads to a "memory leak" which eventually uses up so much memory that the operating system may crash. Freeing space too early leads to a "dangling pointers," which can cause pointers to point to things that the program never intended.

Figure 2.18 summarizes the register conventions for the MIPS assembly language. Figures 2.19 and 2.20 summarize the parts of the MIPS assembly instructions described so far and the corresponding MIPS machine instructions.

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 2.16, 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/Silicon Graphics does not; it uses register 30 as another save register (`$s8`).

`jal` actually saves the address of the instruction that *follows* `jal` into register `$ra`, thereby allowing a procedure return to be simply `jr $ra`.

Check Yourself Which of the following statements about C and Java are generally true?

1. Procedure calls in C are faster than method invocation in Java.
2. C programmers manage data explicitly while it's automatic in Java.
3. C leads to more pointer bugs and memory leak bugs than does Java.
4. C passes parameters in registers while Java passes them on the stack.

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 2.18 MIPS register conventions. Register 1, called `$at`, is reserved for the assembler (see Section 2.10), and registers 26–27, called `$k0–$k1`, are reserved for the operating system.

MIPS operands

Name	Example	Comments
32 registers	`$s0`–`$s7`, `$t0`–`$t9`, `$zero`, `$a0`–`$a3`, `$v0`–`$v1`, `$gp`, `$fp`, `$sp`, `$ra`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. `$gp` (28) is the global pointer, `$sp` (29) is the stack pointer, `$fp` (30) is the frame pointer, and `$ra` (31) is the return address.
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, such as those saved on procedure calls.

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
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 +100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 +100] = $s1`	Data from register to memory
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 $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	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,L`	if (`$s1 == $s2`) go to L	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	if (`$s1 != $s2`) go to L	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; used with `beq`, `bne`
	set on less than immediate	`slt $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare less than immediate; used with `beq`, `bne`
Unconditional jump	jump	`j L`	go to L	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For procedure return
	jump and link	`jal L`	`$ra` = PC + 4; go to L	For procedure call

FIGURE 2.19 MIPS architecture revealed through Section 2.7. Highlighted portions show MIPS assembly language structures introduced in Section 2.7. The J-format, used for jump and jump-and-link instructions, is explained in Section 2.9.

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
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

FIGURE 2.20 MIPS machine language revealed through Section 2.7. Highlighted portions show MIPS assembly language structures introduced in Section 2.7. The J-format, used for jump and jump-and-link instructions, is explained in Section 2.9. This section also explains why putting 25 in the address field of `beq` and `bne` machine language instructions is equivalent to 100 in assembly language.

!(@ | = >
(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)

2.8 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 use 8-bit bytes to represent characters, with the American Standard Code for Information Interchange (ASCII) being the representation that nearly everyone follows. Figure 2.21 summarizes ASCII.

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

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
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 2.21 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 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.

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
```

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.

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, which puts the character into `$t2`:

```
    lb     $t2, 0($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, 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;
                        # restore 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 2.15 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 2.22 is a list of Unicode alphabets; there are about 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.

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 2.22 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 8 bits and uses 16–32 bits for the other characters. UTF-32 uses 32 bits per character. To learn more, see *www.unicode.org*.

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. 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

```
lh $t0,0($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.

Check Yourself

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.

2.9 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 2.23 shows the operation of `lui`.

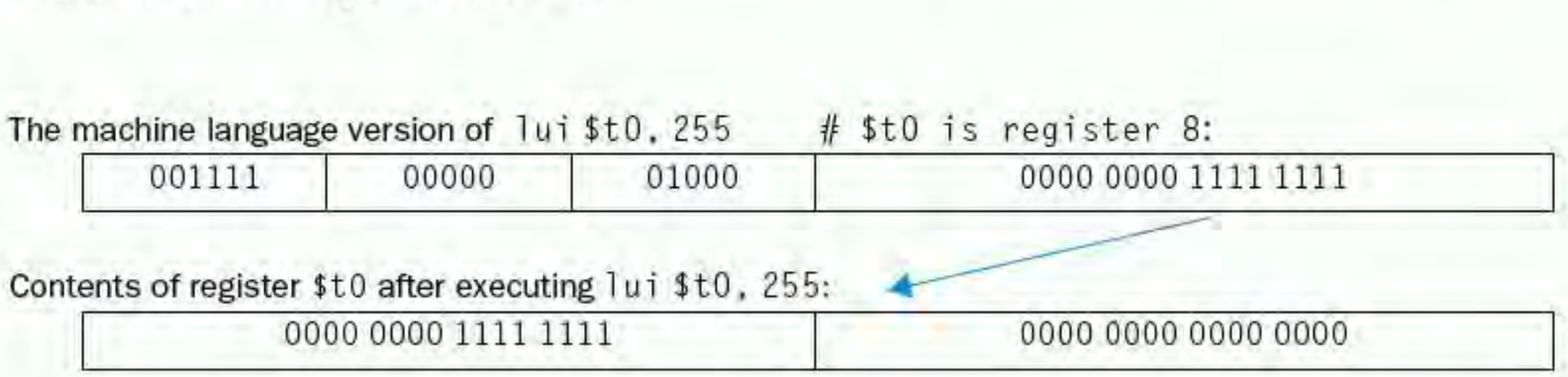


FIGURE 2.23 The effect of the `lui` instruction. The instruction `lui` transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

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 to whatever the creator of an assembler chooses to include (see Section 2.10). 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.

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 add 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
```

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 2.5 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 on the next page):



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:



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. The question is then, 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 SPEC2000 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 5, 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, and 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. 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 2.9) 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.

EXAMPLE

Showing Branch Offset in Machine Language

The *while* loop on page 74 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 would look like this:

80000	0	0	19	9	4	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	. . .					

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word. The `bne` instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction (8 + 80016) instead of relative to the branch instruction (12 + 80012) or using the full destination address (80024). The jump instruction on the last line does use the full address ($20000 \times 4 = 80000$), corresponding to the label `Loop`.

Hardware Software Interface

Nearly every conditional branch is to a nearby location, but occasionally it branches far away, farther than can be represented in the 16 bits of the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump.

Branching Far Away

EXAMPLE

Given a branch on register `$s0` being equal to register `$s1`,

```
beq    $s0,$s1, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

ANSWER

These instructions replace the short-address conditional branch:

```
      bne    $s0,$s1, L2
      j      L1
L2:
```

MIPS Addressing Mode Summary

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

Multiple forms of addressing are generically called **addressing modes**. The MIPS addressing modes are the following:

1. **Register addressing**, where the operand is a register
2. **Base** or **displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
3. **Immediate addressing**, where the operand is a constant within the instruction itself
4. **PC-relative addressing**, where the address is the sum of the PC and a constant in the instruction
5. **Pseudodirect addressing**, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Hardware Software Interface

Although we show the MIPS architecture as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions (see Appendix D). These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in a way that lets software move compatibly upward to the next generation of architecture.

Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing. Figure 2.24 shows how operands are identified for each addressing mode. In More Depth shows other addressing modes found in the IBM PowerPC.

Decoding Machine Language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at a core dump. Figure 2.25 shows the MIPS encoding of the fields for the MIPS machine language. This figure helps when translating by hand between assembly language and machine language.

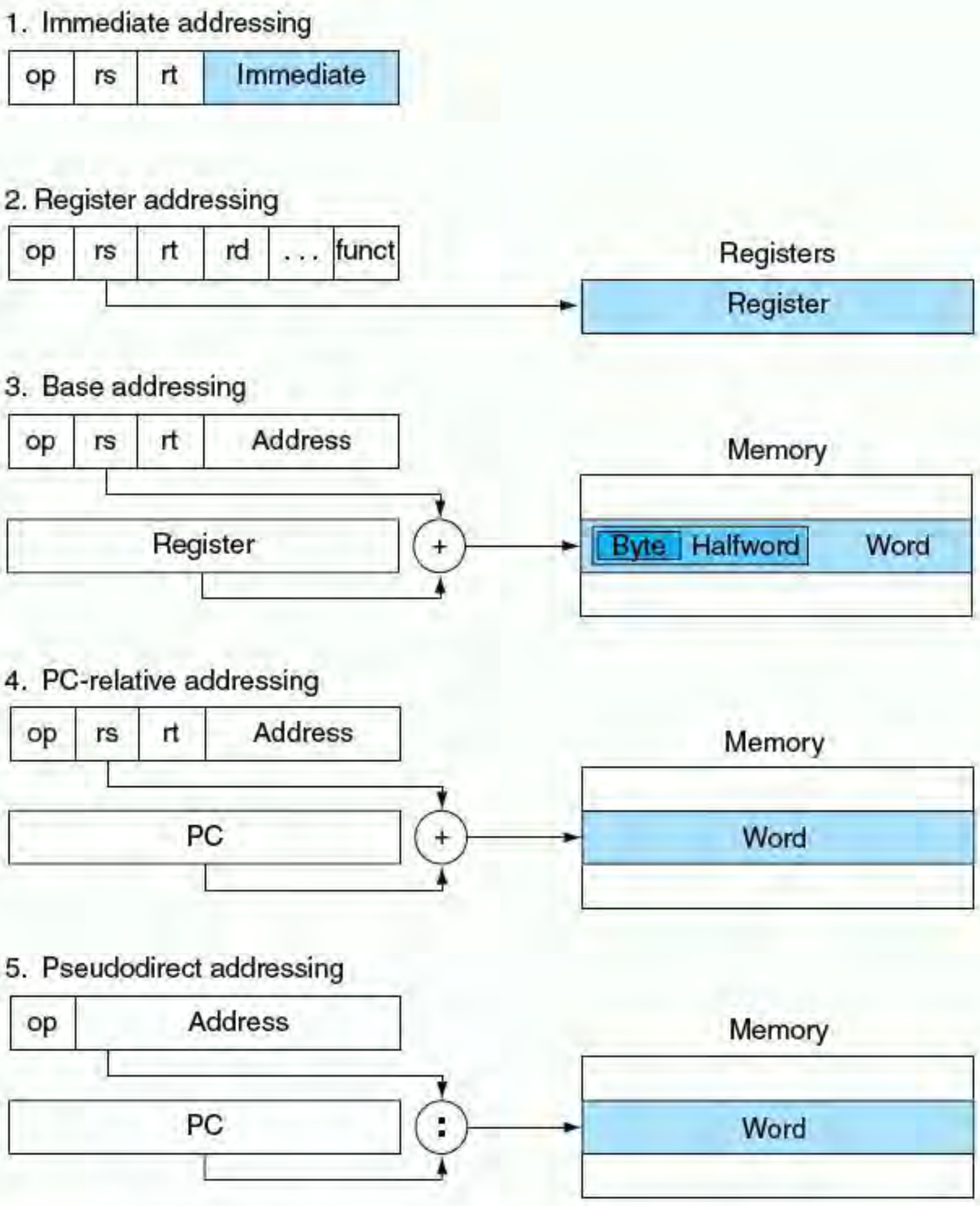


FIGURE 2.24 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

EXAMPLE

Decoding Machine Code

What is the assembly language statement corresponding to this machine instruction?

```
00af8020hex
```

ANSWER

The first step in converting hexadecimal to binary is to find the op fields:

```
(Bits:   31  28 26                                  5   2   0)
         0000 0000 1010 1111 1000 0000 0010 0000
```

We look at the op field to determine the operation. Referring to Figure 2.25, when bits 31–29 are 000 and bits 28–26 are 000, it is an R-format instruction. Let's reformat the binary instruction into R-format fields, listed in Figure 2.26:

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

The bottom portion of Figure 2.25 determines the operation of an R-format instruction. In this case, bits 5–3 are 100 and bits 2–0 are 000, which means this binary pattern represents an `add` instruction.

We decode the rest of the instruction by looking at the field values. The decimal values are 5 for the rs field, 15 for rt, 16 for rd (shamt is unused). Figure 2.18 says these numbers represent registers `$a1`, `$t7`, and `$s0`. Now we can show the assembly instruction:

```
add $s0,$a1,$t7
```

Figure 2.26 shows all the MIPS instruction formats. Figure 2.27 shows the MIPS assembly language revealed in Chapter 2; the remaining hidden portion of MIPS instructions deals mainly with arithmetic covered in the next chapter.

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	sltiu	andi	ori	xori	load upper imm
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	lbu	lhu	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26)=010000 (TLB), rs(25:21)								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump reg.	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	sltu				
6(110)								
7(111)								

FIGURE 2.25 MIPS instruction encoding. This notation gives the value of a field by row and by column. For example, the top portion of the figure shows `load word` in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, `R-format` in row 0 and column 0 (op = 000000_{two}) is defined in the bottom part of the figure. Hence, `subtract` in row 4 and column 2 of the bottom section means that the funct field (bits 5–0) of the instruction is 100010_{two} and the op field (bits 31–26) is 000000_{two}. The `FlPt` value in row 2, column 1 is defined in Figure 3.20 in Chapter 3. `Bltz/gez` is the opcode for four instructions found in Appendix A: `bltz`, `bgez`, `bltzal`, and `bgezal`. Chapter 2 describes instructions given in full name using color, while Chapter 3 describes instructions given in mnemonics using color. Appendix A covers all instructions.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

FIGURE 2.26 MIPS instruction formats in Chapter 2. Highlighted portions show instruction formats introduced in this section.

Check Yourself

What is the range of addresses for conditional branches in MIPS (K = 1024)?

1. Addresses between 0 and 64K – 1
2. Addresses between 0 and 256K – 1
3. Addresses up to about 32K before the branch to about 32K after
4. Addresses up to about 128K before the branch to about 128K after

What is the range of addresses for jump and jump and link in MIPS (M = 1024K)?

1. Addresses between 0 and 64M – 1
2. Addresses between 0 and 256M – 1
3. Addresses up to about 32M before the branch to about 32M after
4. Addresses up to about 128M before the branch to about 128M after
5. Anywhere within a block of 64M addresses where the PC supplies the upper 6 bits
6. Anywhere within a block of 256M addresses where the PC supplies the upper 4 bits

What is the MIPS assembly language instruction corresponding to the machine instruction with the value $0000\ 0000_{hex}$?

1. `j`
2. `R-format`
3. `addi`
4. `sll`
5. `mfc0`
6. Undefined opcode: there is no legal instruction that corresponds to 0.

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7, $t0–$t9, $zero, $a0–$a3, $v0–$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for 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, such as those saved on procedure calls.

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 $s1,$s2,100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load half	`lh $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte	`lb $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immed.	`lui $s1,100`	$s1 = 100 * 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 $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	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 less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional 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 2.27 MIPS assembly language revealed in Chapter 2. Highlighted portions show portions from Sections 2.8 and 2.9.

2.10 Translating and Starting a Program

This section describes the four steps in transforming a C program in a file on disk into a program running on a computer. Figure 2.28 shows the translation hierarchy. Some systems combine these steps to reduce translation time, but these are the logical four phases that programs go through. This section follows this translation hierarchy.

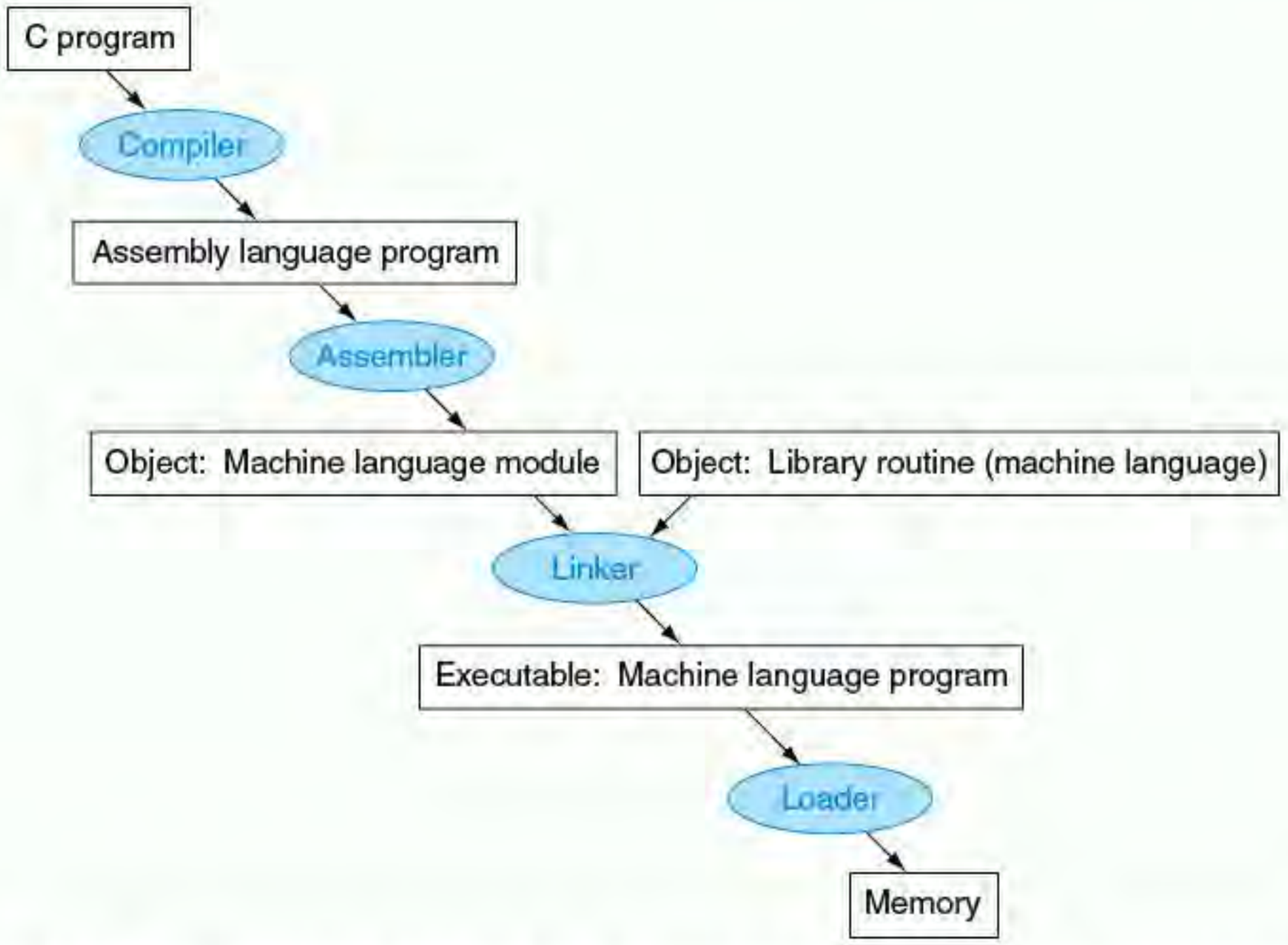


FIGURE 2.28 A translation hierarchy for C. A high-level-language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined together. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routes are `x.so`, and executable files by default are called `a.out`. MS-DOS uses the suffixes `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL`, and `.EXE` to the same effect.

Compiler

The compiler transforms the C program into an *assembly language program*, a symbolic form of what the machine understands. High-level-language programs take many fewer lines of code than assembly language, so programmer productivity is much higher.

In 1975, many operating systems and assemblers were written in **assembly language** because memories were small and compilers were inefficient. The 128,000-fold increase in memory capacity per single DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as good as an assembly language expert, and sometimes even better for large programs.

assembly language A symbolic language that can be translated into binary.

Assembler

As mentioned on page 96, since assembly language is the interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right. The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called **pseudoinstructions**.

pseudoinstruction A common variation of assembly language instructions often treated as if it were an instruction in its own right.

As mentioned above, the MIPS hardware makes sure that register `$zero` always has the value 0. That is, whenever register `$zero` is used, it supplies a 0, and the programmer cannot change the value of register `$zero`. Register `$zero` is used to create the assembly language instruction `move` that copies the contents of one register to another. Thus the MIPS assembler accepts this instruction even though it is not found in the MIPS architecture:

```
move $t0,$t1        # register $t0 gets register $t1
```

The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:

```
add  $t0,$zero,$t1 # register $t0 gets 0 + register $t1
```

The MIPS assembler also converts `blt` (branch on less than) into the two instructions `slt` and `bne` mentioned in the example on page 96. Other examples include `bgt`, `bge`, and `ble`. It also converts branches to faraway locations into a branch and jump. As mentioned above, the MIPS assembler allows 32-bit constants to be loaded into a register despite the 16-bit limit of the immediate instructions.

In summary, pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware. The only cost is reserving one register, `$at`, for use by the assembler. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the MIPS

architecture and to be sure to get best performance, however, study the real MIPS instructions found in Figures 2.25 and 2.27.

Assemblers will also accept numbers in a variety of bases. In addition to binary and decimal, they usually accept a base that is more succinct than binary yet converts easily to a bit pattern. MIPS assemblers use hexadecimal,

Such features are convenient, but the primary task of an assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

machine language Binary representation used for communication within a computer system.

To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instructions in a **symbol table**. As you might expect, the table contains pairs of symbol and address.

symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy.

The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use either *static data,* which is allocated throughout the program, or *dynamic data*, which can grow or shrink as needed by the program.)
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

The next subsection shows how to attach such routines that have already been assembled, such as library routines.

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to

compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a **link editor** or **linker**, which takes all the independently assembled machine language programs and "stitches" them together.

linker Also called **link editor**. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions, jump instructions, and data addresses, so the job of this program is much like that of an editor: It finds the old addresses and replaces them with the new addresses. Editing is the origin of the name "link editor," or linker for short. The reason a linker makes sense is that it is much faster to patch code than it is to recompile and reassemble.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that Figure 2.17 on page 87 shows the MIPS convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module's instructions and data will be placed relative to other modules. When the linker places a module in memory, all *absolute* references, that is, memory addresses that are not relative to a register, must be *relocated* to reflect its true location.

The linker produces an **executable file** that can be run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references. It is possible to have partially linked files, such as library routines, which still have unresolved addresses and hence result in object files.

executable file A functional program in the format of an object file that contains no unresolved references, relocation information, symbol table, or debugging information.

EXAMPLE

Linking Object Files

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures `A` and `B` and the instructions that refer to the addresses of data words `X` and `Y`.

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	—	
	B	—	

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	—	
	A	—	

ANSWER

Procedure A needs to find the address for the variable labeled X to put in the load instruction and to find the address of procedure B to place in the `jal` instruction. Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its `jal` instruction.

From Figure 2.17 on page 87, we know that the text segment starts at address $40\,0000_{hex}$ and the data segment at $1000\,0000_{hex}$. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\,0100_{hex}$, and its data starts at $1000\,0020_{hex}$.

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

From Figure 2.17 on page 87, we know that the text segment starts at address $40\ 0000_{hex}$ and the data segment at $1000\ 0000_{hex}$. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$.

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have two types here:

1. The jals are easy because they use pseudodirect addressing. The jal at address $40\ 0004_{hex}$ gets $40\ 0100_{hex}$ (the address of procedure B) in its address field, and the jal at $40\ 0104_{hex}$ gets $40\ 0000_{hex}$ (the address of procedure A) in its address field.

2. The load and store addresses are harder because they are relative to a base register. This example uses the global pointer as the base register. Figure 2.17 shows that $gp is initialized to $1000\ 8000_{hex}$. To get the address $1000\ 0000_{hex}$ (the address of word X), we place 8000_{hex} in the address field of lw at address $40\ 0000_{hex}$. Chapter 3 explains 16-bit two's complement computer arithmetic, which is why 8000_{hex} in the address field yields $1000\ 0000_{hex}$ as the address. Similarly, we place 8020_{hex} in the address field of sw at address $40\ 0100_{hex}$ to get the address $1000\ 0020_{hex}$ (the address of word Y).

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. It follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the machine registers and sets the stack pointer to the first free location.
6. Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

loader A systems program that places an object program in main memory so that it is ready to execute.

Sections A.3 and A.4 in Appendix A describe linkers and **loaders** in more detail.

Dynamically Linked Libraries

The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

- The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
- It loads the whole library even if all of the library is not used when the program is run. The library can be large relative to the program; for example, the standard C library is 2.5 MB.

These disadvantages lead to dynamically linked libraries (DLLs), where the library routines are not linked and loaded until the program is run. Both the program and library routines keep extra information on the location of nonlocal procedures and their names. In the initial version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called versus those that are called during the running of the program. This observation led to the lazy procedure linkage version of DLLs, where each routine is linked only *after* it is called.

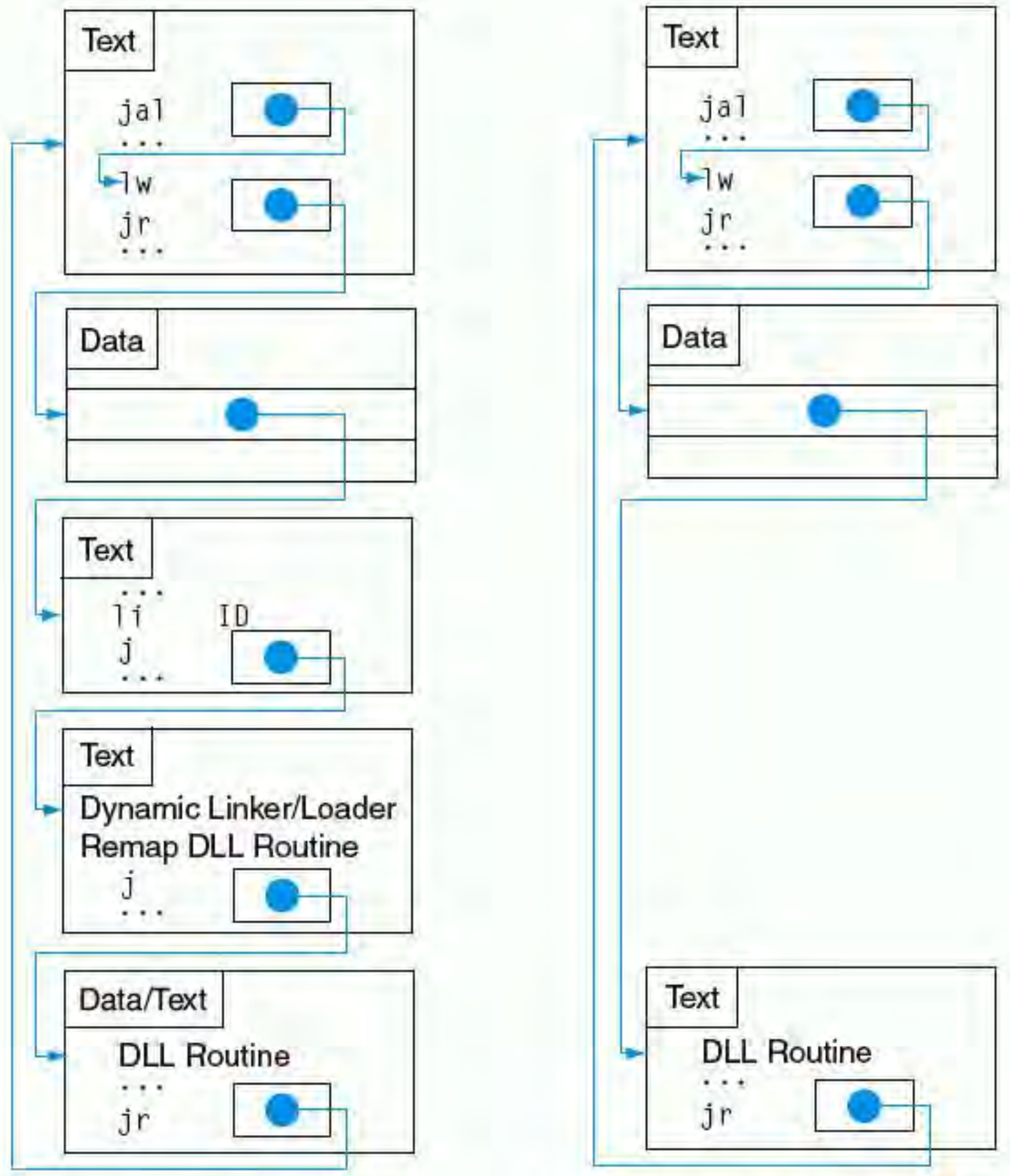


FIGURE 2.29 Dynamically linked library via lazy procedure linkage. (a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 7, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

Like many instances in our field, this trick relies on a level of indirection. Figure 2.29 shows the technique. It starts with the nonlocal routines calling a set of dummy routines at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect jump.

The first time the library routine is called, the program calls the dummy entry and follows the indirect jump. It points to code that puts a number in a register to identify the desired library routine and then jumps to the dynamic linker-loader.

The linker-loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine. It then jumps to it. When the routine completes, it returns to the original calling site. Thereafter, it jumps indirectly to the routine without the extra hops.

In summary, DLLs require extra space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect jump thereafter. Note that the return from the library pays no extra overhead. Microsoft's Windows relies extensively on lazy dynamically linked libraries, and it is also the normal way of executing programs on UNIX systems today.

Starting a Java Program

The discussion above captures the traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a specific implementation of that architecture. Indeed, it is possible to execute Java programs just like C. Java was invented with a different set of goals, however. One was to quickly run safely on any computer, even if it might slow execution time.

Figure 2.30 shows the typical translation and execution steps for Java. Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the Java bytecode instruction set. This instruction set is designed to be close to the Java language so that this com-

Java bytecode Instruction from an instruction set designed to interpret Java programs.

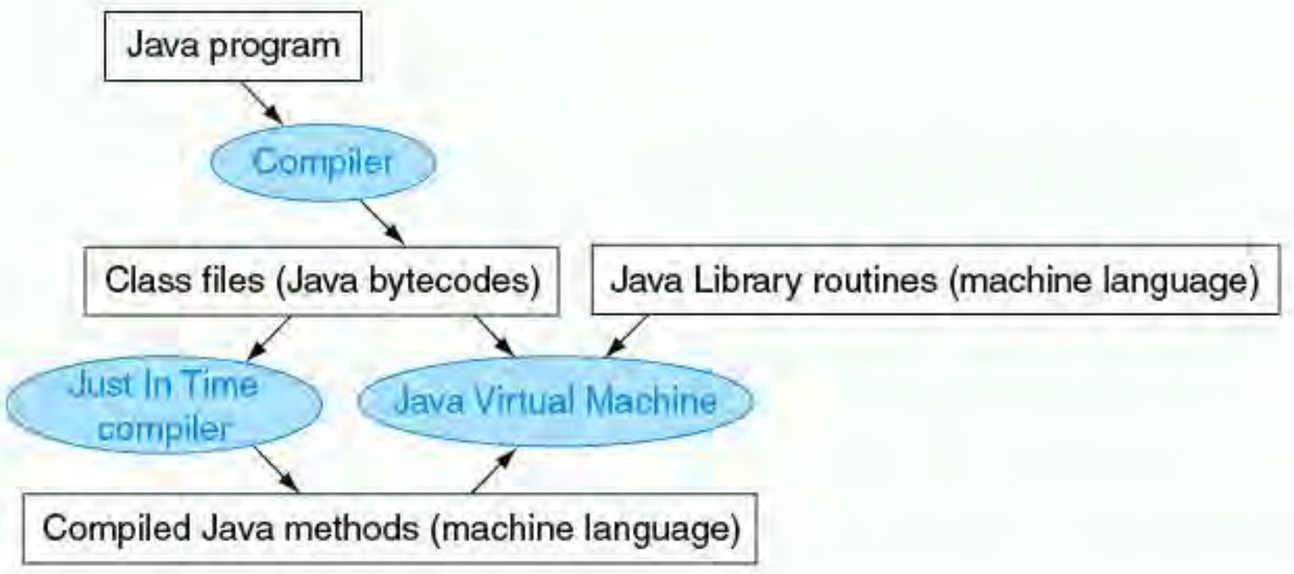


FIGURE 2.30 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the Java Virtual Machine (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the Just In Time (JIT) compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

pilation step is trivial. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

A software interpreter, called a **Java Virtual Machine** (JVM), can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the MIPS simulator used with this book is an interpreter. There is no need for a separate assembly step since either the translation is so simple that the compiler fills in the addresses or JVM finds them at runtime.

Java Virtual Machine (JVM) The program that interprets Java bytecodes.

The upside of interpretation is portability. The availability of software Java virtual machines meant that most could write and run Java programs shortly after Java was announced. Today Java virtual machines are found in millions of devices, in everything from cell phones to Internet browsers.

The downside of interpretation is low performance. The incredible advances in performance of the 1980s and 1990s made interpretation viable for many important applications, but the factor of 10 slowdown when compared to traditionally compiled C programs made Java unattractive for some applications.

To preserve portability and improve execution speed, the next phase of Java development was compilers that translated *while* the program was running. Such **Just In Time compilers (JIT)** typically profile the running program to find where the "hot" methods are, and then compile them into the native instruction set on which the virtual machine is running. The compiled portion is saved for the next time the program is run, so that it can run faster each time it is run. This balance of interpretation and compilation evolves over time, so that frequently run Java programs suffer little of the overhead of interpretation.

Just In Time Compiler (JIT) The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

As computers get faster so that compilers can do more, and as researchers invent betters ways to compile Java on the fly, the performance gap between Java and C or C++ is closing. Section 2.14 goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

Check Yourself

Which of the advantages of an interpreter over a translator do you think was most important for the designers of Java?

1. Ease of writing an interpreter
2. Better error messages
3. Smaller object code
4. Machine independence

2.11 How Compilers Optimize

Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. The purpose of this section is to give a brief overview of optimizations a compiler uses to achieve performance. The following section introduces the internal anatomy of a compiler. To start, Figure 2.31 shows the structure of recent compilers, and we describe the optimizations in the order of the passes of that structure.

High-Level Optimizations

High-level optimizations are transformations that are done at something close to the source level.

The most common high-level transformation is probably *procedure inlining*, which replaces a call to a function by the body of the function, substituting the caller's arguments for the procedure's parameters. Other high-level optimizations

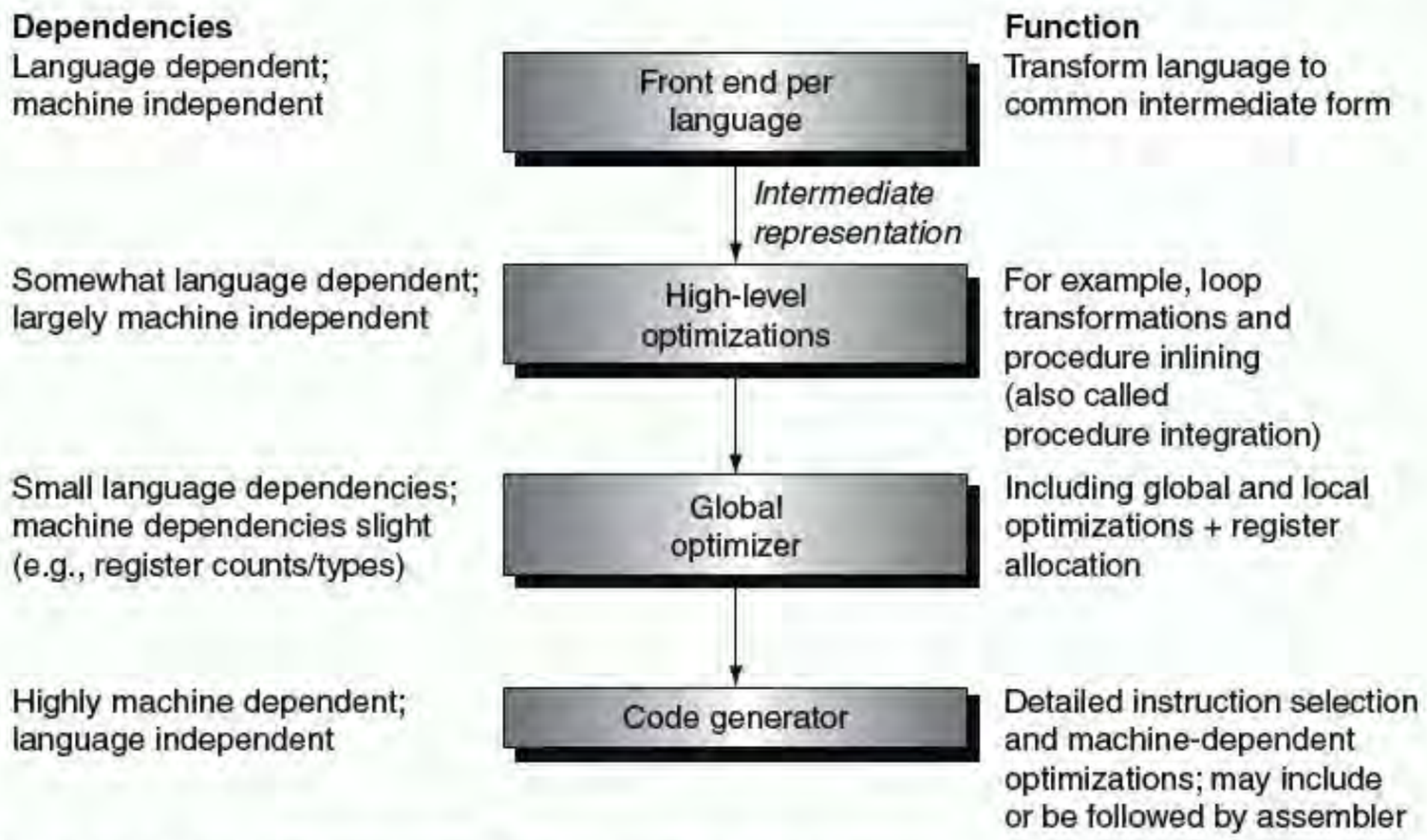


FIGURE 2.31 The structure of a modern optimizing compiler consists of a numbers of passes or phases. Logically each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle a procedure at a time, essentially interleaving with another pass.

involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively. For example, in loops that execute many iterations, such as those traditionally controlled by a *for* statement, the optimization of **loop unrolling** is often useful. Loop unrolling involves taking a loop and replicating the body multiple times and executing the transformed loop fewer times. Loop unrolling reduces the loop overhead and provides opportunities for many other optimizations. Other types of high-level transformations include sophisticated loop transformations such as interchanging nested loops and blocking loops to obtain better memory behavior; see Chapter 7 for examples.

loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

Local and Global Optimizations

Within the pass dedicated to local and global optimization, three classes of optimizations are performed:

1. *Local optimization* works within a single basic block. A local optimization pass is often run as a precursor and successor to global optimization to "clean up" the code before and after global optimization.
2. *Global optimization* works across multiple basic blocks; we will see an example of this shortly.
3. Global *register allocation* allocates variables to registers for regions of the code. Register allocation is crucial to getting good performance in modern processors.

Several optimizations are performed both locally as well as globally, including common subexpression elimination, constant propagation, copy propagation, dead store elimination, and strength reduction. Let's look at some simple examples of these optimizations.

Common subexpression elimination finds multiple instances of the same expression and replaces the second one by a reference to the first. Consider, for example, a code segment to add 4 to an array element:

```
x[i] = x[i] + 4
```

The address calculation for `x[i]` occurs twice and is identical since neither the starting address of x nor the value of i changes. Thus, the calculation can be reused. Let's look at the intermediate code for this fragment, since it allows several other optimizations to be performed. Here is the unoptimized intermediate code on the left, and on the right is the code with common subexpression elimination replacing the second address calculation with the first. Note that the register allocation has not yet occurred, so the compiler is using virtual register numbers like `R100` here.

```
# x[i] + 4
li R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
# value of x[i] is in R104
add R105,R104,4
# x[i] =
li R106,x
lw R107,i
mult R108,R107,4
add R109,R106,R107
sw R105,0(R109)
```

```
# x[i] + 4
li R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
# value of x[i] is in R104
add R105,R104,4
# x[i] =
sw R105,0(R103)
```

If the same optimization was possible across two basic blocks, it would then be an instance of *global common subexpression elimination.*

Let's consider some of the other optimizations:

- *Strength reduction* replaces complex operations by simpler ones and can be applied to this code segment, replacing the `mult` by a shift left.
- *Constant propagation* and its sibling *constant folding* find constants in code and propagates them, collapsing constant values whenever possible.
- *Copy propagation* propagates values that are simple copies, eliminating the need to reload values and possibly enabling other optimizations such as common subexpression elimination.
- *Dead store elimination* finds stores to values that are not used again and eliminates the store; its "cousin" is *dead code elimination*, which finds unused code—code that cannot affect the final result of the program—and eliminates it. With the heavy use of macros, templates, and the similar techniques designed to reuse code in high-level languages, dead code occurs surprisingly often.

Understanding Program Performance

Programmers concerned about performance of critical loops, especially in real-time or embedded applications, often find themselves staring at the assembly language produced by a compiler and wondering why the compiler failed to perform some global optimization or to allocate a variable to a register throughout a loop. The answer often lies in the dictate that the compiler be conservative. The opportunity for improving the code may seem obvious to the programmer, but then the programmer often has knowledge that the compiler does not have, such as the absence of aliasing between two pointers or the absence of side effects by a function call. The compiler may indeed be able to perform the transformation with a little help, which could eliminate the worst-case behavior that it must assume. This insight also illustrates an important observation: programmers who use pointers to try to improve performance in accessing variables, especially pointers to values on the stack that also have names as variables or as elements of arrays, are likely to disable many compiler optimizations. The end result is that the lower-level pointer code may run no better, or perhaps even worse, than the higher-level code optimized by the compiler.

Compilers must be *conservative*. The first task of a compiler is to produce correct code; its second task is usually to produce fast code although other factors such as code size may sometimes be important as well. Code that is fast but incorrect—for any possible combination of inputs—is simply wrong. Thus, when we say a compiler is "conservative," we mean that it performs an optimization only if it knows with 100% certainty that, no matter what the inputs, the code will perform as the user wrote it. Since most compilers translate and optimize one function or procedure at a time, most compilers, especially at lower optimization levels, assume the worst about function calls and about their own parameters.

Global Code Optimizations

Many global code optimizations have the same aims as those used in the local case, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead code elimination.

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops. *Code motion* finds code that is loop invariant: a particular piece of code computes the same value on every loop iteration and, hence, may be computed once outside the loop. *Induction variable elimination* is a combination of

transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses. Rather than examine induction variable elimination in depth, we point the reader to Section 2.15, which compares the use of array indexing and pointers; for most loops, the transformation from the more obvious array code to the pointer code can be performed by a modern optimizing compiler.

Optimization Summary

Figure 2.32 gives examples of typical optimizations, and the last column indicates where the optimization is performed in the gcc compiler. It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator, and some optimizations are done multiple times, especially local optimizations, which may be performed before and after global optimization as well as during code generation.

Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	03
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	01
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	01
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	01
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	02
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	02
Code motion	Remove code from a loop that computes same value each iteration of the loop	02
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	02
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	01
Pipeline scheduling	Reorder instructions to improve pipeline performance	01
Branch offset optimization	Choose the shortest branch displacement that reaches target	01

FIGURE 2.32 Major types of optimizations and examples in each class. The third column shows when these occur at different levels of optimization in gcc. The Gnu organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

Hardware
Software
Interface

Today essentially all programming for desktop and server applications is done in high-level languages, as is most programming for embedded applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. With Moore's law comes the temptation of adding sophisticated operations in an instruction set. The challenge is that they may not exactly match what the compiler needs to produce or be so general that they aren't fast. For example, consider special loop instructions found in some computers. Suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. The loop instruction may be a mismatch. When faced with such objections, the instruction set designer might then generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be slower than a sequence of simple operations.

2.12 How Compilers Work: An Introduction

The purpose of this section is to give a brief overview of the compiler function, which will help the reader understand both how the compiler translates a high-level language program into machine instructions. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course; our introduction will necessarily only touch on the basics. The rest of this section is on the CD.

2.13 A C Sort Example to Put It All Together

One danger of showing assembly language code in snippets is that you will have no idea what a full assembly language program looks like. In this section, we derive the MIPS code from two procedures written in C: one to swap array elements and one to sort them.

The Procedure `swap`

Let's start with the code for the procedure `swap` in Figure 2.33. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

This section describes the `swap` procedure in these three pieces, concluding by putting all the pieces together.

Register Allocation for `swap`

As mentioned on page 79, the MIPS convention on parameter passing is to use registers `$a0`, `$a1`, `$a2`, and `$a3`. Since swap has just two parameters, `v` and `k`, they will be found in registers `$a0` and `$a1`. The only other variable is `temp`, which we associate with register `$t0` since `swap` is a leaf procedure (see page 83). This register allocation corresponds to the variable declarations in the first part of the `swap` procedure in Figure 2.33.

Code for the Body of the Procedure `swap`

The remaining lines of C code in `swap` are

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Recall that the memory address for MIPS refers to the *byte* address, and so words are really 4 bytes apart. Hence we need to multiply the index `k` by 4 before

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

FIGURE 2.33 A C procedure that swaps two locations in memory. The next subsection uses this procedure in a sorting example.

adding it to the address. *Forgetting that sequential word addresses differ by 4 instead of by 1 is a common mistake in assembly language programming*. Hence the first step is to get the address of `v[k]` by multiplying `k` by 4:

```
sll    $t1, $a1,2     # reg $t1 = k * 4
add    $t1, $a0,$t1   # reg $t1 = v + (k * 4)
                      # reg $t1 has the address of v[k]
```

Now we load `v[k]` using `$t1`, and then `v[k+1]` by adding 4 to `$t1`:

```
lw     $t0, 0($t1)    # reg $t0 (temp) = v[k]
lw     $t2, 4($t1)    # reg $t2 = v[k + 1]
                      # refers to next element of v
```

Next we store `$t0` and `$t2` to the swapped addresses:

```
sw     $t2, 0($t1)    # v[k] = reg $t2
sw     $t0, 4($t1)    # v[k+1] = reg $t0 (temp)
```

Now we have allocated registers and written the code to perform the operations of the procedure. What is missing is the code for preserving the saved registers used within `swap`. Since we are not using saved registers in this leaf procedure, there is nothing to preserve.

The Full `swap` Procedure

We are now ready for the whole routine, which includes the procedure label and the return jump. To make it easier to follow, we identify in Figure 2.34 each block of code with its purpose in the procedure.

The Procedure `sort`

To ensure that you appreciate the rigor of programming in assembly language, we'll try a second, longer example. In this case, we'll build a routine that calls the `swap` procedure. This program sorts an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts. Figure 2.35 shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

Register Allocation for `sort`

The two parameters of the procedure `sort`, `v` and `n`, are in the parameter registers `$a0` and `$a1`, and we assign register `$s0` to `i` and register `$s1` to `j`.

Procedure body

```
swap:  sll   $t1, $a1, 2      # reg $t1 = k * 4
       add   $t1, $a0, $t1    # reg $t1 = v + (k * 4)
                              # reg $t1 has the address of v[k]
       lw    $t0, 0($t1)      # reg $t0 (temp) = v[k]
       lw    $t2, 4($t1)      # reg $t2 = v[k + 1]
                              # refers to next element of v
       sw    $t2, 0($t1)      # v[k] = reg $t2
       sw    $t0, 4($t1)      # v[k+1] = reg $t0 (temp)
```

Procedure return

```
       jr    $ra              # return to calling routine
```

FIGURE 2.34 MIPS assembly code of the procedure `swap` in Figure 2.33.

```
void sort (int v[], int n)
{
   int i, j;
   for (i = 0; i < n; i += 1) {
      for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
          swap(v,j);
      }
   }
}
```

FIGURE 2.35 A C procedure that performs a sort on the array `v`.

Code for the Body of the Procedure `sort`

The procedure body consists of two nested *for* loops and a call to `swap` that includes parameters. Let's unwrap the code from the outside to the middle.

The first translation step is the first *for* loop:

```
for (i = 0; i < n; i += 1) {
```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to 0, the first part of the *for* statement:

```
move   $s0, $zero      # i = 0
```

(Remember that `move` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer; see page 107.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

```
addi   $s0, $s0, 1     # i += 1
```

The loop should be exited if i < n is *not* true or, said another way, should be exited if i ≥ n. The set on less than instruction sets register $t0 to 1 if $s0 < $a1 and 0 otherwise. Since we want to test if $s0 ≥ $a1, we branch if register $t0 is 0. This test takes two instructions:

```
for1tst:slt $t0, $s0, $a1    # reg $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq  $t0, $zero,exit1 # go to exit1 if $s0≥$a1 (i≥n)
```

The bottom of the loop just jumps back to the loop test:

```
        j   for1tst          # jump to test of outer loop
exit1:
```

The skeleton code of the first *for* loop is then

```
        move   $s0, $zero      # i = 0
for1tst:slt $t0, $s0, $a1    # reg $t0 = 0 if $s0 ≥ $a1 (i≥n)
        beq    $t0, $zero,exit1 # go to exit1 if $s0≥$a1 (i≥n)
               ...
               (body of first for loop)
               ...
        addi   $s0, $s0, 1     # i += 1
        j      for1tst         # jump to test of outer loop
exit1:
```

Voila! Exercise 2.14 explores writing faster code for similar loops.

The second *for* loop looks like this in C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

The initialization portion of this loop is again one instruction:

```
addi    $s1, $s0, -1 # j = i - 1
```

The decrement of j at the end of the loop is also one instruction:

```
addi    $s1, $s1, -1 # j -= 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails (j < 0):

```
for2tst:slti$t0, $s1, 0      # reg $t0 = 1 if $s1 < 0 (j < 0)
        bne $t0, $zero, exit2 # go to exit2 if $s1<0 (j < 0)
```

This branch will skip over the second condition test. If it doesn't skip, j ≥ 0.

The second test exits if v[j] > v[j + 1] is *not* true, or exits if v[j] ≤ v[j + 1]. First we create the address by multiplying j by 4 (since we need a byte address) and add it to the base address of v:

```
sll     $t1, $s1,2    # reg $t1 = j * 4
add     $t2, $a0,$t1  # reg $t2 = v + (j * 4)
```

Now we load `v[j]`:

```
lw      $t3, 0($t2)   # reg $t3   = v[j]
```

Since we know that the second element is just the following word, we add 4 to the address in register `$t2` to get `v[j + 1]`:

```
lw      $t4, 4($t2)   # reg $t4   = v[j + 1]
```

The test of `v[j]` ≤ `v[j + 1]` is the same as `v[j + 1]` ≥ `v[j]`, so the two instructions of the exit test are

```
slt     $t0, $t4, $t3      # reg $t0 = 0 if $t4 ≥ $t3
beq     $t0, $zero,exit2   # go to exit2 if $t4 ≥ $t3
```

The bottom of the loop jumps back to the inner loop test:

```
j      for2tst    # jump to test of inner loop
```

Combining the pieces together, the skeleton of the second *for* loop looks like this:

```
        addi $s1, $s0, -1      # j = i - 1
for2tst:slti $t0, $s1, 0       # reg $t0 = 1 if $s1 < 0 (j<0)
        bne  $t0, $zero,exit2  # go to exit2 if $s1<0 (j<0)
        sll  $t1, $s1,2        # reg $t1 = j * 4
        add  $t2, $a0,$t1      # reg $t2 = v + (j * 4)
        lw   $t3, 0($t2)       # reg $t3   = v[j]
        lw   $t4, 4($t2)       # reg $t4   = v[j + 1]
        slt  $t0, $t4, $t3     # reg $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero,exit2  # go to exit2 if $t4 ≥ $t3
             ...
             (body of second for loop)
             ...
        addi $s1, $s1, -1      # j -= 1
        j    for2tst           # jump to test of inner loop
exit2:
```

The Procedure Call in `sort`

The next step is the body of the second *for* loop:

```
swap(v,j);
```

Calling `swap` is easy enough:

```
jal     swap
```

Passing Parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `$a0` and `$a1`, yet the `swap` procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for `sort` into other registers earlier in the procedure, making registers `$a0` and `$a1` available for the call of `swap`. (This copy is faster than saving and restoring on the stack.) We first copy `$a0` and `$a1` into `$s2` and `$s3` during the procedure:

```
move    $s2, $a0        # copy parameter $a0 into $s2
move    $s3, $a1        # copy parameter $a1 into $s3
```

Then we pass the parameters to `swap` with these two instructions:

```
move    $a0, $s2        # first swap parameter is v
move    $a1, $s1        # second swap parameter is j
```

Preserving Registers in `sort`

The only remaining code is the saving and restoring of registers. Clearly we must save the return address in register `$ra`, since `sort` is a procedure and is called itself. The `sort` procedure also uses the saved registers `$s0`, `$s1`, `$s2`, and `$s3`, so they must be saved. The prologue of the `sort` procedure is then

```
addi    $sp,$sp,-20     # make room on stack for 5 regs
sw      $ra,16($sp)     # save $ra on stack
sw      $s3,12($sp)     # save $s3 on stack
sw      $s2, 8($sp)     # save $s2 on stack
sw      $s1, 4($sp)     # save $s1 on stack
sw      $s0, 0($sp)     # save $s0 on stack
```

The tail of the procedure simply reverses all these instructions, then adds a `jr` to return.

The Full Procedure `sort`

Now we put all the pieces together in Figure 2.36, being careful to replace references to registers `$a0` and `$a1` in the *for* loops with references to registers `$s2` and `$s3`. Once again to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, 9 lines of the `sort` procedure in C became 35 lines in the MIPS assembly language.

Saving registers				
	sort:	addi	$sp,$sp, -20	# make room on stack for 5 registers
		sw	$ra, 16($sp)	# save $ra on stack
		sw	$s3,12($sp)	# save $s3 on stack
		sw	$s2, 8($sp)	# save $s2 on stack
		sw	$s1, 4($sp)	# save $s1 on stack
		sw	$s0, 0($sp)	# save $s0 on stack
Procedure body				
Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0, $s3	# reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3	# reg $t0 = 0 if $t4 ≥ $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 2.34
Inner loop		addi	$s1, $s1, -1	# j -= 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop
Restoring registers				
	exit1:	lw	$s0, 0($sp)	# restore $s0 from stack
		lw	$s1, 4($sp)	# restore $s1 from stack
		lw	$s2, 8($sp)	# restore $s2 from stack
		lw	$s3,12($sp)	# restore $s3 from stack
		lw	$ra,16($sp)	# restore $ra from stack
		addi	$sp,$sp, 20	# restore stack pointer
Procedure return				
		jr	$ra	# return to calling routine

FIGURE 2.36 MIPS assembly version of procedure `sort` in Figure 2.35 on page 124.

Elaboration: One optimization that works with this example is *procedure inlining*, mentioned in Section 2.11. Instead of passing arguments in parameters and invoking the code with a `jal` instruction, the compiler would copy the code from the body of the `swap` procedure where the call to `swap` appears in the code. Inlining would avoid four

instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger if the inlined procedure is called from several locations. Such a code expansion might turn into *lower* performance if it increased the cache miss rate; see Chapter 7.

The MIPS compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement `$sp` by 16 to make room for all four argument registers (16 bytes). One reason is that C provides a `vararg` option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare `vararg`, it copies the four argument registers onto the stack into the four reserved locations.

Understanding Program Performance

Figure 2.37 shows the impact of compiler optimization on sort program performance, compile time, clock cycles, instruction count, and CPI. Note that unoptimized code has the best CPI and O1 optimization has the lowest instruction count, but O3 is the fastest, reminding us that time is the only accurate measure of program performance.

Figure 2.38 compares the impact of programming languages, compilation versus interpretation, and algorithms on performance of sorts. The fourth column shows that the unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort. Using the Just In Time Java compiler makes Java 2.1 times *faster* than the unoptimized C and within a factor of 1.13 of the highest optimized C code. (The next section gives more details on interpretation versus compilation of Java and the Java and MIPS code for Bubble Sort.) The ratios aren't as close for Quicksort in column 5, presumably because it is harder to amortize the cost of runtime compilation over the shorter execution time. The last column demonstrates the impact of a better algorithm, offering three orders of magnitude performance increase when sorting 100,000 items. Even comparing interpreted Java in column 5 to the C compiler at highest optimization in column 4, Quicksort beats Bubble Sort by a factor of 50 (0.05 × 2468 or 123 versus 2.41).

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
none	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.37 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM memory. It used Linux version 2.4.20.

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	compiler	none	1.00	1.00	2468
	compiler	O1	2.37	1.50	1562
	compiler	O2	2.38	1.50	1555
	compiler	O3	2.41	1.91	1955
Java	interpreter	—	0.12	0.05	1050
	Just In Time compiler	—	2.13	0.29	338

FIGURE 2.38 Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version. The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as Figure 2.37. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

2.14 Implementing an Object-Oriented Language

object-oriented language A programming language that is oriented around objects rather than actions, or data versus logic.

This section is for readers interested in seeing how an objected-oriented language like Java executes on a MIPS architecture. It shows the Java bytecodes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort. The rest of this section is on the CD.

2.15 Arrays versus Pointers

A challenging topic for any new programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insights about pointers. This section shows C and MIPS assembly versions of two procedures to clear a sequence of words in memory: one using array indices and one using pointers. Figure 2.39 shows the two C procedures.

The purpose of this section is to show how pointers map into MIPS instructions, and not to endorse a dated programming style. We'll see the impact of modern compiler optimization on these two procedures at the end of the section.

Array Version of Clear

Let's start with the array version, `clear1`, focusing on the body of the loop and ignoring the procedure linkage code. We assume that the two parameters `array` and `size` are found in the registers `$a0` and `$a1`, and that `i` is allocated to register `$t0`.

```
clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i += 1)
       array[i] = 0;
}

clear2(int *array, int size)
{
  int *p;
  for (p = &array[0]; p < &array[size]; p = p + 1)
       *p = 0;
}
```

FIGURE 2.39 Two C procedures for setting an array to all zeros. `Clear1` uses indices, while `clear2` uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by &, and referring to the object pointed to by a pointer is indicated by *. The declarations declare that `array` and `p` are pointers to integers. The first part of the *for* loop in `clear2` assigns the address of the first element of `array` to the pointer `p`. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of `array`. Incrementing a pointer by one, in the last part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since `p` is a pointer to integers, the compiler will generate MIPS instructions to increment `p` by four, the number of bytes in a MIPS integer. The assignment in the loop places 0 in the object pointed to by `p`.

The initialization of `i`, the first part of the *for* loop, is straightforward:

```
move    $t0,$zero       # i = 0 (register $t0 = 0)
```

To set `array[i]` to 0 we must first get its address. Start by multiplying `i` by 4 to get the byte address:

```
loop1: sll    $t1,$t0,2       # $t1 = i * 4
```

Since the starting address of the array is in a register, we must add it to the index to get the address of `array[i]` using an add instruction:

```
add    $t2,$a0,$t1    # $t2 = address of array[i]
```

(This example is an ideal situation for indexed addressing; see In More Depth in Section 2.20 on page 147.) Finally, we can store 0 in that address:

```
sw     $zero, 0($t2)  # array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment `i`:

```
addi   $t0,$t0,1      # i = i + 1
```

The loop test checks if `i` is less than `size`:

```
        slt   $t3,$t0,$a1      # $t3 = (i < size)
        bne   $t3,$zero,loop1  # if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the MIPS code for clearing an array using indices:

```
        move  $t0,$zero        # i = 0
loop1:  sll   $t1,$t0,2        # $t1 = i * 4
        add   $t2,$a0,$t1      # $t2 = address of array[i]
        sw    $zero, 0($t2)    # array[i] = 0
        addi  $t0,$t0,1        # i = i + 1
        slt   $t3,$t0,$a1      # $t3 = (i < size)
        bne   $t3,$zero,loop1 # if (i < size) go to loop1
```

(This code works as long as `size` is greater than 0.)

Pointer Version of Clear

The second procedure that uses pointers allocates the two parameters `array` and `size` to the registers `$a0` and `$a1` and allocates `p` to register `$t0`. The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array:

```
        move  $t0,$a0          # p = address of array[0]
```

The next code is the body of the *for* loop, which simply stores 0 into `p`:

```
loop2:  sw    $zero,0($t0)     # Memory[p] = 0
```

This instruction implements the body of the loop, so the next code is the iteration increment, which changes `p` to point to the next word:

```
        addi  $t0,$t0,4        # p = p + 4
```

Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers, each of which use 4 bytes, the compiler increments `p` by 4.

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 4 to get its byte address:

```
        add   $t1,$a1,$a1      # $t1 = size * 2
        add   $t1,$t1,$t1      # $t1 = size * 4
```

and then we add the product to the starting address of the array to get the address of the first word *after* the array:

```
      add  $t2,$a0,$t1       # $t2 = address of array[size]
```

The loop test is simply to see if p is less than the last element of array:

```
      slt  $t3,$t0,$t2       # $t3 = (p<&array[size])
      bne  $t3,$zero,loop2   # if (p<&array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
      move $t0,$a0           # p = address of array[0]
loop2:sw$zero,0($t0)         # Memory[p] = 0
      addi $t0,$t0,4         # p = p + 4
      add  $t1,$a1,$a1       # $t1 = size * 2
      add  $t1,$t1,$t1       # $t1 = size * 4
      add  $t2,$a0,$t1       # $t2 = address of array[size]
      slt  $t3,$t0,$t2       # $t3 = (p<&array[size])
      bne  $t3,$zero,loop2   # if (p<&array[size]) go to loop2
```

As in the first example, this code assumes size is greater than 0.

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
      move  $t0,$a0          # p = address of array[0]
      sll   $t1,$a1,2        # $t1 = size * 4
      add   $t2,$a0,$t1      # $t2 = address of array[size]
loop2:sw    $zero,0($t0)     # Memory[p] = 0
      addi  $t0,$t0,4        # p = p + 4
      slt   $t3,$t0,$t2      # $t3 = (p<&array[size])
      bne   $t3,$zero,loop2  # if (p<&array[size]) go to loop2
```

Comparing the Two Versions of Clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
      move  $t0,$zero      # i = 0
loop1:sll   $t1,$t0,2      # $t1 = i * 4
      add   $t2,$a0,$t1    # $t2 = &array[i]
      sw    $zero, 0($t2)  # array[i] = 0
      addi  $t0,$t0,1      # i = i + 1
      slt   $t3,$t0,$a1    # $t3 = (i < size)
      bne   $t3,$zero,loop1# if () go to loop1
```

```
      move  $t0,$a0        # p = & array[0]
      sll   $t1,$a1,2      # $t1 = size * 4
      add   $t2,$a0,$t1    # $t2 = &array[size]
loop2:sw    $zero,0($t0)   # Memory[p] = 0
      addi  $t0,$t0,4      # p = p + 4
      slt   $t3,$t0,$t2    # $t3=(p<&array[size])
      bne   $t3,$zero,loop2# if () go to loop2
```

The version on the left must have the "multiply" and add inside the loop because `i` is incremented and each address must be recalculated from the new index; the memory pointer version on the right increments the pointer `p` directly. The pointer version reduces the instructions executed per iteration from 7 to 4. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops).

Elaboration: The C compiler would add a test to be sure that `size` is greater than 0. One way would be to add a jump just before the first instruction of the loop to the `slt` instruction.

Understanding Program Performance

People used to be taught to use pointers in C to get greater efficiency than available with arrays: "Use pointers, even if you can't understand the code." Modern optimizing compilers can produce just as good code for the array version. Most programmers today prefer that the compiler do the heavy lifting.

Beauty is altogether in the eye of the beholder.

Margaret Wolfe Hungerford, *Molly Bawn*, 1877

2.16 Real Stuff: IA-32 Instructions

Designers of instruction sets sometimes provide more powerful operations than those found in MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence (see Section 4.8).

The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions. Section 2.17 demonstrates the pitfalls of complexity.

The Intel IA-32

MIPS was the vision of a single small group in 1985; the pieces of this architecture fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the IA-32; it is the product of several independent groups who evolved the architecture over almost 20 years, adding new features to the original

instruction set as someone might add clothing to a packed bag. Here are important IA-32 milestones:

- **1978**: The Intel 8086 architecture was announced as an assembly-language-compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike MIPS, the registers have dedicated uses, and hence the 8086 is not considered a general-purpose register architecture.

general-purpose register (GPR) A register that can be used for addresses or for data with virtually any instruction.

- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see Section 2.19 and Section 3.9).
- **1982**: The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see Chapter 7), and by adding a few instructions to round out the instruction set and to manipulate the protection model.
- **1985**: The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 7). Like the 80286, the 80386 has a mode to execute 8086 programs without change.
- **1989–95**: The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (Chapter 9) and a conditional move instruction.
- **1997**: After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX (Multi Media Extensions). This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the tradition of single instruction, multiple data (SIMD) architectures (see Chapter 9). Pentium II did not introduce any new instructions.
- **1999**: Intel added another 70 instructions, labeled SSE (Streaming SIMD Extensions) as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single-precision floating-point data type. Hence four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE included cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

- **2001**: Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double-precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable more multimedia operations, it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers like those found in other computers. This change has boosted floating-point performance on the Pentium 4, the first microprocessor to include SSE2 instructions.
- **2003**: A company other than Intel enhanced the IA-32 architecture this time. AMD announced a set of architectural extensions to increase the address space from 32 to 64 bits. Similar to the transition from a 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and increases the number of 128-bit SSE registers to 16. The primary ISA change comes from adding a new mode called *long mode* that redefines the execution of all IA-32 instructions with 64-bit addresses and data. To address the larger number of registers, it adds a new prefix to instructions. Depending how you count, long mode also adds 4 to 10 new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to IA-32 (*legacy mode*) plus a mode that restricts user programs to IA-32 but allows operating systems to use AMD64 (*compatability mode*). These modes allow a more graceful transition to 64-bit addressing than the HP/Intel IA-64 architecture.
- **2004**: Intel capitulates and embraces AMD64, relabeling it Extended Memory 64 Technology (EM64T). The major difference is that Intel added a 128-bit atomic compare and swap instruction, which probably should have been included in AMD64. At the same time, Intel announced another generation of media extensions. SSE3 adds 13 instructions to support complex arithmetic, graphics operations on arrays of structures, video encoding, floating point conversion, and thread synchronization (see Chapter 9). AMD will offer SSE3 in subsequent chips and it will almost certainly add the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.

This history illustrates the impact of the "golden handcuffs" of compatibility on the IA-32, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the IA-32, keep in mind that there are more instances of this architectural family on desktops than of any other architecture, increasing by 100 million per year. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write IA-32 programs; the goal instead is to give

you familiarity with the strengths and weaknesses of the world's most popular desktop architecture.

Rather than show the entire 16-bit and 32-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386, as this portion of the architecture is what is used. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

IA-32 Registers and Data Addressing Modes

The registers of the 80386 shows the evolution of the instruction set (Figure 2.40). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits,



FIGURE 2.40 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

FIGURE 2.41 Instruction types for the arithmetic, logical, and data transfer instructions. The IA-32 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.40 (not EIP or EFLAGS).

prefixing an *E* to their name to indicate the 32-bit version. We'll refer to them generically as GPRs (general-purpose registers). The 80386 contains only eight GPRs. This means MIPS programs can use four times as many.

The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure 2.41. There are two important differences here. The IA-32 arithmetic and logical instructions must have one operand act as both a source and a destination; MIPS allows separate registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus virtually any instruction may have one operand in memory, unlike MIPS and PowerPC.

The seven data memory-addressing modes, described in detail below, offer two sizes of addresses within the instruction. These so-called *displacements* can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restrictions on which *registers* can be used in a mode. Figure 2.42 shows the IA-32 addressing modes and which GPRs cannot be used with that mode, plus how you would get the same effect using MIPS instructions.

IA-32 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The 80386 adds 32-bit addresses and data (*double words*) in the IA-32. The data type distinctions apply to register operations as well as memory accesses. Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode and is either 16 bits or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80386 architects provide a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominates

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1) # ≤16-bit # displacement`
Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)`
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0) # ≤16-bit # displacement`

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

most programs, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus to support a semaphore (see Chapter 9), or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The IA-32 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including test, integer, and decimal arithmetic operations
3. Control flow, including conditional branches, unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location. Figure 2.43 shows some typical IA-32 instructions and their functions.

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Conditional branches on the IA-32 are based on *condition codes* or *flags*. Condition codes are set as a side effect of an operation; most are used to compare the value of a result to 0. Branches then test the condition codes. The argument for condition codes is that they occur as part of normal operations and are faster to test than it is to compare registers, as MIPS does for beq and bne. The argument against condition codes is that the compare to 0 extends the time of the operation, since it uses extra hardware after the operation, and that often the programmer must use compare instructions to test a value that is not the result of an operation. Moreover, PC-relative branch addresses must be specified in the number of bytes, since unlike MIPS, 80386 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the IA-32 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the fallacy on page 143).

Figure 2.44 lists some of the integer IA-32 instructions. Many of the instructions are available in both byte and word formats.

IA-32 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 80836 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 17 bytes.

Figure 2.45 shows the instruction format for several of the example instructions in Figure 2.43. The opcode byte usually contains a bit saying whether the operand is

Instruction	Meaning
Control	**Conditional and unconditional branches**
JNZ, JZ	Jump if condition to EIP + 8-bit offset; JNE (for JNZ), JE (for JZ) are alternative names
JMP	Unconditional jump—8-bit or 16-bit offset
CALL	Subroutine call—16-bit offset; return address pushed onto stack
RET	Pops return address from stack and jumps to it
LOOP	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
MOV	Move between two registers or between register and memory
PUSH, POP	Push source operand on stack; pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
ADD, SUB	Add source to destination; subtract source from destination; register-memory format
CMP	Compare source and destination; register-memory format
SHL, SHR, RCR	Shift left; shift logical right; rotate right with carry condition code as fill
CBW	Convert byte in 8 rightmost bits of EAX to 16-bit word in right of EAX
TEST	Logical AND of source and destination sets condition codes
INC, DEC	Increment destination, decrement destination
OR, XOR	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
MOVS	Copies from string source to destination by incrementing ESI and EDI; may be repeated
LODS	Loads a byte, word, or double word of a string into the EAX register

FIGURE 2.44 Some typical operations on the IA-32. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

8 bits or 32 bits. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form "register = register op immediate." Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m," which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled "sc, index, base."

Figure 2.46 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to fully understand which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes even the encoding of the instructions.

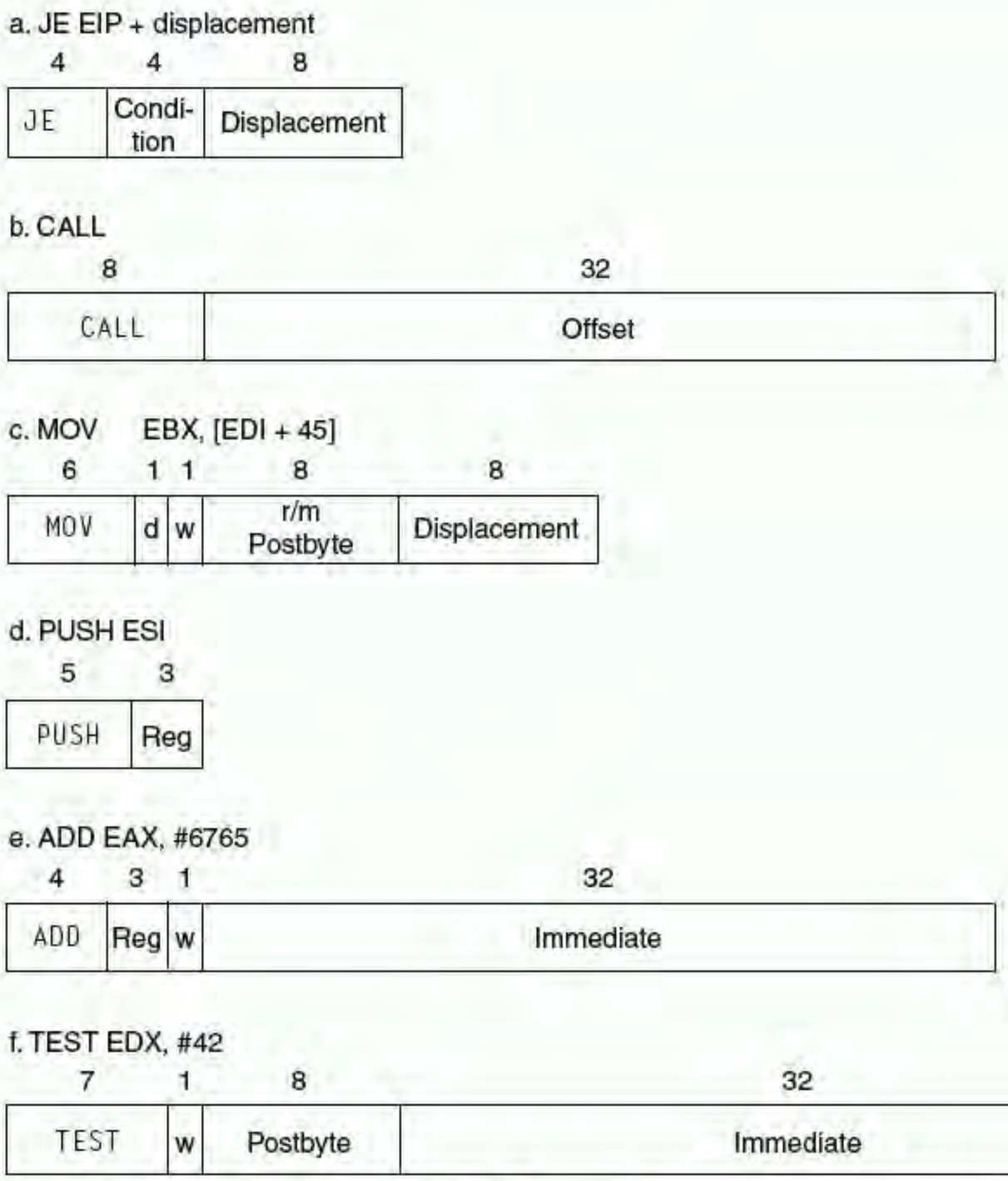


FIGURE 2.45 Typical IA-32 instruction formats. Figure 2.46 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or double word. The d field in MOV is used in instructions that may move to or from memory and shows the direction of the move. The ADD instruction requires 32 bits for the immediate field because in 32-bit mode the immediates are either 8 bits or 32 bits. The immediate field in the TEST is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 17 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

IA-32 Conclusion

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this headstart led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the IA-32 is more difficult to build than machines like MIPS, but the much

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	as
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

FIGURE 2.46 The encoding of the first address specifier of the IA-32, *"mod, reg, r/m."* The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod ≠ 3, where (sib) means use the scaled index mode shown in Figure 2.42. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

larger market means Intel can afford more resources to help overcome the added complexity. What the IA-32 lacks in style is made up in quantity, making it beautiful from the right perspective.

The saving grace is that the most frequently used IA-32 architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

2.17 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel IA-32 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the following instruction until a counter counts down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to

memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times faster. A third version, which used the larger floating-point registers instead of the integer registers of the IA-32, copies at about 2.0 times faster than the complex instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naive instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to thoroughly understand the concepts in Chapters 6 and 7 (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is one situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore such hints because the compiler does a better job at allocation than the programmer.

Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are longer time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to future machines, it also makes the software easier to maintain and allows the program to run on more brands of computers.

Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in Figure 2.16, the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

2.18 Concluding Remarks

Less is more.

Robert Browning, *Andrea del Sarto*, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid environmental scientists, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. Four design principles guide the authors of instruction sets in making that delicate balance:

1. *Simplicity favors regularity.* Regularity motivates many features of the MIPS instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.
2. *Smaller is faster.* The desire for speed is the reason that MIPS has 32 registers rather than many more.
3. *Make the common case fast.* Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for constant operands.
4. *Good design demands good compromises.* One MIPS example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. Figure 2.47 lists the MIPS instructions we have covered so far, both real and pseudoinstructions.

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.48 shows the popularity of each class of instructions for SPEC2000. The varying popularity of instructions plays an important role in the chapters on performance, datapath, control, and pipelining.

Each category of MIPS instructions is associated with constructs that appear in programming languages:

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
store byte	sb	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURE 2.47 The MIPS instruction set covered so far, with the real MIPS instructions on the left and the pseudoinstructions on the right. Appendix A (Section A.10, on page A-45) describes the full MIPS architecture. Figure 2.27 shows more details of the MIPS architecture revealed in this chapter.

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	operations in assignment statements	24%	48%
Data transfer	lw, sw, lb, sb, lui	references to data structures, such as arrays	36%	39%
Logical	and, or, nor, andi, ori, sll, srl	operations in assignment statements	18%	4%
Conditional branch	beq, bne, slt, slti	*if* statements and loops	18%	6%
Jump	j, jr, jal	procedure calls, returns, and *case/switch* statements	3%	0%

FIGURE 2.48 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for average of five SPEC2000 integer programs and five SPEC2000 floating point programs. Figure 3.26 shows the percentage of the individual MIPS instructions executed.

- The arithmetic instructions correspond to the operations found in assignment statements.
- Data transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- The conditional branches are used in *if* statements and in loops.
- The unconditional jumps are used in procedure calls and returns and for *case/switch* statements.

After we explain computer arithmetic in Chapter 3, we reveal more of the MIPS instruction set architecture.



2.19 Historical Perspective and Further Reading

This section surveys the history of instruction set architraves over time, and we give a short history of programming languages and compilers. ISAs include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of the IA-32. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes the key milestones and the pioneers who achieved them. The rest of this section is on the CD.



2.20 Exercises

Appendix A describes the MIPS simulator, which is helpful for these exercises. Although the simulator accepts pseudoinstructions, try not to use pseudoinstructions for any exercises that ask you to produce MIPS code. Your goal should be to learn the real MIPS instruction set, and if you are asked to count instructions, your count should reflect the actual instructions that will be executed and not the pseudoinstructions.

There are some cases where pseudoinstructions must be used (for example, the `la` instruction when an actual value is not known at assembly time). In many cases, they are quite convenient and result in more readable code (for example, the `li` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why.

2.1 [15] <§2.4> For More Practice: Instruction Formats

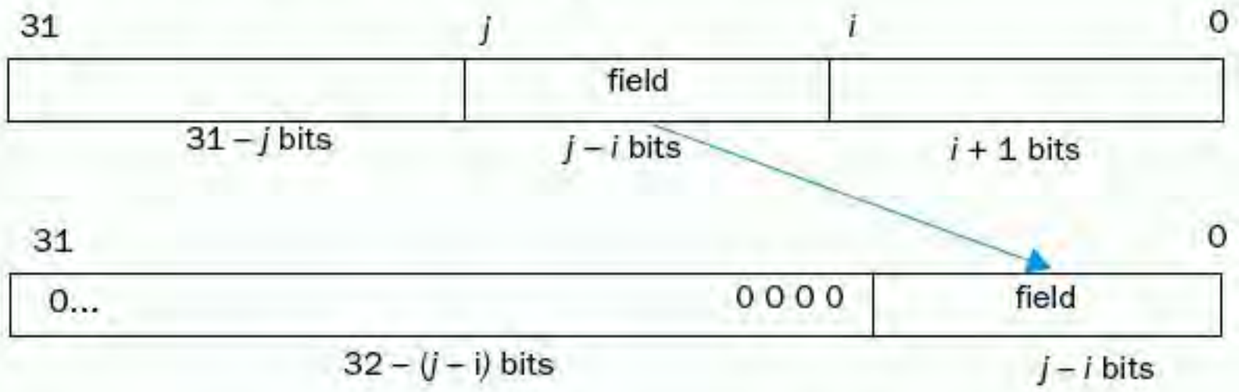
2.2 [5] <§2.4> What binary number does this hexadecimal number represent: 7fff fffa$_{hex}$? What decimal number does it represent?

2.3 [5] <§2.4> What hexadecimal number does this binary number represent: 1100 1010 1111 1110 1111 1010 1100 1110$_{two}$?

2.4 [5] <§2.4> Why doesn't MIPS have a subtract immediate instruction?

2.5 [15] <§2.5> For More Practice: MIPS Code and Logical Operations

2.6 [15] <§2.5> Some computers have explicit instructions to extract an arbitrary field from a 32-bit register and to place it in the least significant bits of a register. The figure below shows the desired operation:



Find the shortest sequence of MIPS instructions that extracts a field for the constant values $i = 5$ and $j = 22$ from register `$t3` and places it in register `$t0`. (Hint: It can be done in two instructions.)

2.7 [10] <§2.5> For More Practice: Logical Operations in MIPS

2.8 [20] <§2.5> In More Depth: Bit Fields in C

2.9 [20] <§2.5> In More Depth: Bit Fields in C

2.10 [20] <§2.5> In More Depth: Jump Tables

2.11 [20] <§2.5> In More Depth: Jump Tables

2.12 [20] <§2.5> In More Depth: Jump Tables

2.13 [10] <§2.6> Construct a control flow graph (like the one shown in Fig. 2.11) for the following section of C or Java code:

```
for (i=0; i<x; i=i+1)
    y = y + i;
```

2.14 [10] <§2.6> For More Practice: Writing Assembly Code

2.15 [25] <§2.7> Implement the following C code in MIPS, assuming that `set_array` is the first function called:

```
int i;
void set_array(int num) {
    int array[10];
    for (i=0; i<10; i++) {
        array[i] = compare(num, i);
    }
}
int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}
int sub (int a, int b) {
    return a-b;
}
```

Be sure to handle the stack and frame pointers appropriately. The variable code font is allocated on the stack, and `i` corresponds to `$s0`. Draw the status of the stack before calling `set_array` and during each function call. Indicate the names of registers and variables stored on the stack and mark the location of `$sp` and `$fp`.

2.16 [30] <§2.7> In More Depth: Tail Recursion

2.17 [30] <§2.7> In More Depth: Tail Recursion

2.18 [20] <§2.7> In More Depth: Tail Recursion

2.19 [5] <§2.8> Iris and Julie are students in computer engineering who are learning about ASCII and Unicode character sets. Help them by spelling their names and your first name in both ASCII (using decimal notation) and Unicode (using hex notation and the Basic Latin character set).

2.20 [10] <§2.8> Compute the decimal byte values that form the null-terminated ASCII representation of the following string:

```
A byte is 8 bits
```

2.21 [30] <§§2.7, 2.8> For More Practice: MIPS Coding and ASCII Strings

2.22 [20] <§§2.7, 2.8> For More Practice: MIPS Coding and ASCII Strings

2.23 [20] <§§2.7, 2.8> {Ex. 2.22} For More Practice: MIPS Coding and ASCII Strings

2.24 [30] <§§2.7, 2.8> For More Practice: MIPS Coding and ASCII Strings

2.25 <§2.8> For More Practice: Comparing C/Java to MIPS

2.26 <§2.8> For More Practice: Translating MIPS to C

2.27 <§2.8> For More Practice: Understanding MIPS Code

2.28 <§2.8> For More Practice: Understanding MIPS Code

2.29 [5] <§§2.3, 2.6, 2.9> Add comments to the following MIPS code and describe in one sentence what it computes. Assume that `$a0` and `$a1` are used for the input and both initially contain the integers *a* and *b*, respectively. Assume that `$v0` is used for the output.

```
        add   $t0, $zero, $zero
loop:   beq   $a1, $zero, finish
        add   $t0, $t0, $a0
        sub   $a1, $a1, 1
        j     loop
finish: addi  $t0, $t0, 100
        add   $v0, $t0, $zero
```

2.30 [12] <§§2.3, 2.6, 2.9> The following code fragment processes two arrays and produces an important value in register `$v0`. Assume that each array consists of 2500 words indexed 0 through 2499, that the base addresses of the arrays are stored in `$a0` and `$a1` respectively, and their sizes (2500) are stored in `$a2` and `$a3`, respectively. Add comments to the code and describe in one sentence what this code does. Specifically, what will be returned in `$v0`?

```
        sll   $a2, $a2, 2
        sll   $a3, $a3, 2
        add   $v0, $zero, $zero
        add   $t0, $zero, $zero
outer:  add   $t4, $a0, $t0
```

```
        lw      $t4, 0($t4)
        add     $t1, $zero, $zero
inner:  add     $t3, $a1, $t1
        lw      $t3, 0($t3)
        bne     $t3, $t4, skip
        addi    $v0, $v0, 1
skip:   addi$   t1, $t1, 4
        bne     $t1, $a3, inner
        addi    $t0, $t0, 4
        bne     $t0, $a2, outer
```

2.31 [10] <§§2.3, 2.6, 2.9> Assume that the code from Exercise 2.30 is run on a machine with a 2 GHz clock that requires the following number of cycles for each instruction:

Instruction	Cycles
add,addi,sll	1
lw, bne	2

In the worst case, how many seconds will it take to execute this code?

2.32 [5] <§2.9> Show the single MIPS instruction or minimal sequence of instructions for this C statement:

```
b = 25 | a;
```

Assume that a corresponds to register $t0 and b corresponds to register $t1.

2.33 [10] <§2.9> For More Practice: Translating from C to MIPS

2.34 [10] <§§ 2.3, 2.6, 2.9> The following program tries to copy words from the address in register $a0 to the address in register $a1, counting the number of words copied in register $v0. The program stops copying when it finds a word equal to 0. You do not have to preserve the contents of registers $v1, $a0, and $a1. This terminating word should be copied but not counted.

```
      addi $v0, $zero, 0 # Initialize count
loop: lw   $v1, 0($a0)   # Read next word from source
      sw   $v1, 0($a1)   # Write to destination
      addi $a0, $a0, 4   # Advance pointer to next source
      addi $a1, $a1, 4   # Advance pointer to next destination
```

```
beq $v1, $zero, loop  # Loop if word copied != zero
```

There are multiple bugs in this MIPS program; fix them and turn in a bug-free version. Like many of the exercises in this chapter, the easiest way to write MIPS programs is to use the simulator described in Appendix A.

2.35 [10] <§§2.2, 2.3, 2.6, 2.9> For More Practice: Reverse Translation from MIPS to C

2.36 <§2.9> For More Practice: Translating from C to MIPS

2.37 [25] <§2.10> As discussed on page 107 (Section 2.10, "Assembler"), pseudoinstructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudoinstruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences. In the following table, `big` refers to a specific number that requires 32 bits to represent and `small` to a number that can fit in 16 bits.

Pseudoinstruction	What it accomplishes
`move $t1, $t2`	`$t1 = $t2`
`clear $t50`	`$t0 = 0`
`beq $t1, small, L`	`if ($t1 = small) go to L`
`beq $t2, big, L`	`if ($t2 = big) go to L`
`li $t1, small`	`$t1 = small`
`li $t2, big`	`$t2 = big`
`ble $t3, $t5, L`	`if ($t3 <= $t5) go to L`
`bgt $t4, $t5, L`	`if ($t4 > $t5) go to L`
`bge $t5, $t3, L`	`if ($t5 >= $t3) go to L`
`addi $t0, $t2, big`	`$t0 = $t2 + big`
`lw $t5, big($t2)`	`$t5 = Memory[$t2 + big]`

2.38 [5] <§§2.9, 2.10> Given your understanding of PC-relative addressing, explain why an assembler might have problems directly implementing the branch instruction in the following code sequence:

```
here:               beq   $s0, $s2, there
...
there               add   $s0, $s0, $s0
```

Show how the assembler might rewrite this code sequence to solve these problems.

2.39 <§2.10> For More Practice: MIPS Pseudoinstructions

2.40 <§2.10> For More Practice: Linking MIPS Code

2.41 <§2.10> For More Practice: Linking MIPS Code

2.42 [20] <§2.11>Find a large program written in C (for example, gcc, which can be obtained from *http://gcc.gnu.org*) and compile the program twice, once with optimizations (use `-03`) and once without. Compare the compilation time and run time of the program. Are the results what you expect?

2.43 [20] <§2.12> For More Practice: Enhancing MIPS Addressing Modes

2.44 [10] <§2.12> For More Practice: Enhancing MIPS Addressing Modes

2.45 [10] <§2.12> In More Depth: The IBM/Motorola versus MIPS in C

2.46 [15] <§§2.6, 2.13> The MIPS translation of the C (or Java) segment

```
while (save[i] == k)
    i += 1;
```

on page 129 (Section 2.6, "Compiling a While Loop in C") uses both a conditional branch and an unconditional jump each time through the loop. Only poor compilers would produce code with this loop overhead. Assuming that this code is in Java (not C), rewrite the assembly code so that it uses at most one branch or jump each time through the loop. Additionally, add code to perform the Java checking for index out of bounds and ensure that this code uses at most one branch or jump each time through the loop. How many instructions are executed before and after the optimization if the number of iterations of the loop is 10 and the value of `i` is never out of bounds?

2.47 [30] <§§2.6, 2.13> Consider the following fragment of Java code:

```
for (i=0; i<=100; i=i+1)
    a[i] = b[i] + c;
```

Assume that `a` and `b` are arrays of words and the base address of `a` is in `$a0` and the base address of `b` is in `$a1`. Register `$t0` is associated with variable `i` and register `$s0` with the value of `c`. You may also assume that any address constants you need are available to be loaded from memory. Write the code for MIPS. How many instructions are executed during the running of this code if there are no array out-of-bounds exceptions thrown? How many memory data references will be made during execution?

2.48 [5] <§2.13> Write the MIPS code for the Java method `compareTo` (found in Figure 2.35 on page 124).

2.49 [15] <§2.17> When designing memory systems, it becomes useful to know the frequency of memory reads versus writes as well as the frequency of accesses for instructions versus data. Using the average instruction mix information for MIPS for the program SPEC2000int in Figure 2.48 (on page 141), find the following:

a. The percentage of all memory accesses (both data and instruction) that are for data.

b. The percentage of all memory accesses (both data and instruction) that are for reads. Assume that two-thirds of data transfers are loads.

2.50 [10] <§2.17> Perform the same calculations as for Exercise 2.49, but replace the program SPEC2000int with SPEC2000fp.

2.51 [15] <§2.17> Suppose we have made the following measurements of average CPI for instructions:

Instruction	Average CPI
Arithmetic	1.0 clock cycles
Data transfer	1.4 clock cycles
Conditional branch	1.7 clock cycles
Jump	1.2 clock cycles

Compute the effective CPI for MIPS. Average the instruction frequencies for SPEC2000int and SPEC2000fp in Figure 2.48 on page 146 to obtain the instruction mix.

2.52 [20] <§2.18> In More Depth: Instruction Set Styles

2.53 [20] <§2.18> In More Depth: Instruction Set Styles

2.54 [10] <§2.18> In More Depth: The Single Instruction Computer

2.55 [20] <§2.18> In More Depth: The Single Instruction Computer

2.56 [5] <§2.19>The stored-program concept, introduced in the late 1940s, brought about a significant change in how computers were designed and operated. What is a possible example of a nonstored-program machine, and what are the problems with such a machine? How can these problems be overcome by a stored-program machine?

2.57 [5] <§2.19> In More Depth: The IBM/Motorola versus MIPS in C

2.58 [15] <§2.19> In More Depth: The IBM/Motorola versus MIPS in C

2.59 [15] <§2.19> In More Depth: Logical Instructions

Computers in the Real World

Helping Save Our Environment with Data

Problem to solve: Monitor plants and animals of our environment to collect information that may influence environmental polices.

Solution: Develop rugged, battery-operated, embedded computers with sensors, wireless communication, and appropriate software.

Stanford biologist Barbara Block studies bluefin tuna. One policy question was whether the tuna on one side of the Atlantic are different from those on the other side. If so, then each region could set its own quotas. If not, then we need oceanwide quotas.

To answer this question, she started implanting tuna with devices that could monitor their journeys. Every two minutes a pop-up satellite archival tag (PSAT) records water pressure, ambient light, temperature, time of day, and other measurements. Data are saved in 1 MB of flash memory. The onboard 8-bit microprocessor estimates depth from the water pressure. It finds longitude using light intensity data and time of day. It determines sunrise, sunset, and therefore high noon, and calculates the time shift between local noon and Greenwich Mean Time noon, like a navigator using a sextant and chronometer. The water temperature is later matched to satellite records to determine latitude. Block does not rely on fishermen to catch the tuna and return PSATs. A PSAT is attached to a fish with a pin that dissolves via electrolysis after the computer turns on a battery. The tag then floats to the surface and begins transmitting data to satellites. The floating tag can transmit for up to two weeks, sending the data directly to Block's lab.



Block and students tag a bluefin tuna, which can grow to 2000 pounds and 10 feet in length.



A pop-up archival satellite tag and internal electronics.

Block discovered that bluefin tuna travel more than 10,000 miles per year; tuna tagged near the East Coast of the United States will cross the Atlantic and spawn in both the Gulf of Mexico and the Eastern Mediterranean. Her discovery changed regulations so that tuna are no longer managed separately in the Eastern and Western Atlantic. She is now developing a census of Pacific marine life using smaller tags for smaller animals and tags that transmit each time a fish surfaces. She speculates that tagged tuna could be ideal "vehicles" to monitor ocean change.

Berkeley biologist Todd Dawson studies the ecology of the coastal redwood, *Sequoia sempervirens,* particularly the interaction of sea fog with trees. For years his research involved installing 50 kilograms of gear and kilometers of wire strung to sensors. This work is often done more than 80 meters above the ground. Data could only be retrieved by climbing up to a printer-sized data logger.

Berkeley computer scientist David Culler proposed a new approach. Dawson is now placing miniature wireless sensors the size of film canisters in these trees. Each micromote is less than 3 cubic inches, can transmit up to 40 KB/sec, and can run for months on a C battery. Since micromotes are small and cheap, many can be placed in a tree. Data is collected with a compatible laptop by simply walking to the base of the tree.

Dawson found that summertime fog accounts for 25% to 40% of the water that the redwoods receive for the whole year. The trees may even be drinking water directly from fog via a symbiotic relationship with fungi living on their leaves.

Dawson predicts wireless sensor networks will change the way people do ecological research.



Professor Dawson and student climbing a sequoia to install fog monitors.

To learn more see these references on the library

Block et al., "Migratory movements, depth preferences, and thermal biology of atlantic bluefin tuna," *Science* 293: 1310–14, 2001

"Redwoods," Prof. Dawson's laboratory site

"Redwood's drinking water from fog," *The Forestry Source*, Nov. 2002

"Tagging of the Pacific Pelagics," www.toppcensus.org



The Mica micromote with C battery. It is about the size of a film canister.

3

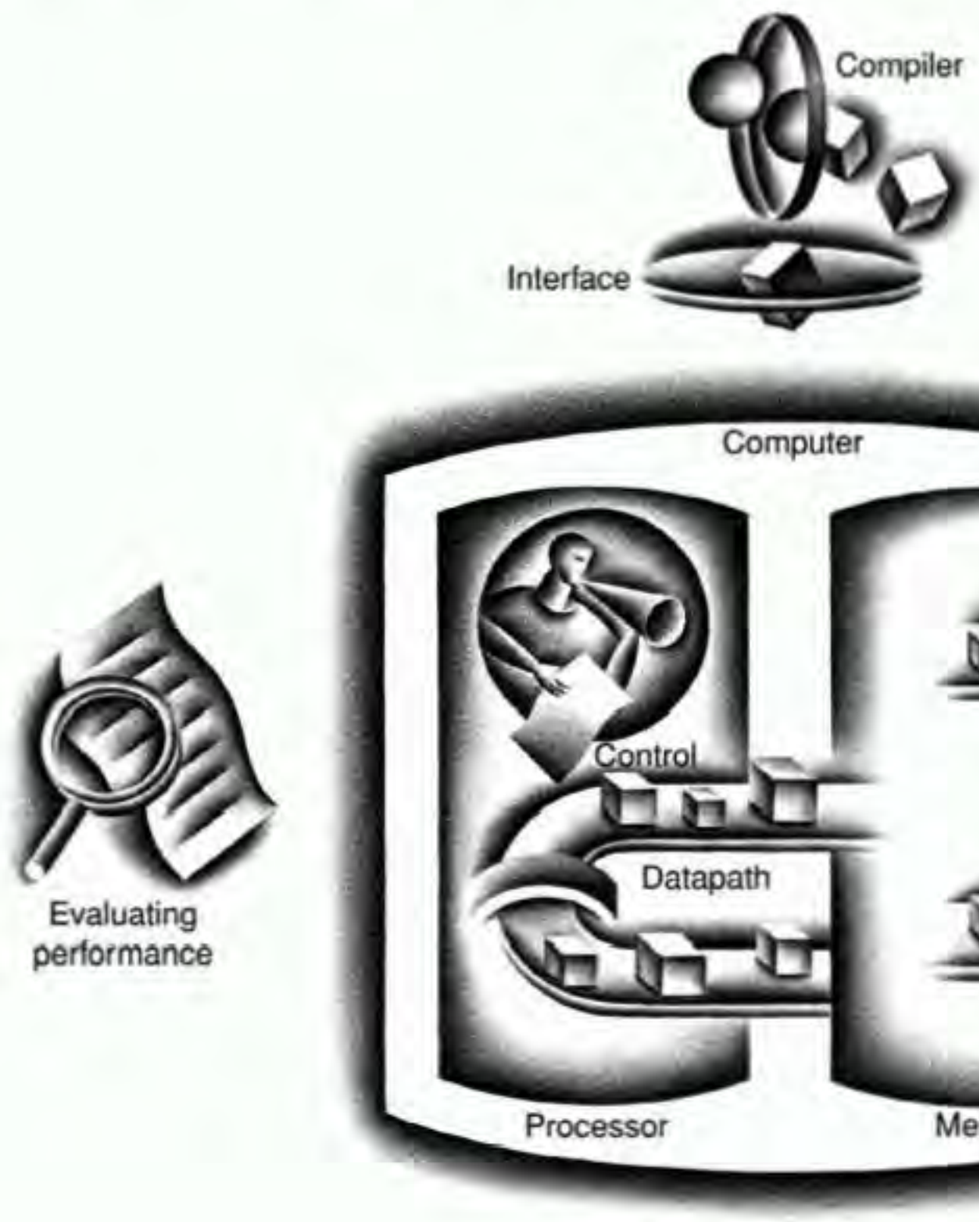
Arithmetic for Computers

Numerical precision
is the very soul
of science.

Sir D'arcy Wentworth Thompson
On Growth and Form, 1917

3.1 **Introduction** 160
3.2 **Signed and Unsigned Numbers** 160
3.3 **Addition and Subtraction** 170
3.4 **Multiplication** 176
3.5 **Division** 183
3.6 **Floating Point** 189
3.7 **Real Stuff: Floating Point in the IA-32** 217
3.8 **Fallacies and Pitfalls** 220
3.9 **Concluding Remarks** 225
3.10 **Historical Perspective and Further Reading** 229
3.11 **Exercises** 229

The Five Classic Components of a Computer



3.1 Introduction

Computer words are composed of bits; thus words can be represented as binary numbers. Although the natural numbers 0, 1, 2, and so on can be represented either in decimal or binary form, what about the other numbers that commonly occur? For example:

- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if an operation creates a number bigger than can be represented?
- What about fractions and real numbers?

And underlying all these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel this mystery, including representation of numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may even explain quirks that you have already encountered with computers. (If you are familiar with signed binary numbers, you may wish to skip the next section and go to Section 3.3 on page 170.)

3.2 Signed and Unsigned Numbers

Numbers can be represented in any base; humans prefer base 10 and, as we examined in Chapter 2, base 2 is best for computers. To avoid confusion we subscript decimal numbers with *ten* and binary numbers with *two*.

In any number base, the value of *i*th digit *d* is

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

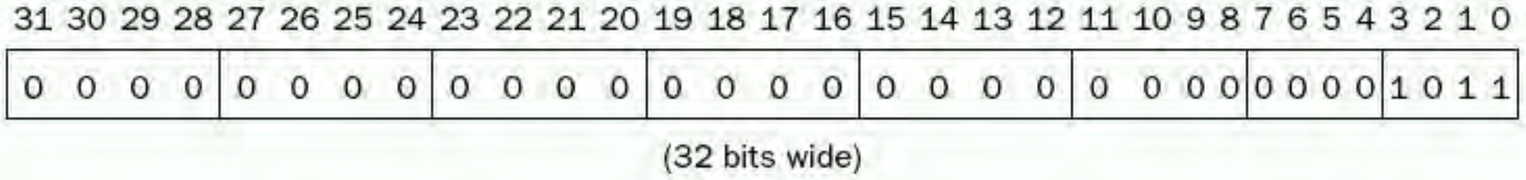
where *i* starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: Simply use the power of the base for that bit. For example,

$$1011_{\text{two}}$$

represents

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

Hence the bits are numbered 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}:



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*):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^1) + (x0 \times 2^0)$$

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.

EXAMPLE

ASCII versus Binary Numbers

We could represent numbers as strings of ASCII digits instead of as integers (see Figure 2.21 on page 91). 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 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.

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 operating system and 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 negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude 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:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

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 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 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 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

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. For similar reasons, *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.

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.

Comparison instructions must deal with this dichotomy. 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 take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking in a few pages.)

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`).

Hardware Software Interface

Signed versus Unsigned Comparison

EXAMPLE

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 if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1 in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Before going on to addition and subtraction, let's examine a few 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 $111\ldots111_{two}$, which represents –1. Since $x + x \equiv -1$, therefore $x + x + 1 = 0$ or $x + 1 = -x$.

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,

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

The second 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*.

Sign Extension Shortcut

EXAMPLE

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

$$\begin{array}{r} 1111\ 1111\ 1111\ 1101_{two} \\ +\qquad\qquad\qquad\quad 1_{two} \\ \hline \end{array}$$

$$= 1111\ 1111\ 1111\ 1110_{two}$$

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 those that are 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.

The third shortcut reduces the cost 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.

EXAMPLE

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$a1` ≥ `$t2` or if `$a1` is negative.

ANSWER

The checking code just uses `sltu` to do both checks:

```
sltu $t0,$a1,$t2 # Temp reg $t0=0 if k>=length or k<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

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. Figure 3.1 shows the additions to the MIPS assembly language revealed in this section. (The MIPS machine language is also illustrated on the back endpapers of this book.)

Check Yourself

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 (which uses Unicode)

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for 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, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 - $s3	Three operands
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	+ constant
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	`lhu $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte unsigned	`lbu $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1,100`	$s1 = 100 * 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 $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR 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; two's complement
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned numbers
Uncondi-tional 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 3.1 MIPS architecture revealed thus far. Color indicates portions from this section added to the MIPS architecture revealed in Chapter 2 (Figure 3.26 on page 228). MIPS machine language is listed in the MIPS summary reference card in the front of this book.

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 $2^n - x$.

A third alternative representation 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 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, is to represent the most negative value by $00\ldots000_{two}$ and the most positive value represented 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.

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.

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 bit strings can encode the sign, and hence we do not normally use "+" or "–" with binary or hexadecimal notation.

Subtraction: Addition's Tricky Pal

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., *Book of Top Ten Lists*, 1990

3.3 Addition and Subtraction

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: The appropriate operand is simply negated before being added.

EXAMPLE

Binary Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten} \end{array}$$

ANSWER

The 4 bits to the right have all the action; Figure 3.2 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

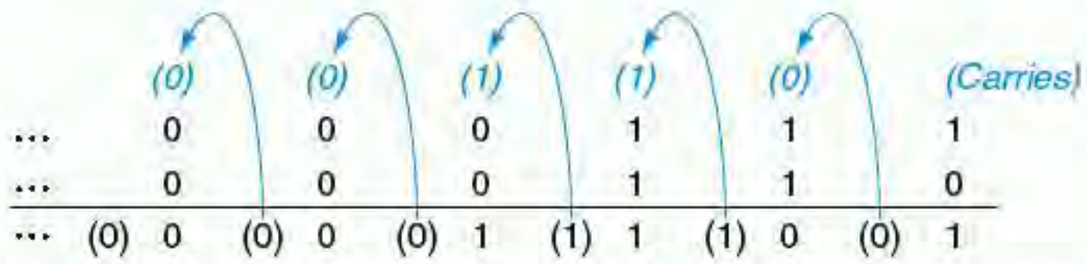


FIGURE 3.2 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

or via addition using the two's complement representation of −6:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

We said earlier that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, −10 + 4 = −6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: When the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $x - y = x + (-y)$ because we subtract by negating the second operand and then add. So, when we subtract operands of

the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Having examined when overflow cannot occur in addition and subtraction, we still haven't answered how to detect when it *does* occur. Overflow occurs when adding two positive numbers and the sum is negative, or vice versa. Clearly, adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs the sign bit is being set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. This means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means a borrow occurred from the sign bit. Figure 3.3 shows the combination of operations, operands, and results that indicate an overflow.

We have just seen how to detect overflow for two's complement numbers in a computer. What about unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu` no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.3 Overflow conditions for addition and subtraction.

The computer designer must decide how to handle arithmetic overflows. Although some languages like C ignore integer overflow, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.

MIPS detects overflow with an **exception**, also called an **interrupt** on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 5.6 covers exceptions in more detail; Chapters 7 and 8 describe other situations where exceptions and interrupts occur.)

MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

Hardware Software Interface

exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Elaboration: MIPS can trap on overflow, but unlike many other computers there is no conditional branch to test overflow. A sequence of MIPS instructions can discover overflow. For signed addition, the sequence is the following (see the In More Depth segment on logical instruction in Chapter 2 for the definition of the `xor` instructions):

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
so no overflow
xor  $t3, $t0, $t1 # signs =; sign of sum match too?
                   # $t3 negative if sum sign different
slt  $t3, $t3, $zero # $t3 = 1 if sum sign different
bne  $t3, $zero, Overflow # All three signs ≠; go to
overflow
```

For unsigned addition ($t0 = $t1 + $t2), the test is

```
addu $t0, $t1, $t2 # $t0 = sum
nor  $t3, $t1, $zero # $t3 = NOT $t1
        # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2 # (2^32 - $t1 - 1) < $t2
       #⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1 < $t1 + $t2) go to
overflow
```

Summary

The main point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require detection of overflow, so today all computers have a way to detect it. Figure 3.4 shows the additions to the MIPS architecture from this section.

Check Yourself Some programming languages allow two's complement integer arithmetic on variables declared byte and half. What MIPS instructions would be used?

1. Load with `lbu`, `lhu`; arithmetic with `add`, `sub`, `mult`, `div`; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mult`, `div`; then store using `sb`, `sh`.
3. Loads with `lb`, `lh`; arithmetic with `add`, `sub`, `mult`, `div`, using `and` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.

Elaboration: In the preceding text, we said that you copy EPC into a register via `mfc0` and then return to the interrupted code via jump register. This leads to an interesting question: Since you must first transfer EPC to a register to use with jump register, how can jump register return to the interrupted code *and* restore the original values of *all* registers? You either restore the old registers first, thereby destroying your return address from EPC that you placed in a register for use in jump register, or you restore all registers but the one with the return address so that you can jump—meaning an exception would result in changing that one register at any time during program execution! Neither option is satisfactory.

To rescue the hardware from this dilemma, MIPS programmers agreed to reserve registers `$k0` and `$k1` for the operating system; these registers are *not* restored on exceptions. Just as the MIPS compilers avoid using register `$at` so that the assembler

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow detected
	add unsigned	`addu $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow undetected
	subtract unsigned	`subu $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands; overflow undetected
	add immediate unsigned	`addiu $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow undetected
	move from coprocessor register	`mfc0 $s1,$epc`	`$s1 = $epc`	Used to copy Exception PC plus other special registers
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load half unsigned	`lhu $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Halfword memory to register
	store half	`sh $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Halfword register to memory
	load byte unsigned	`lbu $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1,100`	`$s1 = 100 * 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 $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR 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; two's complement
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; two's complement
	set less than unsign	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; unsigned
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; unsigned
Uncondi-tional 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 3.4 MIPS architecture revealed thus far. To save space in the table, it does not include the registers and memory found in Figure 3.1 on page 169. Color indicates the portions revealed since Figure 3.1. MIPS machine language is also on the MIPS reference summary card.

can use it as a temporary register (see the Hardware Software Interface section on page 96 in Chapter 2), compilers also abstain from using registers `$k0` and `$k1` to make them available for the operating system. Exception routines place the return address in one of these registers and then use jump register to restore the instruction address.

Multiplication is vexation,
Division is as bad;
The rule of three doth puzzle me,
And practice drives me mad.

Anonymous, Elizabethan manuscript, 1570

3.4 Multiplication

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiply.

But first let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. Multiplying 1000_{ten} by 1001_{ten}:

```
Multiplicand                    1000ten
Multiplier              X       1001ten
                        ---------------
                         1000
                        0000
                       0000
                      1000
                      ---------------
Product                 1001000ten
```

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier and shifting the intermediate product one digit to the left of the earlier intermediate products.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is $n + m$ bits long. That is, $n + m$ bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In this example we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Although the decimal example above happened to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

Sequential Version of the Multiplication Algorithm and Hardware

This design mimics the algorithm we learned in grammar school; the hardware is shown in Figure 3.5. We have drawn the hardware so that data flows from top to bottom to more closely resemble the paper-and-pencil method.

Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step as it may be added to the intermediate products. Over 32 steps a 32-bit multiplicand would move 32 bits to the left. Hence we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and 0 in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

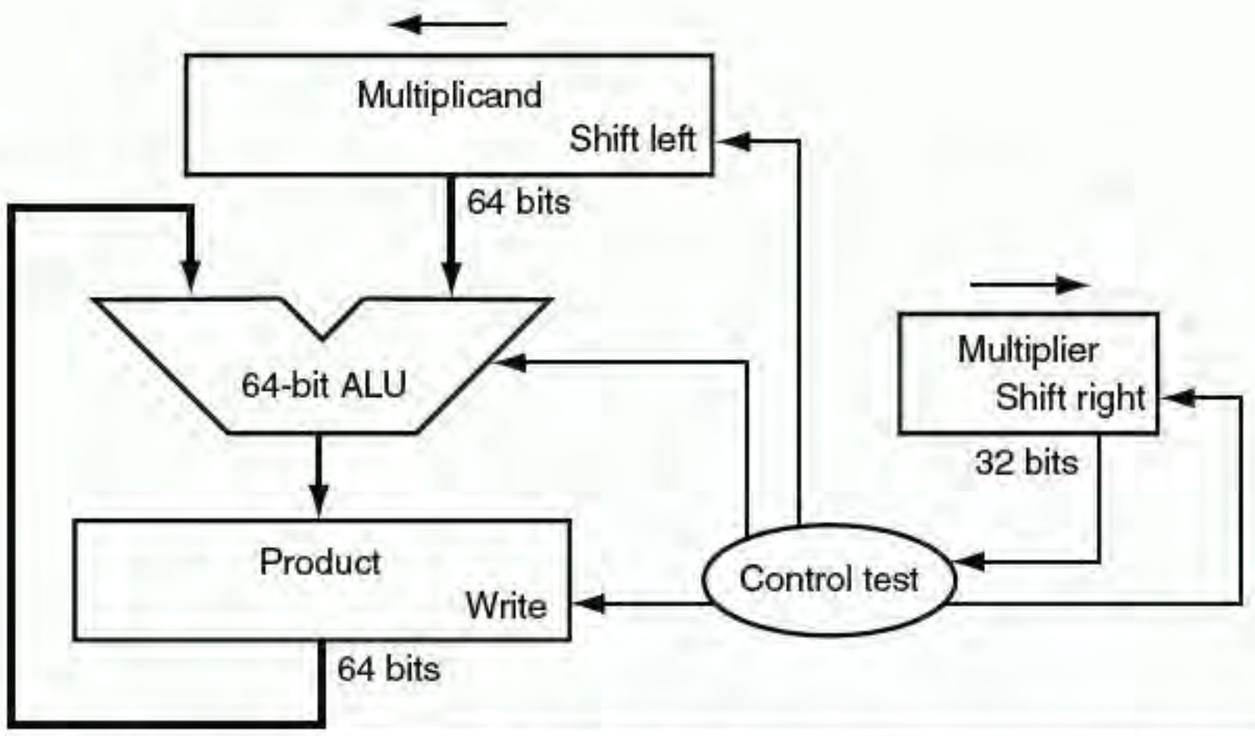


FIGURE 3.5 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Figure 3.6 shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product. If each step took a

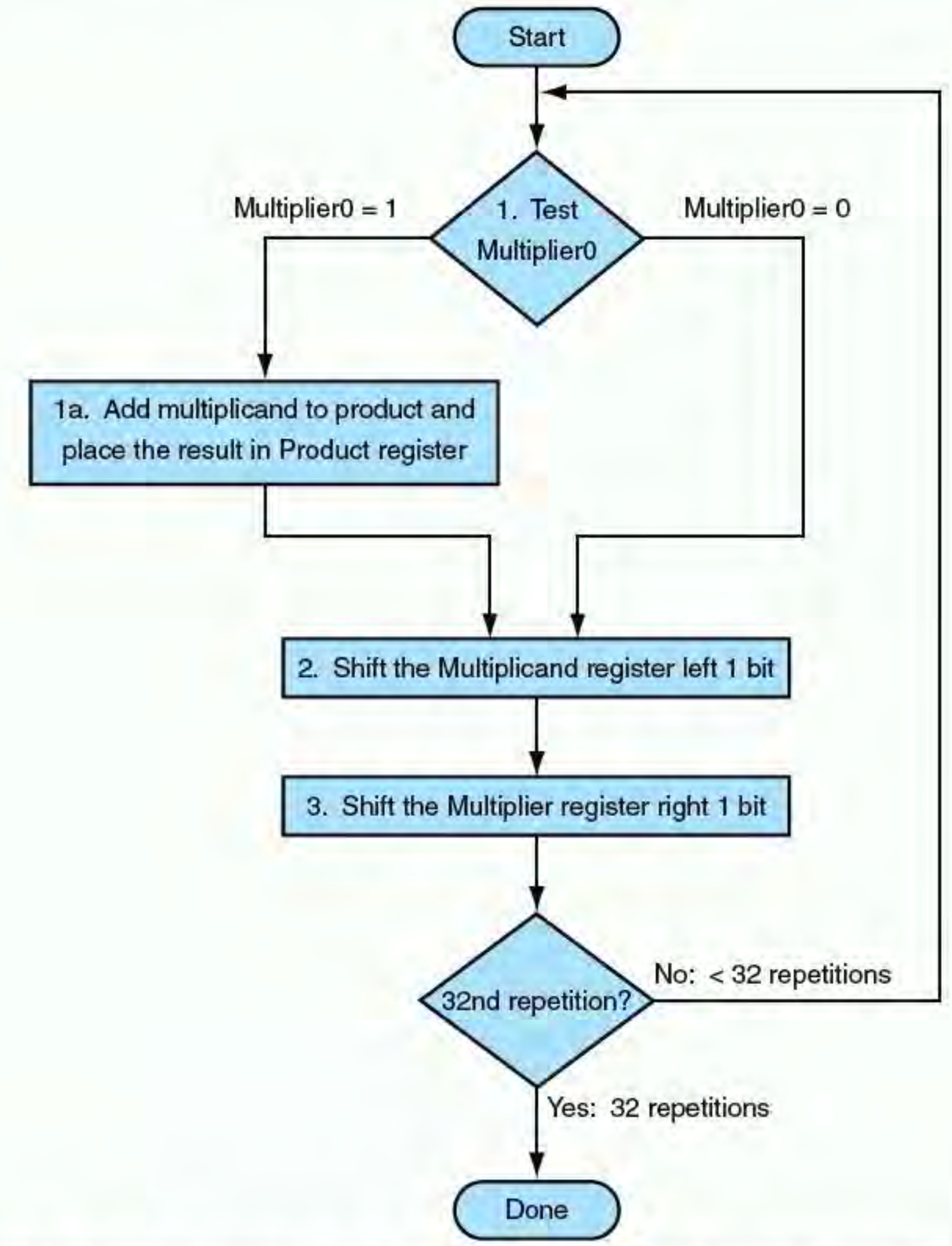


FIGURE 3.6 The first multiplication algorithm, using the hardware shown in Figure 3.5. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's law (see Chapter 4, page 267) reminds us that even a moderate frequency for a slow operation can limit performance.

This algorithm and hardware are easily refined to take 1 clock cycle per step. The speedup comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a one. The hardware just has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.7 shows the revised hardware.

Hardware Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. As mentioned in Chapter 2, almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

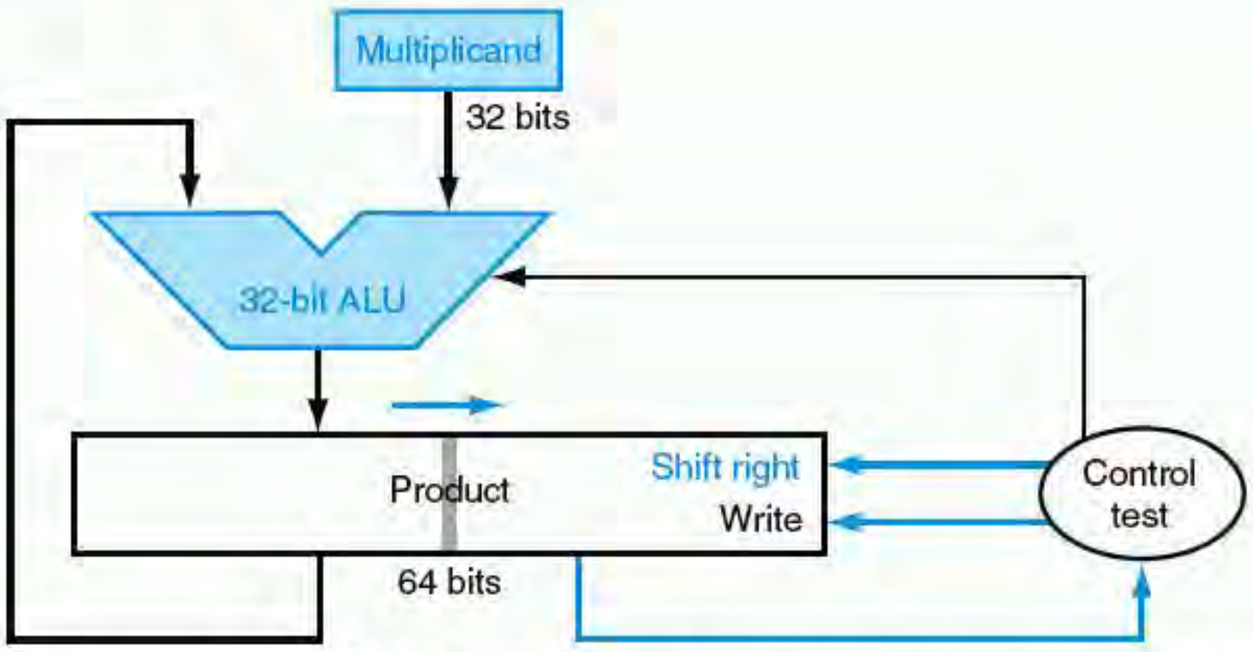


FIGURE 3.7 Refined version of the multiplication hardware. Compare with the first version in Figure 3.5. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color.

EXAMPLE

ANSWER

A Multiply Algorithm

Using 4-bit numbers to save space, multiply $2_{ten} \times 3_{ten}$, or $0010_{two} \times 0011_{two}$.

Figure 3.8 shows the value of each register for each of the steps labeled according to Figure 3.6, with the final value of $0000\ 0110_{two}$ or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Signed Multiplication

So far we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

It turns out that the last algorithm will work for signed numbers provided that we remember that the numbers we are dealing with have infinite digits, and that we are only representing them with 32 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.8 Multiply example using algorithm in Figure 3.6. The bit examined to determine the next step is circled in color.

Faster Multiplication

Moore's Law has provided so much more in resources that hardware designers can now build a much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit and the other is the output of a prior adder. Figure 3.9 shows how they would be connected.

Why is this much hardware faster? The sequential multiplier pays the overhead of a clock for each bit of the product. This multiplier array of adders does not. A second reason is this large collection of adders lends itself to many optimizations to gain further improvements. One example is using *carry save adders* to add such a large column of numbers; see Exercises 3.24 and 3.49. A third reason is that it is easy to pipeline such a design to be able to support many multiplies simultaneously (see Chapter 6).

Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*. To produce a properly signed or unsigned product, MIPS has two instructions: multiply (`mult`) and multiply unsigned (`multu`). To fetch the integer 32-bit product, the programmer uses *move from lo* (`mflo`). The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

Summary

Multiplication is accomplished by simple shift and add hardware, derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of two.

Hardware Software Interface

Both MIPS multiply instructions ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits. There is no overflow if Hi is 0 for `multu` or the replicated sign of Lo for `mult`. The instruction *move from hi* (`mfhi`) can be used to transfer Hi to a general-purpose register to test for overflow.

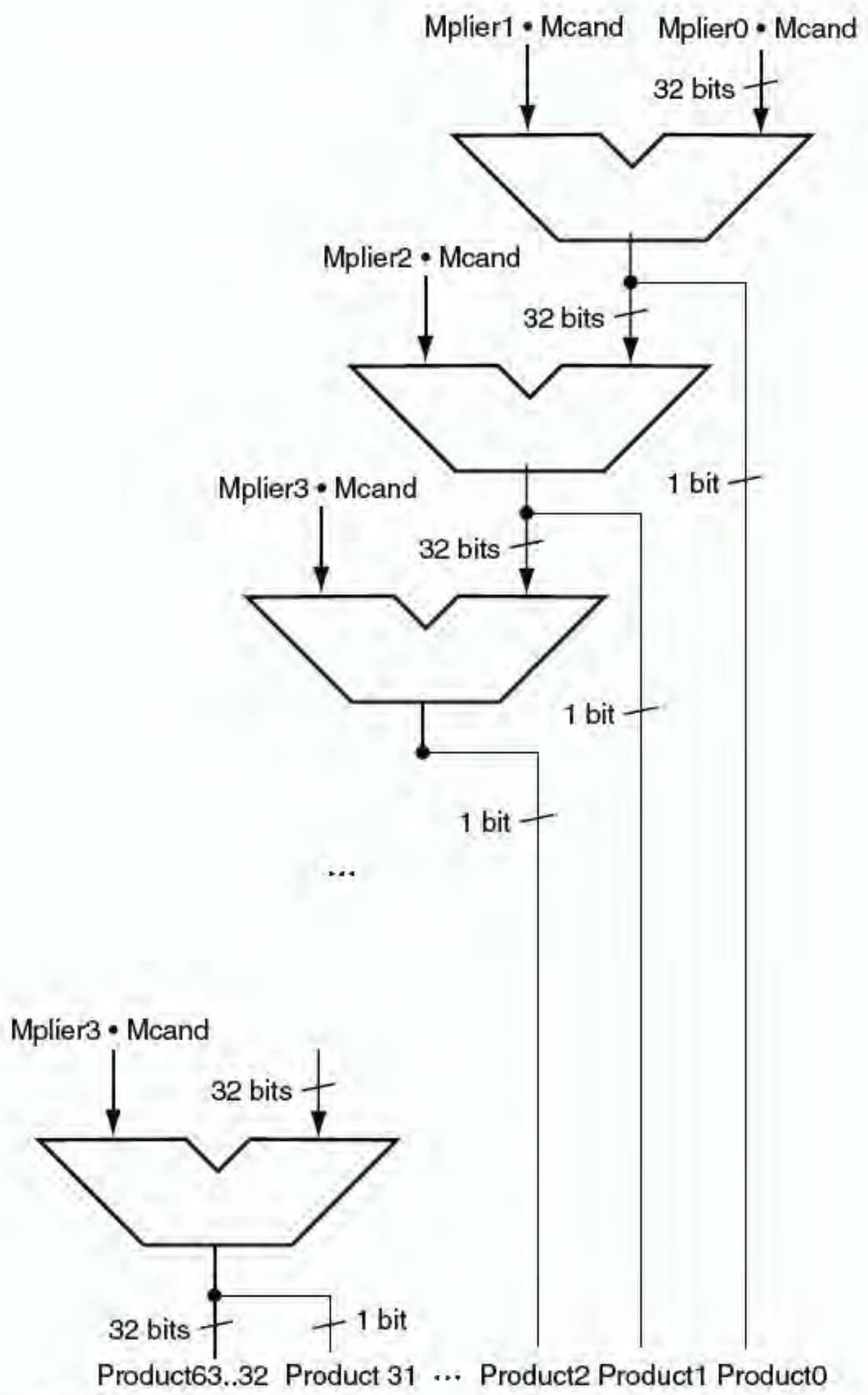


FIGURE 3.9 Fast multiplication hardware. Rather than use a single 32-bit adder 32 times, this hardware "unrolls the loop" to use 32 adders. Each adder produces a 32-bit sum and a carry out. The least significant bit is a bit of the product, and the carry out and the upper 31 bits of the sum are passed along to the next adder.

3.5 Division

Divide et impera.

Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let's start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & -1000 & \\
 & \overline{10} & \\
 & 101 & \\
 & 1010 & \\
 & -1000 & \\
 & \overline{10_{ten}} & \text{Remainder}
\end{array}
$$

The two operands *(***dividend** and **divisor***)* and the result *(***quotient***)* of divide are accompanied by a second result called the **remainder**. Here is another way to express the relationship between the components:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient.

The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses only the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend: it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

Let's assume that both the dividend and divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 32-bit values, and we will ignore the sign for now.

dividend A number being divided.

divisor A number that the dividend is divided by.

quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

A Division Algorithm and Hardware

Figure 3.10 shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

Figure 3.11 shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

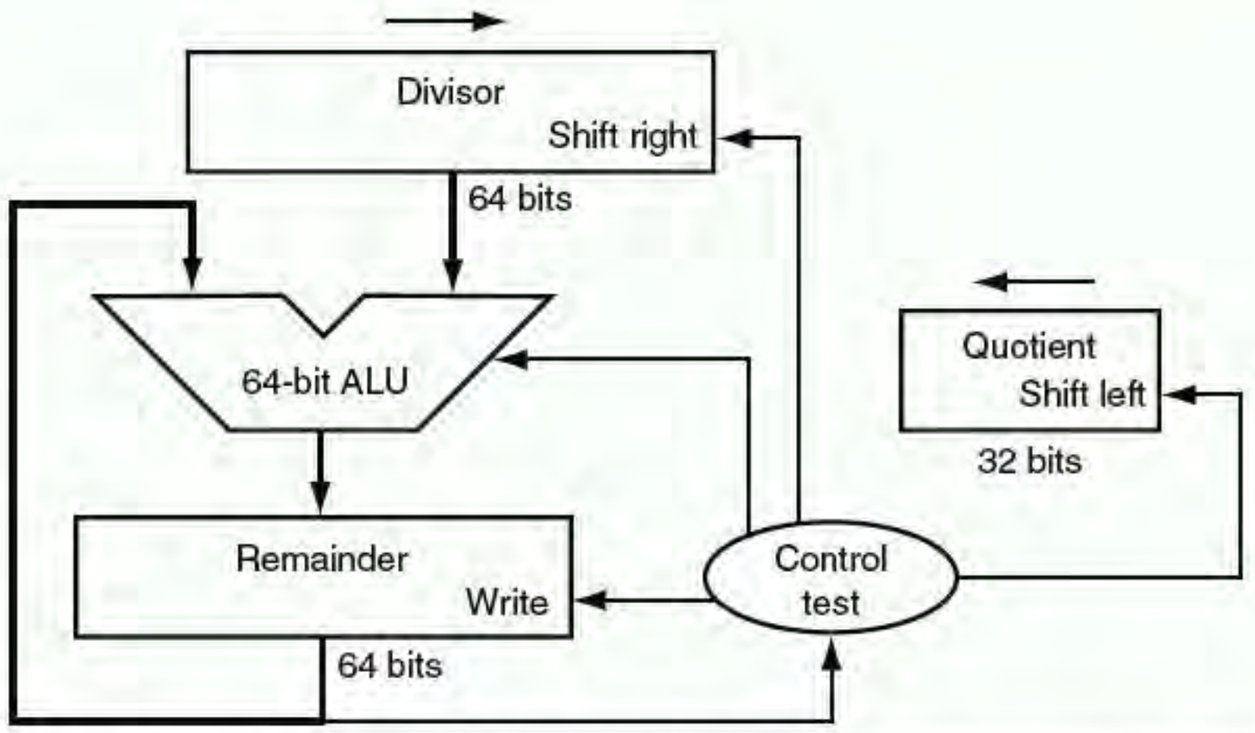


FIGURE 3.10 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

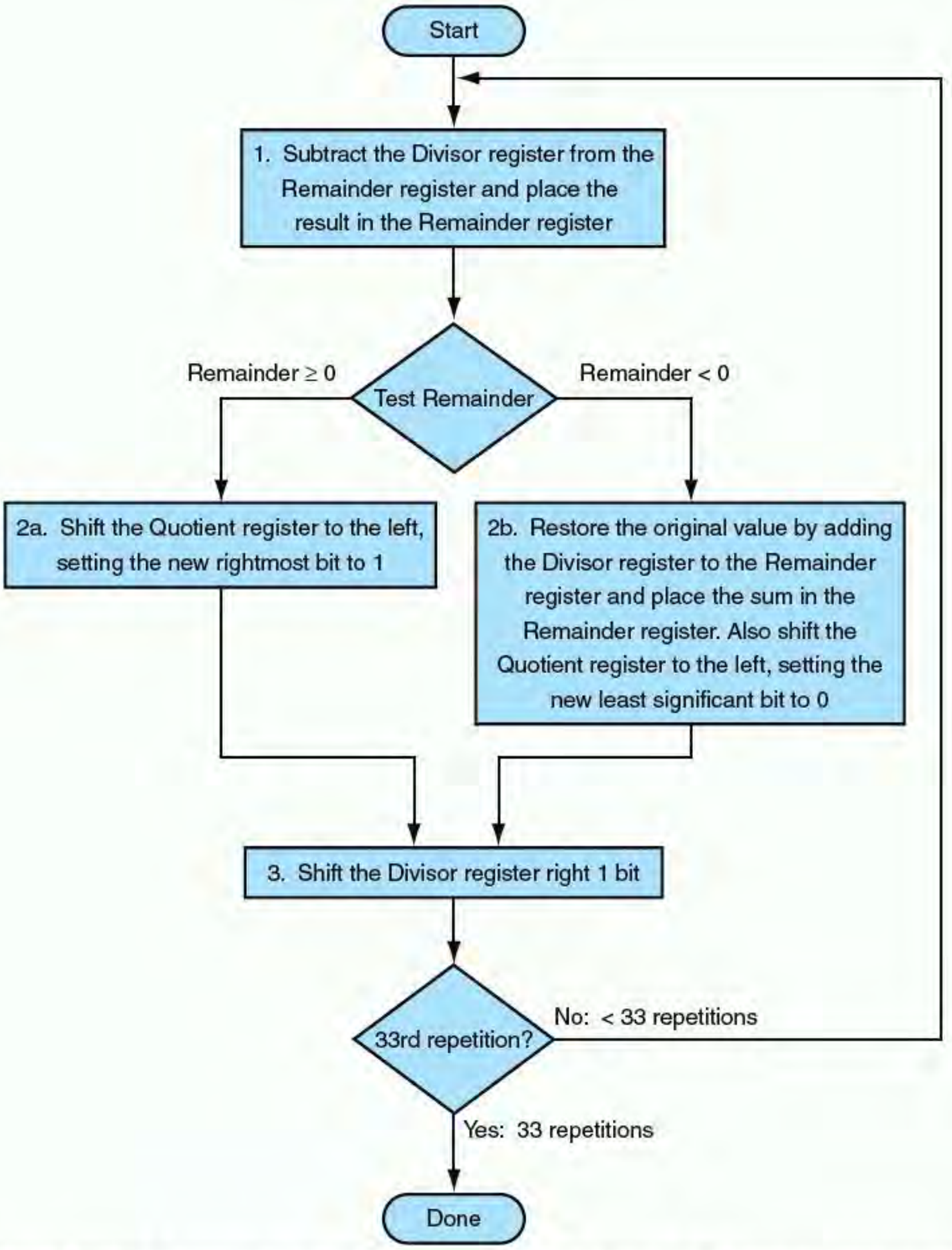


FIGURE 3.11 A division algorithm, using the hardware in Figure 3.10. If the Remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative Remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.

EXAMPLE

A Divide Algorithm

Using a 4-bit version of the algorithm to save pages, let's try dividing 7_{ten} by 2_{ten}, or $0000\ 0111_{two}$ by 0010_{two}.

ANSWER

Figure 3.12 shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

This algorithm and hardware can be refined to be faster and cheaper. The speedup comes from shifting the operands and quotient at the same time as the subtract. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. Figure 3.13 shows the revised hardware.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.12 Division example using the algorithm in Figure 3.11. The bit examined to determine the next step is circled in color.

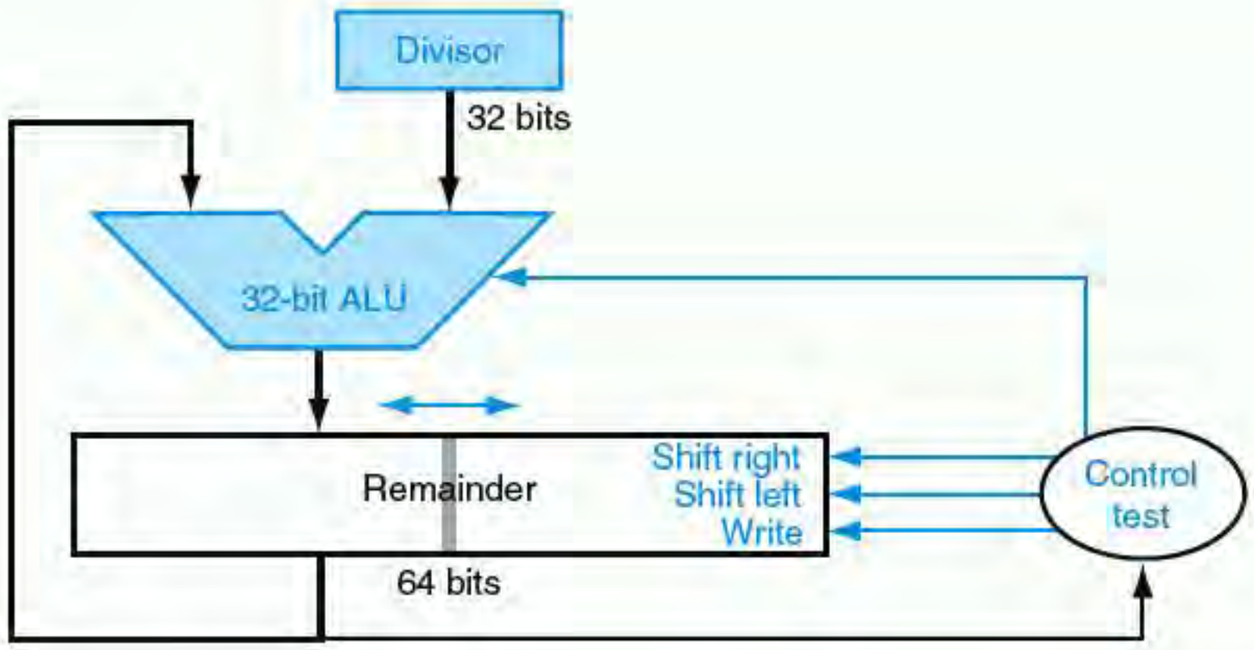


FIGURE 3.13 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.10, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register.

Signed Division

So far we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration: The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$. The first case is easy:

$+7 \div +2$: Quotient = +3, Remainder = +1

Checking the results:

$$7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$-7 \div +2$: Quotient = –3

Rewriting our basic formula to calculate the remainder:

$$\text{Remainder} = (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3 \times +2) = -7 - (-6) = -1$$

So,

$-7 \div +2$: Quotient = –3, Remainder = –1

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of –4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly if

$$-(x \div y) \neq (-x) \div y$$

programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

We calculate the other combinations by following the same rule:

+7 ÷ –2: Quotient = –3, Remainder = +1
–7 ÷ –2: Quotient = +3, Remainder = –1

Thus the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster Division

We used 32 adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per bit. The *SRT division* technique tries to guess several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong guesses. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from divisor to index a table that determines the guess for each step.

The accuracy of this fast method depends on having proper values in the lookup table. The fallacy on page 222 in Section 3.8 shows what can happen if the table is incorrect.

Divide in MIPS

You may have already observed that the same sequential hardware can be used for both multiply and divide in Figures 3.7 and 3.13. The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts. Hence, MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide. As we might expect from the algorithm above, Hi contains the remainder, and Lo contains the quotient after the divide instruction completes.

To handle both signed integers and unsigned integers, MIPS has two instructions: *divide* (`div`) and *divide unsigned* (`divu`). The MIPS assembler allows divide instructions to specify three registers, generating the `mflo` or `mfhi` instructions to place the desired result into a general-purpose register.

Summary

The common hardware support for multiply and divide allows MIPS to provide a single pair of 32-bit registers that are used both for multiply and divide. Figure 3.14 summarizes the additions to the MIPS architecture for the last two sections.

Hardware Software Interface
MIPS divide instructions ignore overflow, so software must determine if the quotient is too large. In addition to overflow, division can also result in an improper calculation: division by 0. Some computers distinguish these two anomalous events. MIPS software must check the divisor to discover division by 0 as well as overflow.

Elaboration: An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply *adds* the dividend to the shifted remainder in the following step since $(r + d) \times 2 - d = r \times 2 + d \times 2 - d = r \times 2 + d$. This *nonrestoring* division algorithm, which takes 1 clock per step, is explored further in Exercise 3.29; the algorithm here is called *restoring* division.

3.6 Floating Point

Speed gets you nowhere if you're headed the wrong way.

American proverb

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\ldots_{ten}$ (π)

$2.71828\ldots_{ten}$ (e)

0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called **scientific notation**, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a **normalized** number, which is the usual way to write it. For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

normalized A number in floating-point notation that has no leading 0s.

MIPS assembly language

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo
Data transfer	load word	lw	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw	$s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load half unsigned	lhu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	sh	$s1,100($s2)	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte unsigned	lbu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb	$s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui	$s1,100	$s1 = 100 * 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	$s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND with constant
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR 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; two's complement
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
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 3.14 MIPS architecture revealed thus far. The memory and registers of the MIPS architecture are not included for space reasons, but this section added the hi and lo registers to support multiply and divide. Color indicates the portions revealed since Figure 3.4 on page 175. MIPS machine language is listed in the MIPS summary reference card at the front of this book.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{two} \times 2^{-1}$$

To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; *binary point* will do fine.

Computer arithmetic that supports such numbers is called **floating point** because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

$$1.xxxxxxxxx_{two} \times 2^{yyyy}$$

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we show the exponent in decimal.)

A standard scientific notation for reals in normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

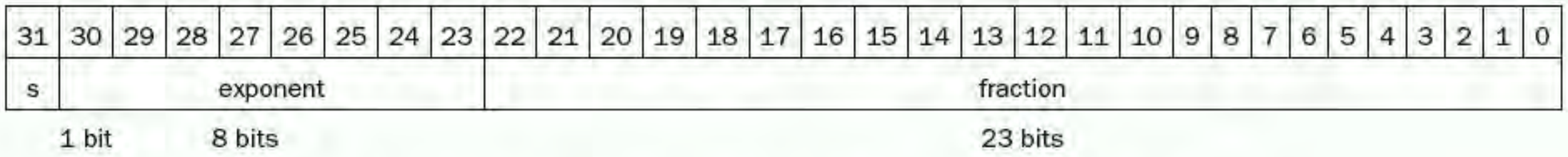
Floating-Point Representation

A designer of a floating-point representation must find a compromise between the size of the **fraction** and the size of the **exponent** because a fixed word size means you must take a bit from one to add a bit to the other. This trade-off is between precision and range: Increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from Chapter 2 reminds us, good design demands good compromise.

fraction The value, generally between 0 and 1, placed in the fraction field.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number. This representation is called *sign and magnitude*, since the sign has a separate bit from the rest of the number.



In general, floating-point numbers are generally of the form

$$(-1)^S \times F \times 2^E$$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that MIPS does something slightly more sophisticated.)

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{ten} \times 10^{-38}$ and numbers almost as large as $2.0_{ten} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that **overflow** here means that the exponent is too large to be represented in the exponent field.

overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, people call this event **underflow**. This situation occurs when the negative exponent is too large to fit in the exponent field.

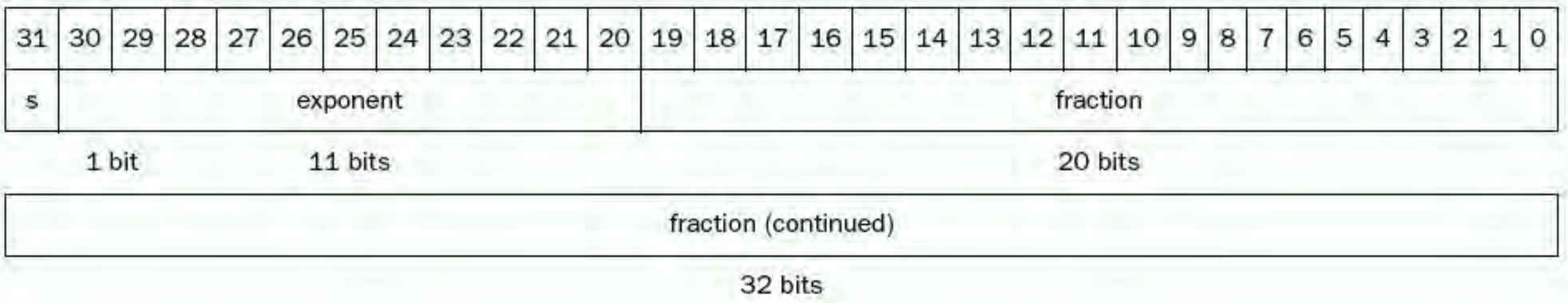
underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

One way to reduce chances of underflow or overflow is to offer another format that has a larger exponent. In C this number is called *double*, and operations on doubles are called **double precision** floating-point arithmetic; **single precision** floating point is the name of the earlier format.

double precision A floating-point value represented in two 32-bit words.

single precision A floating-point value represented in a single 32-bit word.

The representation of a double precision floating-point number takes two MIPS words, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *fraction* is the 52-bit number in the fraction.



MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the larger significand.

These formats go beyond MIPS. They are part of the *IEEE 754 floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

To pack even more bits into the significand, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or 53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus $00\ldots00_{two}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s1, s2, s3, . . . , then the value is

$$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$$

Figure 3.15 shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\infty$ or $-\infty$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

IEEE 754 even has a symbol for the result of invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is *NaN*, for *Not a Number*. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when it is convenient.

The designers of IEEE 754 also wanted a floating-point representation that could be easily processed by integer comparisons, especially for sorting. This desire is why the sign is in the most significant bit, allowing a quick test of less than, greater than, or equal to 0. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± denormalized number
1–254	anything	1–2046	anything	± floating-point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

FIGURE 3.15 IEEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the elaboration on page 217.

Placing the exponent before the significand also simplifies sorting of floating-point numbers using integer comparison instructions, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{two} \times 2^{-1}$ would be represented as

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	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The desirable notation must therefore represent the most negative exponent as $00\ldots00_{two}$ and the most positive as $11\ldots11_{two}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so –1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ 1110_{two}$, and +1 is represented by $1 + 127$, or $128_{ten} = 1000\ 0000_{two}$. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The exponent bias for double precision is 1023.

Thus IEEE 754 notation can be processed by integer compares to accelerate sorting of floating-point numbers. Let's show the representation.

Floating-Point Representation

EXAMPLE

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

ANSWER

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2{}_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2{}_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

When we subtract the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$, the result is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126 - 127)}$$

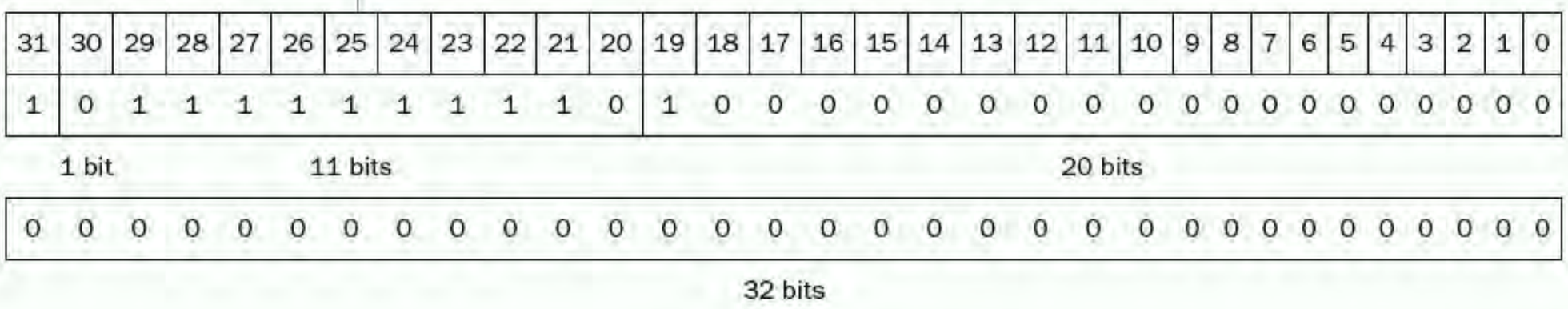
The single precision binary representation of -0.75_{ten} is then

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
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\text{two}) \times 2^{(1022-1023)}$$

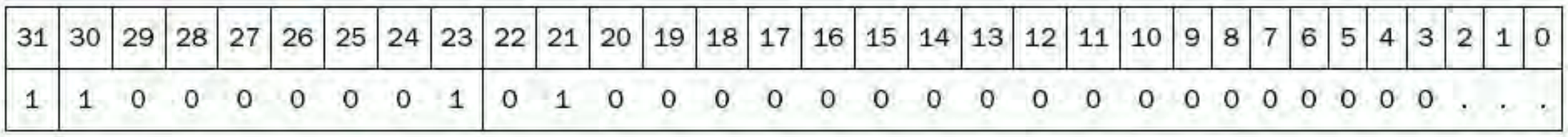


Now let's try going the other direction.

Converting Binary to Decimal Floating Point

EXAMPLE

What decimal number is represented by this single precision float?



ANSWER

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$

In the next sections we will give the algorithms for floating-point addition and multiplication. At their core, they use the corresponding integer operations on the significands, but extra bookkeeping is necessary to handle the exponents and normalize the result. We first give an intuitive derivation of the algorithms in decimal, and then give a more detailed, binary version in the figures.

Elaboration: In an attempt to increase range without removing bits from the significand, some computers before the IEEE 754 standard used a base other than 2. For example, the IBM 360 and 370 mainframe computers use base 16. Since changing the IBM exponent by one means shifting the significand by 4 bits, "normalized" base 16 numbers can have up to 3 leading bits of 0s! Hence, hexadecimal digits mean that up to 3 bits must be dropped from the significand, which leads to surprising problems in the accuracy of floating-point arithmetic, as noted in Section .

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1. To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{ten} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{ten} \times 10^{1}$. Thus the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really:

$$0.016_{ten} \times 10^{1}$$

Step 2. Next comes the addition of the significands:

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

The sum is $10.015_{ten} \times 10^{1}$.

Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}$$

Thus, after the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows

shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4. Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{ten} \times 10^2$$

is rounded to four digits in the significand to

$$1.002_{ten} \times 10^2$$

since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

Figure 3.16 shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all zero bits in the exponent is reserved and used for the floating-point representation of zero. Also, the pattern of all one bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the Elaboration on page 217). Thus, for single precision, the maximum exponent is 127, and the minimum exponent is –126. The limits for double precision are 1023 and –1022.

EXAMPLE

Decimal Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in Figure 3.16.

ANSWER

Let's first look at the binary version of the two numbers in normalized scientific notation, assuming that we keep 4 bits of precision:

$$\begin{aligned} 0.5_{ten} &= 1/2_{ten} &&= 1/2^{1}_{ten} \\ &= 0.1_{two} &&= 0.1_{two} \times 2^{0} &&= 1.000_{two} \times 2^{-1} \\ -0.4375_{ten} &= -7/16_{ten} &&= -7/2^{4}_{ten} \\ &= -0.0111_{two} &&= -0.0111_{two} \times 2^{0} &&= -1.110_{two} \times 2^{-2} \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{two} \times 2^{-1} &= 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} \\ &= 1.000_{two} \times 2^{-4} \end{aligned}$$

Since $127 \geq -4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{two} \times 2^{-4} &= 0.0001000_{two} &&= 0.0001_{two} \\ &= 1/2^{4}_{ten} &&= 1/16_{ten} &&= 0.0625_{ten} \end{aligned}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Many computers dedicate hardware to run floating-point operations as fast as possible. Figure 3.17 sketches the basic organization of hardware for floating-point addition.

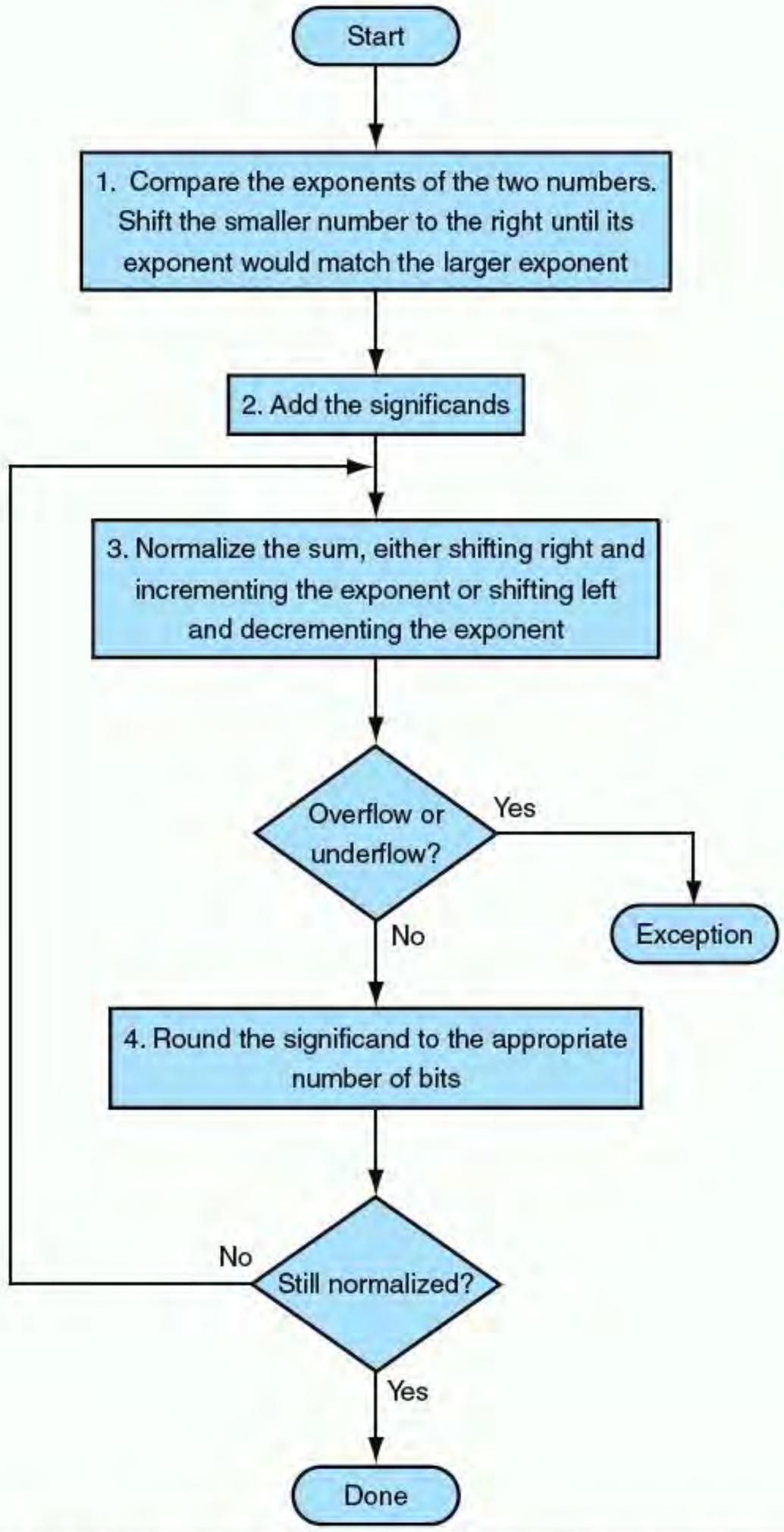


FIGURE 3.16 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

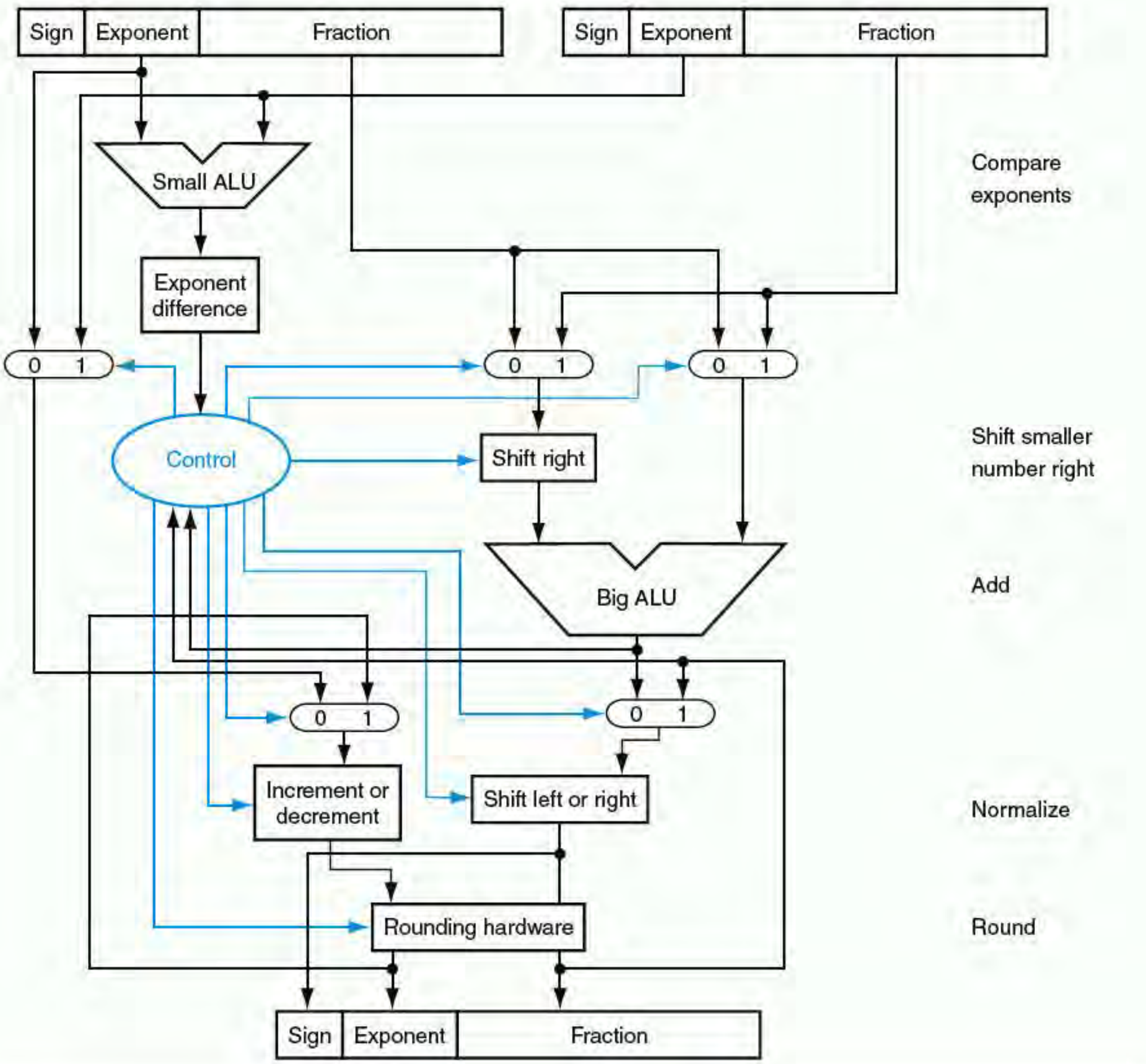


FIGURE 3.17 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.16 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Floating-Point Multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1. Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\text{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: 10 + 127 = 137, and –5 + 127 = 122, so

$$\text{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\text{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\text{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

Step 2. Next comes the multiplication of the significands:

$$\begin{array}{r} 1.110_{ten} \\ \times\ 9.200_{ten} \\ \hline 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{ten} \end{array}$$

There are three digits to the right of the decimal for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{ten}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212×10^5.

Step 3. This product is unnormalized, so we need to normalize it:

$$10.212_{ten} \times 10^5 = 1.0212_{ten} \times 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4. We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$1.0212_{ten} \times 10^6$$

is rounded to four digits in the significand to

$$1.021_{ten} \times 10^6$$

Step 5. The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise it's negative. Hence the product is

$$+1.021_{ten} \times 10^6$$

The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication the sign of the product is determined by the signs of the operands.

Once again, as Figure 3.18 shows, multiplication of binary floating-point numbers is quite similar to the steps we have just completed. We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result. Next is multiplication of significands, followed by an optional normalization step. The size of the exponent is checked for overflow or underflow, and then the product is rounded. If rounding leads to further normalization, we once again check for exponent size. Finally, set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

Decimal Floating-Point Multiplication

EXAMPLE

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in Figure 3.18.

ANSWER

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence the product is

$$-1.110_{two} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{two} \times 2^{-3} &= -0.001110_{two} = -0.00111_{two} \\ &= -7/2^{5}{}_{ten} = -7/32_{ten} = -0.21875_{ten} \end{aligned}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

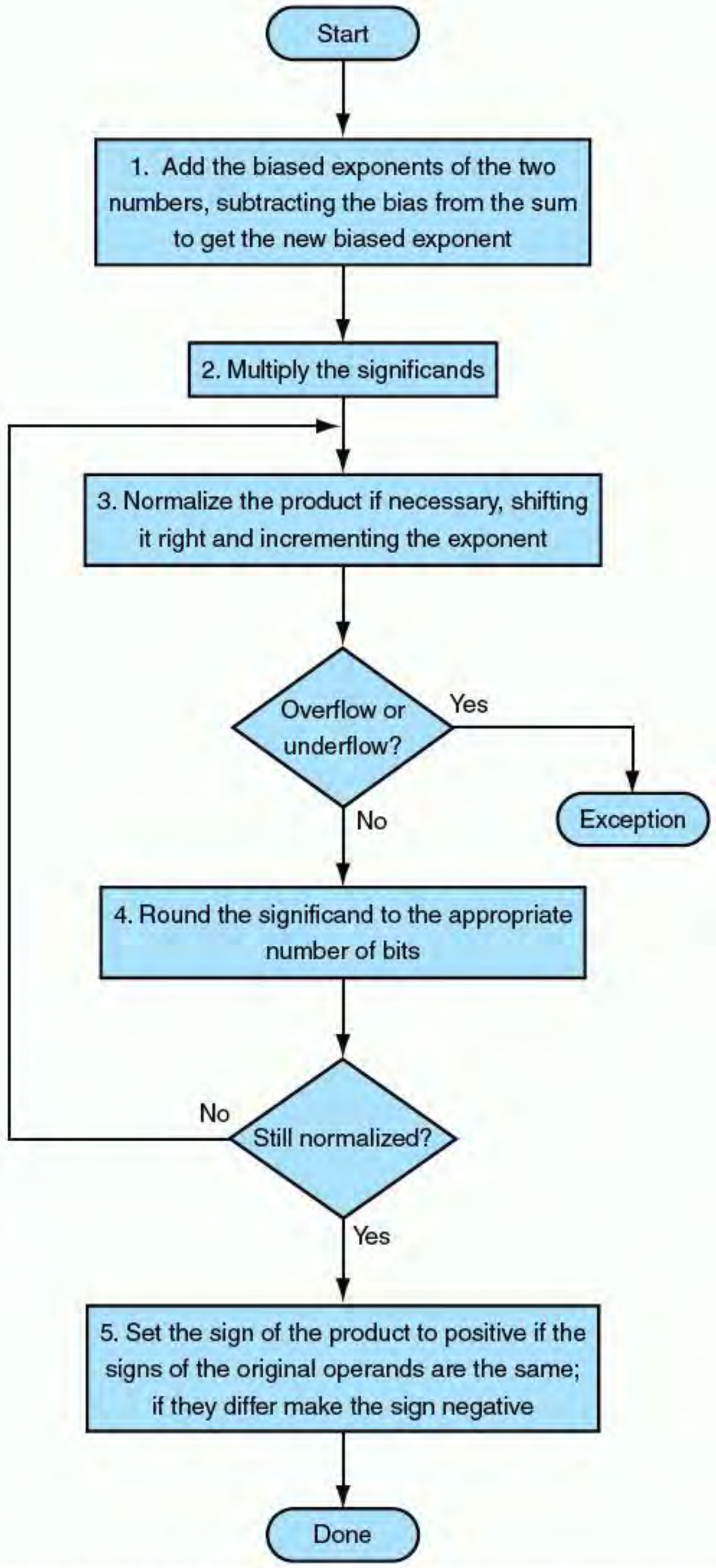


FIGURE 3.18 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

(page 205)

Floating-Point Instructions in MIPS

MIPS supports the IEEE 754 single precision and double precision formats with these instructions:

- Floating-point *addition, single* (`add.s`) and *addition, double* (`add.d`)
- Floating-point *subtraction, single* (`sub.s`) and *subtraction, double* (`sub.d`)
- Floating-point *multiplication, single* (`mul.s`) and *multiplication, double* (`mul.d`)
- Floating-point *division, single* (`div.s`) and *division, double* (`div.d`)
- Floating-point *comparison, single* (`c.x.s`) and *comparison, double* (`c.x.d`), where `x` may be *equal* (`eq`), *not equal* (`neq`), *less than* (`lt`), *less than or equal* (`le`), *greater than* (`gt`), or *greater than or equal* (`ge`)
- Floating-point *branch, true* (`bc1t`) and *branch, false* (`bc1f`)

Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.

The MIPS designers decided to add separate floating-point registers—called `$f0`, `$f1`, `$f2`, ...—used either for single precision or double precision. Hence, they included separate loads and stores for floating-point registers: `lwc1` and `swc1`. The base registers for floating-point data transfers remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
lwc1    $f4,x($sp)  # Load 32-bit F.P. number into F4
lwc1    $f6,y($sp)  # Load 32-bit F.P. number into F6
add.s   $f2,$f4,$f6 # F2 = F4 + F6 single precision
swc1    $f2,z($sp)  # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name.

Figure 3.19 summarizes the floating-point portion of the MIPS architecture revealed in this chapter, with the additions to support floating point shown in color. Similar to Figure 2.25 on page 103 in Chapter 2, we show the encoding of these instructions in Figure 3.20.

MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	`$f0, $f1, $f2, . . . , $f31`	MIPS floating-point registers are used in pairs for double precision numbers.
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, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS floating-point assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	`Memory[$s2 + 100] = $f1`	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
add.s	R	17	16	6	4	2	0	add.s $f2,$f4,$f6
sub.s	R	17	16	6	4	2	1	sub.s $f2,$f4,$f6
mul.s	R	17	16	6	4	2	2	mul.s $f2,$f4,$f6
div.s	R	17	16	6	4	2	3	div.s $f2,$f4,$f6
add.d	R	17	17	6	4	2	0	add.d $f2,$f4,$f6
sub.d	R	17	17	6	4	2	1	sub.d $f2,$f4,$f6
mul.d	R	17	17	6	4	2	2	mul.d $f2,$f4,$f6
div.d	R	17	17	6	4	2	3	div.d $f2,$f4,$f6
lwc1	I	49	20	2	100			lwc1 $f2,100($s4)
swc1	I	57	20	2	100			swc1 $f2,100($s4)
bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

FIGURE 3.19 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, on page A-49, for more detail.

op(31:26):								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	andi	ori	xori	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FIPt), (rt(16:16) = 0 => *c* = f, rt(16:16) = 1 => *c* = t), rs(25:21):								
23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								

op(31:26) = 010001 (FIPt), (*f* above: 10000 => *f* = s, 10001 => *f* = d), funct(5:0):								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

FIGURE 3.20 MIPS floating-point instruction encoding. This notation gives the value of a field by row and by column. For example, in the top portion of the figure lw is found in row number 4 (100_{two} for bits 31–29 of the instruction) and column number 3 (011_{two} for bits 28–26 of the instruction), so the corresponding value of the op field (bits 31–26) is 100011_{two}. Underscore means the field is used elsewhere. For example, FlPt in row 2 and column 1 (op = 010001_{two}) is defined in the bottom part of the figure. Hence sub.f in row 0 and column 1 of the bottom section means that the funct field (bits 5–0) of the instruction) is 000001_{two} and the op field (bits 31–26) is 010001_{two}. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (f = s so rs = 10000) or double precision (f = d so rs = 10001). Similarly, bit 16 of the instruction determines if the bc1.c instruction tests for true (bit 16 = 1 =>bc1.t) or false (bit 16 = 0 =>bc1.f). Instructions in color are described in Chapters 2 or 3, with Appendix A covering all instructions.

Hardware Software Interface

One issue that computer designers face in supporting floating-point arithmetic is whether to use the same registers used by the integer instructions or to add a special set for floating point. Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program. The major impact is to create a separate set of data transfer instructions to move data between floating-point registers and memory.

The benefits of separate floating-point registers are having twice as many registers without using up more bits in the instruction format, having twice the register bandwidth by having separate integer and floating-point register sets, and being able to customize registers to floating point; for example, some computers convert all sized operands in registers into a single internal format

EXAMPLE

Compiling a Floating-Point C Program into MIPS Assembly Code

Let's convert a temperature in Fahrenheit to Celsius:

```
float f2c (float fahr)
        {
                return ((5.0/9.0) * (fahr - 32.0));
        }
```

Assume that the floating-point argument `fahr` is passed in `$f12` and the result should go in `$f0`. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

ANSWER

We assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer `$gp`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
        lwc1 $f16,const5($gp) # $f16 = 5.0 (5.0 in memory)
        lwc1 $f18,const9($gp) # $f18 = 9.0 (9.0 in memory)
```

They are then divided to get the fraction 5.0/9.0:

```
div.s $f16, $f16, $f18 # $f16 = 5.0 / 9.0
```

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next we load the constant 32.0 and then subtract it from `fahr` (`$f12`):

```
lwc1 $f18, const32($gp) # $f18 = 32.0
sub.s $f18, $f12, $f18 # $f18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in `$f0` as the return result, and then return:

```
mul.s$f0,  $f16, $f18 # $f0 = (5/9)*(fahr - 32.0)
jr  $ra # return
```

Now let's perform floating-point operations on matrices, code commonly found in scientific programs.

EXAMPLE

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of X = X + Y * Z. Let's assume X, Y, and Z are all square matrices with 32 elements in each dimension.

```
void mm (double x[][], double y[][], double z[][])
{
        int i, j, k;

        for (i = 0; i! = 32; i = i + 1)
        for (j = 0; j! = 32; j = j + 1)
        for (k = 0; k! = 32; k = k + 1)
          x[i][j] = x[i][j] + y[i][k] * z[k][j];
}
```

The array starting addresses are parameters, so they are in `$a0`, `$a1`, and `$a2`. Assume that the integer variables are in `$s0`, `$s1`, and `$s2`, respectively. What is the MIPS assembly code for the body of the procedure?

ANSWER

Note that `x[i][j]` is used in the innermost loop above. Since the loop index is `k`, the index does not affect `x[i][j]`, so we can avoid loading and storing `x[i][j]` each iteration. Instead, the compiler loads `x[i][j]` into a register outside the loop, accumulates the sum of the products of `y[i][k]` and `z[k][j]` in that same register, and then stores the sum into `x[i][j]` upon termination of the innermost loop.

We keep the code simpler by using the assembly language pseudoinstructions `li` (which loads a constant into a register), and `l.d` and `s.d` (which the assembler turns into a pair of data transfer instructions, `lwc1` or `swc1`, to a pair of floating-point registers).

The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables:

```
mm:...
        li    $t1, 32  # $t1 = 32 (row size/loop end)
        li    $s0, 0   # i = 0; initialize 1st for loop
L1:     li    $s1, 0   # j = 0; restart 2nd for loop
L2:     li    $s2, 0   # k = 0; restart 3rd for loop
```

To calculate the address of `x[i][j]`, we need to know how a 32 × 32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the `i` "single-dimensional arrays," or rows, to get the one we want. Thus we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
sll  $t2, $s0, 5 # $t2 = i * 2^5 (size of row of x)
```

Now we add the second index to select the `j`th element of the desired row:

```
addu  $t2, $t2, $s1 # $t2 = i * size(row) + j
```

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3:

```
sll  $t2, $t2, 3 # $t2 = byte offset of [i][j]
```

Next we add this sum to the base address of `x`, giving the address of `x[i][j]`, and then load the double precision number `x[i][j]` into `$f4`:

```
addu  $t2, $a0, $t2 # $t2 = byte address of x[i][j]
l.d   $f4, 0($t2) # $f4 = 8 bytes of x[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number `z[k][j]`.

```
L3:   sll $t0, $s2, 5 # $t0 = k * 2^5 (size of row of z)
      addu $t0, $t0, $s1 # $t0 = k * size(row) + j
      sll $t0, $t0, 3 # $t0 = byte offset of [k][j]
      addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
      l.d $f16, 0($t0) # $f16 = 8 bytes of z[k][j]
```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number `y[i][k]`.

```
sll     $t2, $s0, 5 # $t0 = i * 2^5 (size of row of y)
addu    $t0, $t0, $s2 # $t0 = i * size(row) + k
sll     $t0, $t0, 3 # $t0 = byte offset of [i][k]
addu    $t0, $a1, $t0 # $t0 = byte address of y[i][k]
l.d     $f18, 0($t0) # $f18 = 8 bytes of y[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of `y` and `z` located in registers `$f18` and `$f16`, and then accumulate the sum in `$f4`.

```
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16 # f4 = x[i][j]+ y[i][k] * z[k][j]
```

The final block increments the index `k` and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in `$f4` into `x[i][j]`.

```
addiu  $s2, $s2, 1 # $k k + 1
bne    $s2, $t1, L3 # if (k != 32) go to L3
s.d    $f4, 0($t2) # x[i][j] = $f4
```

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```
addiu  $s1, $s1, 1 # $j = j + 1
bne    $s1, $t1, L2 # if (j != 32) go to L2
addiu  $s0, $s0, 1 # $i = i + 1
bne    $s0, $t1, L1 # if (i != 32) go to L1
...
```

Elaboration: The array layout discussed in the example, called *row major order,* is used by C and many other programming languages. Fortran instead uses *column major order,* whereby the array is stored column by column.

Only 16 of the 32 MIPS floating-point registers could originally be used for single precision operations: `$f0, $f2, $f4, ..., $f30`. Double precision is computed using pairs

of these registers. The odd-numbered floating-point registers were used only to load and store the right half of 64-bit floating-point numbers. MIPS-32 added `l.d` and `s.d` to the instruction set. MIPS-32 also added "paired single" versions of all floating-point instructions, where a single instruction results in two parallel floating-point operations on two 32-bit operands inside 64-bit registers. For example, `add.ps F0, F2, F4` is equivalent to `add.s F0, F2, F4` followed by `add.s F1, F3, F5`.

Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn't have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence the floating-point unit, including the floating-point registers, were optionally available as a second chip. Such optional accelerator chips are called *coprocessors*, and explain the acronym for floating-point loads in MIPS: `lwc1` means load word to coprocessor 1, the floating-point unit. (Coprocessor 0 deals with virtual memory, described in Chapter 7.) Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and hence the term "coprocessor" joins "accumulator" and "core memory" as quaint terms that date the speaker.

Elaboration: Although there are many ways to throw hardware at floating-point multiply to make it go fast, floating-point division is considerably more challenging to make fast and accurate. Slow divides in early computers led to removal of divides from many algorithms, but parallel computers have inspired rediscovery of divide-intensive algorithms that work better on these computers. Hence, we may need faster divides.

One technique to leverage a fast multiplier is *Newton's iteration*, where division is recast as finding the zero of a function to find the reciprocal $1/x$, which is then multiplied by the other operand. Iteration techniques *cannot* be rounded properly without calculating many extra bits. A TI chip solves this problem by calculating an extra-precise reciprocal.

Elaboration: Java embraces IEEE 754 by name in its definition of Java floating-point data types and operations. Thus, the code in the first example could have well been generated for a class method that converted Fahrenheit to Celsius.

The second example uses multiple dimensional arrays, which are not explicitly supported in Java. Java allows arrays of arrays, but each array may have its own length, unlike multiple dimensional arrays in C. Like the examples in Chapter 2, a Java version of this second example would require a good deal of checking code for array bounds, including a new length calculation at the end of row. It would also need to check that the object reference is not null.

Accurate Arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent. The reason is that an infinite variety of real numbers exists between, say, 0 and 1, but no more than 2^{53} can be represented

exactly in double precision floating point. The best we can do is get the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

Rounding sounds simple enough, but to round accurately requires the hardware to include extra bits in the calculation. In the preceding examples, we were vague on the number of bits that an intermediate representation can occupy, but clearly if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps 2 extra bits on the right during intermediate additions, called **guard** and **round**, respectively. Let's do a decimal example to illustrate the value of these extra digits.

guard The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

EXAMPLE

Rounding with Guard Digits

Add $2.56_{ten} \times 10^0$ to $2.34_{ten} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

ANSWER

First we must shift the smaller number to the right to align the exponents, so $2.56_{ten} \times 10^0$ becomes $0.0256_{ten} \times 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{rl} & 2.3400_{ten} \\ + & 0.0256_{ten} \\ \hline & 2.3656_{ten} \end{array}$$

Thus the sum is $2.3656_{ten} \times 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tie-breaker. Rounding the sum up with three significant digits yields $2.37_{ten} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{rl} & 2.34_{ten} \\ + & 0.02_{ten} \\ \hline & 2.36_{ten} \end{array}$$

The answer is $2.36_{ten} \times 10^2$, off by 1 in the last digit from the sum above.

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of **units in the last place**, or **ulp**. If a number was off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there is no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

units in the last place (ulp) The number of bits in error in the least significant bits of the significand between the actual number and the number that can be prepresented.

Elaboration: Although the example above really needed just one extra digit, multiply can need two. A binary product may have one leading 0 bit; hence, the normalizing step must shift the product 1 bit left. This shifts the guard digit into the least significant bit of the product, leaving the round bit to help accurately round the product.

There are four rounding modes: always round up (toward $+\infty$), always round down (toward $-\infty$), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The Internal Revenue Service always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one; if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

The goal of the extra rounding bits is to allow the computer to get the same results as if the intermediate results were calculated to infinite precision and then rounded.

To support this goal and rounding to the nearest even, the standard has a third bit in addition to guard and round; it is set whenever there are nonzero bits to the right of the round bit. This sticky bit allows the computer to see the difference between $0.50 \ldots 00_{ten}$ and $0.50 \ldots 01_{ten}$ when rounding.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{ten} \times 10^{-1}$ to $2.34_{ten} \times 10^{2}$ in the example above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000...00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000...00, we round instead to 2.35.

Summary

The Big Picture below reinforces the stored-program concept from Chapter 2; the meaning of the information cannot be determined just by looking at the bits, for the same bits can represent a variety of objects. This section shows that computer arithmetic is finite and thus can disagree with natural arithmetic. For example, the IEEE 754 standard floating-point representation

$$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{bias})}$$

is almost always an approximation of the real number. Computer systems must take care to minimize this gap between computer arithmetic and arithmetic in the real world, and programmers at times need to be aware of the implications of this approximation.

C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, and, andi, or, ori, nor, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
char	—	lb, sb, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, and, andi, or, ori, nor, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

The BIG Picture

Bit patterns have no inherent meaning. They may represent signed integers, unsigned integers, floating-point numbers, instructions, and so on. What is represented depends on the instruction that operates on the bits in the word.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size, hence limited precision; it's possible to calculate a number too big or too small to be represented in a word. Programmers must remember these limits and write programs accordingly.

Hardware Software Interface

In the last chapter we presented the storage classes of the programming language C (see the Hardware Software Interface section on page 142). The table above shows some of the C and Java data types together with the MIPS data transfer instructions and instructions that operate on those types that appear in Chapters 2 and 3. Note that Java omits unsigned integers.

Check Yourself

Suppose there was a 16-bit IEEE 754 floating-point format with 5 exponent bits. What would be the likely range of numbers it could represent?

1. $1.0000\ 0000\ 00 \times 2^{0}$ to $1.1111\ 1111\ 11 \times 2^{31}, 0$
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \times 2^{15}, \pm 0, \pm \infty$, NaN
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \times 2^{15}, \pm 0, \pm \infty$, NaN
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ to $\pm 1.1111\ 1111\ 11 \times 2^{14}, \pm 0, \pm \infty$, NaN

Elaboration: To accommodate comparisons that may include NaNs, the standard includes *ordered* and *unordered* as options for compares. Hence the full MIPS instruction set has many flavors of compares to support NaNs. (Java does not support unordered compares.)

In an attempt to squeeze every last bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than having a gap between 0 and the smallest normalized number, IEEE allows *denormalized numbers* (also known as *denorms* or *subnormals*). They have the same exponent as zero but a nonzero significand. They allow a number to degrade in significance until it becomes 0, called *gradual underflow.* For example, the smallest positive single precision normalized number is

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{two} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{two} \times 2^{-126}, \text{ or } 1.0_{two} \times 2^{-149}$$

For double precision, the denorm gap goes from 1.0×2^{-1022} to 1.0×2^{-1074}.

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Also, if programmers do not expect denorms, their programs may be surprised.

3.7 Real Stuff: Floating Point in the IA-32

The IA-32 has regular multiply and divide instructions that operate entirely on registers, unlike the reliance on Hi and Lo in MIPS. (In fact, later versions of the MIPS instruction set have added similar instructions.)

The main differences are found in floating-point instructions. The IA-32 floating-point architecture is different from all other computers in the world.

The IA-32 Floating-Point Architecture

The Intel 8087 floating-point coprocessor was announced in 1980. This architecture extended the 8086 with about 60 floating-point instructions.

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the two top elements of the stacks, and stores can pop elements off the stack. Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers on-chip below the top of the stack. Thus, a complete stack instruction set is supplemented by a limited set of register-memory instructions.

This hybrid is still a restricted register-memory model, however, since loads always move data to the top of the stack while incrementing the top-of-stack pointer and stores can only move the top of stack to memory. Intel uses the notation `ST` to indicate the top of stack, and `ST(i)` to represent the *i*th register below the top of stack.

Another novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Unlike the maximum of 64 bits on MIPS, the IA-32 floating-point operands on the stack are 80 bits wide. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. This *double extended precision* is not supported by programming languages, although it has been useful to programmers of mathematical software.

Memory data can be 32-bit (single precision) or 64-bit (double precision) floating-point numbers. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to floating point, and vice versa, for integer loads and stores.

The IA-32 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value
3. Comparison, including instructions to send the result to the integer processor so that it can branch
4. Transcendental instructions, including sine, cosine, log, and exponentiation

Figure 3.21 shows some of the 60 floating-point operations. Note that we get even more combinations when including the operand modes for these operations. Figure 3.22 shows the many options for floating-point add.

Data transfer	Arithmetic	Compare	Transcendental
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

FIGURE 3.21 The floating-point instructions of the IA-32. We use the curly brackets {} to show optional variations of the basic operations: {I} means there is an integer version of the instruction, {P} means this variation will pop one operand off the stack after the operation, and {R} means reverse the order of the operands in this operation. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations in the first column push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the FSTSW instruction, either into the AX register or into memory, followed by an SAHF instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations. Not all combinations suggested by the notation are provided. Hence, F{I}SUB{R}{P} operations represents these instructions found in the IA-32: FSUB, FISUB, FSUBR, FISUBR, FSUBP, FSUBRP. For the integer subtract instructions, there is no pop (FISUBP) or reverse pop (FISUBRP).

Instruction	Operands	Comment
FADD		Both operands in stack; result replaces top of stack.
FADD	ST(i)	One source operand is *i*th register below the top of stack; result replaces the top of stack.
FADD	ST(i), ST	One source operand is the top of stack; result replaces *i*th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory; result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory; result replaces the top of stack.

FIGURE 3.22 The variations of operands for floating-point add in the IA-32.

The floating-point instructions are encoded using the ESC opcode of the 8086 and the postbyte address specifier (see Figure 2.46 on page 143). The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit floating point or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Floating-point performance of the IA-32 family has traditionally lagged far behind other computers. Whether it is simply a lack of attention by Intel engineers or a flaw with its architecture is hard to know. We can say that many new architectures have been announced since 1980, and none have followed in Intel's footsteps. In addition, Intel created a more traditional floating-point architecture as part of SSE2.

The Intel Streaming SIMD Extension 2 (SSE2) Floating-Point Architecture

Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture, including double precision floating-point registers and operations. It includes eight registers that can be used for floating-point operands, giving the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE2 registers as floating-point registers like those found in other computers. AMD expanded the number to 16 as part of AMD64, which Intel relabled EM64T for its use.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles or two doubles. This new architecture can more than double performance over the stack architecture.

Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.

Bertrand Russell, *Recent Words on the Principles of Mathematics*, 1901

3.8 Fallacies and Pitfalls

Arithmetic fallacies and pitfalls generally stem from the difference between the limited precision of computer arithmetic and the unlimited precision of natural arithmetic.

Fallacy: Floating-point addition is associative; that is, $x + (y + z) = (x + y) + z$.

Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number.

For example, suppose $x = -1.5_{ten} \times 10^{38}$, $y = 1.5_{ten} \times 10^{38}$, and $z = 1.0$, and that these are all single precision numbers. Then

$$\begin{aligned} x + (y + z) &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38} + 1.0) \\ &= -1.5_{ten} \times 10^{38} + (1.5_{ten} \times 10^{38}) = 0.0 \\ (x + y) + z &= (-1.5_{ten} \times 10^{38} + 1.5_{ten} \times 10^{38}) + 1.0 \\ &= (0.0_{ten}) + 1.0 \\ &= 1.0 \end{aligned}$$

Therefore, $x + (y + z) \neq (x + y) + z$.

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{ten} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{ten} \times 10^{38} + 1.0$ is still $1.5_{ten} \times 10^{38}$. That is why the sum of x, y, and z is 0.0 or 1.0, depending on the order of the floating-point additions, and hence floating-point add is *not* associative.

Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.

Recall that a binary number x, where xi means the ith bit, represents the number

$$\ldots + (x3 \times 2^3) + (x2 \times 2^2) + (x1 \times 2^1) + (x0 \times 2^0)$$

Shifting the bits of x right by n bits would seem to be the same as dividing by 2^n. And this *is* true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide -5_{ten} by 4_{ten}; the quotient should be -1_{ten}. The two's complement representation of -5_{ten} is

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{two}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $1{,}073{,}741{,}822_{ten}$ instead of -1_{ten}.

A solution would be to have an arithmetic right shift (see **In More Depth:** Booth's Algorithm) that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

The result is -2_{ten} instead of -1_{ten}; close, but no cigar.

The PowerPC, however, does have a fast shift instruction (*shift right algebraic*) that in conjunction with a special add (add with carry) gives the same answer as dividing by a power of 2.

Pitfall: The MIPS instruction add immediate unsigned `addiu` *sign-extends its 16-bit immediate field.*

Despite its name, add immediate unsigned (`addiu`) is used to add constants to signed integers when we don't care about overflow. MIPS has no subtract immediate instruction and negative numbers need sign extension, so the MIPS architects decided to sign-extend the immediate field.

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see Figure 3.23). The following is the inside story behind the headlines.

The Pentium uses a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing –2, –1, 0, +1, or +2. The guess is multiplied by the divisor and



FIGURE 3.23 A sampling of newspaper and magazine articles from November 1994, including the *New York Times*, *San Jose Mercury News*, *San Francisco Chronicle*, and *Infoworld*. The Pentium floating-point divide bug even made the "Top 10 List" of the *David Letterman Late Show* on television. Intel eventually took a $300 million write-off to replace the buggy chips.

subtracted from the remainder to generate a new remainder. Like nonrestoring division (see Exercise 3.29), if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently there were five elements of the table from the 80486 that Intel thought could never be accessed, and they optimized the PLA to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11 bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

The following is a time line of the Pentium bug morality play:

- *July 1994:* Intel discovers the bug in the Pentium. The actual cost to fix the bug was several hundred thousand dollars. Following normal bug fix procedures, it will take months to make the change, reverify, and put the corrected chip into production. Intel planned to put good chips into production in January 1995, estimating that *3* to 5 million Pentiums would be produced with the bug.
- *September 1994:* A math professor at Lynchburg College in Virginia, Thomas Nicely, discovers the bug. After calling Intel technical support and getting no official reaction, he posts his discovery on the Internet. It quickly gained a following, and some pointed out that even small errors become big when multiplying by big numbers: the fraction of people with a rare disease times the population of Europe, for example, might lead to the wrong estimate of the number of sick people.
- *November 7, 1994: Electronic Engineering Times* puts the story on its front page, which is soon picked up by other newspapers.
- *November 22, 1994:* Intel issues a press release, calling it a "glitch." The Pentium "can make errors in the ninth digit. . . . Even most engineers and financial analysts require accuracy only to the fourth or fifth decimal point. Spreadsheet and word processor users need not worry. . . . There are maybe several dozen people that this would affect. So far, we've only heard from one. . . . [Only] theoretical mathematicians (with Pentium computers purchased before the summer) should be concerned." What irked many was that customers were told to describe their application to Intel, and then *Intel* would decide whether or not their application merited a new Pentium without the divide bug.
- *December 5, 1994:* Intel claims the flaw happens once in 27,000 years for the typical spreadsheet user. Intel assumes a user does 1000 divides per day and multiplies the error rate assuming floating-point numbers are random, which is one in 9 billion, and then gets 9 million days, or 27,000 years.

Things begin to calm down, despite Intel neglecting to explain why a typical customer would access floating-point numbers randomly.

- *December 12, 1994:* IBM Research Division disputes Intel's calculation of the rate of errors (you can access this article by visiting *www.mkp.com/books_catalog/cod/links.htm*). IBM claims that common spreadsheet programs, recalculating for 15 minutes a day, could produce Pentium-related errors as often as once every 24 days. IBM assumes 5000 divides per second, for 15 minutes, yielding 4.2 million divides per day, and does not assume random distribution of numbers, instead calculating the chances as one in 100 million. As a result, IBM immediately stops shipment of all IBM personal computers based on the Pentium. Things heat up again for Intel.
- *December 21, 1994:* Intel releases the following, signed by Intel's president, chief executive officer, chief operating officer, and chairman of the board: "We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer."Analysts estimate that this recall cost Intel $500 million, and Intel employees did not get a Christmas bonus that year.

This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

In April 1997 another floating-point bug was revealed in the Pentium Pro and Pentium II microprocessors. When the floating-point-to-integer store instructions (`fist`, `fistp`) encounter a negative floating-point number that is too large to fit in a 16- or 32-bit word after being converted to integer, they set the wrong bit in the FPO status word (precision exception instead of invalid operation exception). To Intel's credit, this time they publicly acknowledged the bug and offered a software patch to get around it—quite a different reaction from what they did in 1994.

3.9 Concluding Remarks

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called "overflow" or "underflow," may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. Chapter 5 discusses exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation are part of the inspiration for the field of numerical analysis.

Over the years, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic and IEEE 754 binary floating-point arithmetic are found in the vast majority of computers sold today. For example, every desktop computer sold since this book was first printed follows these conventions.

A side effect of the stored-program computer is that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, instruction, and so on. It is the instruction that operates on the word that determines its meaning.

With the explanation of computer arithmetic in this chapter comes a description of much more of the MIPS instruction set. One point of confusion is the instructions covered in these chapters versus instructions executed by MIPS chips versus the instructions accepted by MIPS assemblers. The next two figures try to make this clear.

Figure 3.24 lists the MIPS instructions covered in this chapter and Chapter 2. We call the set of instructions on the left-hand side of the figure the *MIPS core*. The instructions on the right we call the *MIPS arithmetic core*. On the left of Figure 3.25 are the instructions the MIPS processor executes that are not found in Figure 3.24. We call the full set of hardware instructions *MIPS-32*. On the right of Figure 3.25 are the instructions accepted by the assembler that are not part of MIPS-32. We call this set of instructions *Pseudo MIPS*.

Instruction subset	Integer	Fl. pt.
MIPS core	95%	57%
MIPS arithmetic core	0%	41%
Remaining MIPS-32	5%	2%

MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
and	and	R	move from system control (EPC)	mfc0	R
and immediate	andi	I	floating-point add single	add.s	R
or	or	R	floating-point add double	add.d	R
or immediate	ori	I	floating-point subtract single	sub.s	R
nor	nor	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
branch on equal	beq	I	branch on floating-point false	bc1f	I
branch on not equal	bne	I	floating-point compare single	c.x.s	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J	floating-point compare double	c.x.d	R
jump register	jr	R	(x = eq, neq, lt, le, gt, ge)		
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

FIGURE 3.24 The MIPS instruction set covered so far. This book concentrates on the instructions in the left column.

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or ($rs \oplus rt$)	xor	R	move	move	rd,rs
exclusive or immediate	xori	I	absolute value	abs	rd,rs
shift right arithmetic	sra	R	not ($\neg rs$)	not	rd,rs
shift left logical variable	sllv	R	negate *(signed or unsigned)*	negs	rd,rs
shift right logical variable	srlv	R	rotate left	rol	rd,rs,rt
shift right arithmetic variable	srav	R	rotate right	ror	rd,rs,rt
move to Hi	mthi	R	multiply and don't check oflw *(signed or uns.)*	muls	rd,rs,rt
move to Lo	mtlo	R	multiply and check oflw *(signed or uns.)*	mulos	rd,rs,rt
load halfword	lh	I	divide and check overflow	div	rd,rs,rt
load byte	lb	I	divide and don't check overflow	divu	rd,rs,rt
load word left *(unaligned)*	lwl	I	remainder *(signed or unsigned)*	rems	rd,rs,rt
load word right *(unaligned)*	lwr	I	load immediate	li	rd,imm
store word left *(unaligned)*	swl	I	load address	la	rd,addr
store word right *(unaligned)*	swr	I	load double	ld	rd,addr
load linked (*atomic update*)	ll	I	store double	sd	rd,addr
store cond. (*atomic update*)	sc	I	unaligned load word	ulw	rd,addr
move if zero	movz	R			
move if not zero	movn	R	unaligned store word	usw	rd,addr
multiply and add (S or *uns.*)	madds	R			
multiply and subtract (S or *uns.*)	msubs	I	unaligned load halfword *(signed or uns.)*	ulhs	rd,addr
branch on ≥ zero and link	bgezal	I	unaligned store halfword	ush	rd,addr
branch on < zero and link	bltzal	I	branch	b	Label
jump and link register	jalr	R	branch on equal zero	beqz	rs,L
branch compare to zero	b*x*z	I	branch on compare *(signed or unsigned)*	bxs	rs,rt,L
branch compare to zero likely	b*x*zl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			set equal	seq	rd,rs,rt
branch compare reg likely	b*x*l	I	set not equal	sne	rd,rs,rt
trap if compare reg	t*x*	R	set on compare *(signed or unsigned)*	sxs	rd,rs,rt
trap if compare immediate	t*x*i	I	(x = lt, le, gt, ge)		
(x = eq, neq, lt, le, gt, ge)			load to floating point *(s or d)*	l.*f*	rd,addr
return from exception	rfe	R	store from floating point *(s or d)*	s.*f*	rd,addr
system call	syscall	I			
break *(cause exception)*	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move *(s or d)*	mov.*f*	R			
FP move if zero *(s or d)*	movz.*f*	R			
FP move if not zero *(s or d)*	movn.*f*	R			
FP square root *(s or d)*	sqrt.*f*	R			
FP absolute value *(s or d)*	abs.*f*	R			
FP negate *(s or d)*	neg.*f*	R			
FP convert *(w, s, or d)*	cvt.*f*.*f*	R			
FP compare un *(s or d)*	c.xn.*f*	R			

FIGURE 3.25 Remaining MIPS-32 and "Pseudo MIPS" instruction sets. *f* means single (s) or double precision (d) floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.*f*/msub.*f*), ceiling (ceil.*f*), truncate (trunc.*f*), round (round.*f*), and reciprocal (recip.*f*).

(page 227)

Figure 3.26 gives the popularity of the MIPS instructions for SPEC2000 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 1% of the instructions executed. The following table summarizes that information.

Note that although programmers and compiler writers may use MIPS-32 to have a richer menu of options, MIPS core instructions dominate integer SPEC2000 execution, and the integer core plus arithmetic core dominate SPEC2000 floating point.

For the rest of the book, we concentrate on the MIPS core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. As we can see, the MIPS core includes the most popular MIPS instructions, and be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious computers.

Core MIPS	Name	Integer	Fl. pt.	Arithmetic core + MIPS-32	Name	Integer	Fl. pt.
add	add	0%	0%	FP add double	add.d	0%	8%
add immediate	addi	0%	0%	FP subtract double	sub.d	0%	3%
add unsigned	addu	7%	21%	FP multiply double	mul.d	0%	8%
add immediate unsigned	addiu	12%	2%	FP divide double	div.d	0%	0%
subtract unsigned	subu	3%	2%	load word to FP double	l.d	0%	15%
and	and	1%	0%	store word to FP double	s.d	0%	7%
and immediate	andi	3%	0%	shift right arithmetic	sra	1%	0%
or	or	7%	2%	load half	lhu	1%	0%
or immediate	ori	2%	0%	branch less than zero	bltz	1%	0%
nor	nor	3%	1%	branch greater or equal zero	bgez	1%	0%
shift left logical	sll	1%	1%	branch less or equal zero	blez	0%	1%
shift right logical	srl	0%	0%	multiply	mul	0%	1%
load upper immediate	lui	2%	5%				
load word	lw	24%	15%				
store word	sw	9%	2%				
load byte	lbu	1%	0%				
store byte	sb	1%	0%				
branch on equal (zero)	beq	6%	2%				
branch on not equal (zero)	bne	5%	1%				
jump and link	jal	1%	0%				
jump register	jr	1%	0%				
set less than	slt	2%	0%				
set less than immediate	slti	1%	0%				
set less than unsigned	sltu	1%	0%				
set less than imm. uns.	sltiu	1%	0%				

FIGURE 3.26 The frequency of the MIPS instructions for SPEC2000 integer and floating point. All instructions that accounted for at least 1% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here. This data is from Chapter 2 of *Computer Architecture: A Quantitative Approach,* third edition.

3.10 Historical Perspective and Further Reading

Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."

W. Kahan, 1992

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, plus the rationale for the 80-bit stack architecture for floating point in the IA-32. See Section 3.10.

3.11 Exercises

Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.

Winston Churchill, address at Harrow School, 1941, *Abroad*, 1869

3.1 [3] <§3.2> Convert 4096_{ten} into a 32-bit two's complement binary number.

3.2 [3] <§3.2> Convert -2047_{ten} into a 32-bit two's complement binary number.

3.3 [5] <§3.2> Convert $-2{,}000{,}000_{ten}$ into a 32-bit two's complement binary number.

3.4 [5] <§3.2> What decimal number does this two's complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0000\ 0110_{two}$?

3.5 [5] <§3.2> What decimal number does this two's complement binary number represent: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_{two}$?

3.6 [5] <§3.2> What decimal number does this two's complement binary number represent: $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_{two}$?

3.7 [10] <§3.2> Find the shortest sequence of MIPS instructions to determine the absolute value of a two's complement integer. Convert this instruction (accepted by the MIPS assembler):

```
abs   $t2,$t3
```

This instruction means that register `$t2` has a copy of register `$t3` if register `$t3` is positive, and the two's complement of register `$t3` if `$t3` is negative. (Hint: It can be done with three instructions.)

3.8 [10] <§3.2> For More Practice: Number Representations

3.9 [10] <§3.2> If `A` is a 32-bit address, typically an instruction sequence such as

```
lui $t0, A_upper
ori $t0, $t0, A_lower
lw $s0, 0($t0)
```

can be used to load the word at `A` into a register (in this case, `$s0`). Consider the following alternative, which is more efficient:

```
lui $t0, A_upper_adjusted
lw $s0, A_lower($t0)
```

Describe how `A_upper` is adjusted to allow this simpler code to work. (Hint: `A_upper` needs to be adjusted because `A_lower` will be sign-extended.)

3.10 [10] <§3.3> Find the shortest sequence of MIPS instructions to determine if there is a carry out from the addition of two registers, say, registers `$t3` and `$t4`. Place a 0 or 1 in register `$t2` if the carry out is 0 or 1, respectively. (Hint: It can be done in two instructions.)

3.11 [15] <§3.3> For More Practice: Writing MIPS Code to Perform Arithmetic

3.12 [15] <§3.3> Suppose that all of the conditional branch instructions except `beq` and `bne` were removed from the MIPS instruction set along with `slt` and all of its variants (`slti`, `sltu`, `sltui`). Show how to perform

```
slt $t0, $s0, $s1
```

using the modified instruction set in which `slt` is not available. (Hint: It requires more than two instructions.)

3.13 [10] <§3.3> Draw the gates for the Sum bit of an adder, given the equation on page B-28.

3.14 [5] <§3.4> For More Practice: Writing MIPS Code to Perform Arithmetic

3.15 [20] <§3.4> For More Practice: Writing MIPS Code to Perform Arithmetic

3.16 [2 weeks] <§3.4> For More Practice: Simulating MIPS Machines

3.17 [1 week] <§3.4> For More Practice: Simulating MIPS Machines

3.18 [5] <§3.4> For More Practice: Carry Lookahead Adders

3.19 [15] <§3.4> For More Practice: Carry Lookahead Adders

3.20 [10] <§3.4> For More Practice: Relative Performance of Adders

3.21 [15] <§3.4> For More Practice: Relative Performance of Adders

3.22 [15] <§3.4> For More Practice: Relative Performance of Adders

3.23 [30] <§3.4> In More Depth: Booth's Algorithm

3.24 [15][30] <§3.4> For More Practice: Special MIPS Registers

3.25 [10] <§§3.5, 3.4> In More Depth: The Power PC's Multiply-Add Instruction

3.26 [20] <§3.5> In More Depth: The Power PC's Multiply-Add Instruction

3.27 <§§3.3, 3.4, 3.5> With $x = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 1011_{two}$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two}$ representing two's complement signed integers, perform, showing all work:

a. $x + y$

b. $x - y$

c. $x * y$

d. x/y

3.28 [20] <§§3.3, 3.4, 3.5> Perform the same operations as Exercise 3.27, but with $x = 1111\ 1111\ 1111\ 1111\ 1011\ 0011\ 0101\ 0011$ and $y = 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 1101\ 0111_{two}$.

3.29 [30] <§3.5> The division algorithm in Figure 3.11 on page 185 is called *restoring division*, since each time the result of subtracting the divisor from the dividend is negative you must add the divisor back into the dividend to restore the original value. Recall that shift left is the same as multiplying by 2. Let's look at the value of the left half of the Remainder again, starting with step 3b of the divide algorithm and then going to step 2:

$$(\text{Remainder} + \text{Divisor}) \times 2 - \text{Divisor}$$

This value is created from restoring the Remainder by adding the Divisor, shifting the sum left, and then subtracting the Divisor. Simplifying the result we get

$$\text{Remainder} \times 2 + \text{Divisor} \times 2 - \text{Divisor} = \text{Remainder} \times 2 + \text{Divisor}$$

Based on this observation, write a *nonrestoring division* algorithm using the notation of Figure 3.11 that does not add the Divisor to the Remainder in step 3b. Show that your algorithm works by dividing $0000\ 1011_{two}$ by 0011_{two}.

3.30 [15] <§§3.2, 3.6> The Big Picture on page 216 mentions that bits have no inherent meaning. Given the bit pattern:

1010 1101 0001 0000 0000 0000 0000 0010

what does it represent, assuming that it is

a. a two's complement integer?

b. an unsigned integer?

c. a single precision floating-point number?

d. a MIPS instruction?

You may find Figures 3.20 (page 208) and A.10.2 (page A-50) useful.

3.31 <§§3.2, 3.6> This exercise is similar to Exercise 3.30, but this time use the bit pattern

0010 0100 1001 0010 0100 1001 0010 0100

3.32 [10] [10] <§3.6> For More Practice: Floating Point Number Representations

3.33 [10] <§3.6> For More Practice: Floating Point Number Representations

3.34 [10] <§3.6> For More Practice: Writing MIPS Code to Perform FP Arithmetic

3.35 [5] <§3.6> Add $2.85_{ten} \times 10^3$ to $9.84_{ten} \times 10^4$, assuming that you have only three significant digits, first with guard and round digits and then without them.

3.36 [5] <§3.6> This exercise is similar to Exercise 3.35, but this time use the numbers $3.63_{ten} \times 10^4$ and $6.87_{ten} \times 10^3$.

3.37 [5] <§3.6> Show the IEEE 754 binary representation for the floating-point number 20_{ten} in single and double precision.

3.38 [5] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 20.5_{ten}.

3.39 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with 0.1_{ten}.

3.40 [10] <§3.6> This exercise is similar to Exercise 3.37, but this time replace the number 20_{ten} with the decimal fraction –5/6.

3.41 [10] <§3.6> Suppose we introduce a new instruction that adds three floating-point numbers. Assuming we add them together with a triple adder, with guard, round, and sticky bits, are we guaranteed results within 1 ulp of the results using two distinct add instructions?

3.42 [15] <§3.6> With $x = 0100\ 0110\ 1101\ 1000\ 0000\ 0000\ 0000\ 0000_{two}$ and $y = 1011\ 1110\ 1110\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

a. $x + y$

b. $x * y$

3.43 [15] <§3.6> With $x = 0101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{two}$, $y = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_{two}$, and $z = 1101\ 1111\ 1011\ 1110\ 0100\ 0000\ 0000\ 0000_{two}$ representing single precision IEEE 754 floating-point numbers, perform, showing all work:

a. $x + y$

b. (result of a) + z

c. Why is this result counterintuitive?

3.44 [20] <§§3.6, 3.7> The IEEE 754 floating-point standard specifies 64-bit double precision with a 53-bit significand (including the implied 1) and an 11-bit exponent. IA-32 offers an extended precision option with a 64-bit significand and a 16-bit exponent.

a. Assuming extended precision is similar to single and double precision, what is the bias in the exponent?

b. What is the range of numbers that can be represented by the extended precision option?

c. How much greater is this accuracy compared to double precision?

3.45 [5] <§§3.6, 3.7> The internal representation of floating point numbers in IA-32 is 80 bits wide. This contains a 16 bit exponent. However it also advertises a 64 bit significand. How is this possible?

3.46 [10] <§3.7> While the IA-32 allows 80-bit floating-point numbers internally, only 64-bit floating-point numbers can be loaded or stored. Starting with only 64-bit numbers, how many operations are required before the full range of the 80-bit exponents are used? Give an example.

3.47 [25] <§3.8> **For More Practice:** Floating Point on Algorithms

3.48 [30] <§3.8> **For More Practice:** Floating Point Rounding Modes

3.49 [30] <§3.8> **For More Practice:** Denormalized Numbers

3.50 [10] <§3.9> **For More Practice:** Evaluating Instruction Frequencies

3.51 [10] <§3.9> **For More Practice:** Evaluating Instruction Frequencies

3.52 [10] <§3.9> For More Practice: Evaluating Instruction Frequencies

3.53 [10] <§3.9> For More Practice: Evaluating Instruction Frequencies

3.54 [15] <§3.9> For More Practice: Evaluating Performance

3.55 [15] <§3.9> For More Practice: Evaluating Performance

Answers To Check Yourself

§3.2. page 168: 3, since each character in a Java string takes 16 bits plus one word for length.
§3.3, page 174: 2.
§3.6, page 217: 3.

Computers in the Real World

Reconstructing the Ancient World

Problem: Analyzing and understanding archeological sites is challenging. Can we find ways to use computers to help researchers to explore archeological sites and artifacts discovered at those sites?

Solution: Archeology is undergoing a revolution with the use of digital tools for mapping out ancient sites, reconstructing damaged artifacts, and recreating ancient sites in three dimensions. Among the important new techniques being used to analyze and recreate sites are the following:

- The use of geographical information systems (GIS) to help accurately measure sites. GIS uses global positioning systems (GPS) to accurately pinpoint locations, allowing fast and precise measurements of a site.
- Laser range finding to obtain accurate measurements of the two- and three-dimensional structure of objects. Laser range-finding is even being used with low-flying aircraft to obtain height measurements.



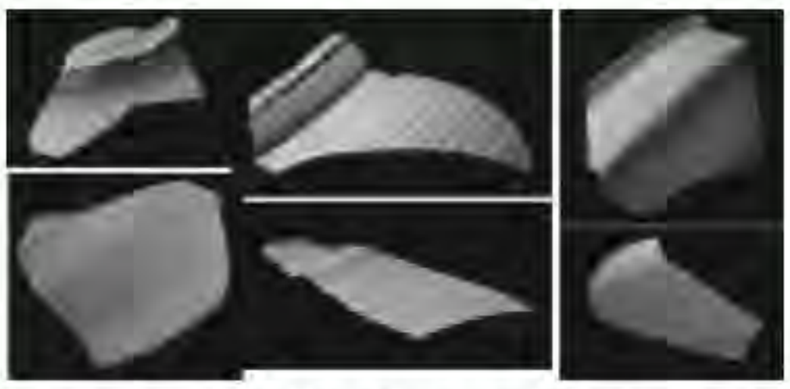
A digital photograph taken from a virtual reality model of the new temple at Chavín de Huántar.

- Digital photography to obtain accurate images of sites as well as individual objects.
- Virtual reality and three-dimensional visualization systems can use digital photographic data and accurate geospatial information to create realistic versions of archeological sites, allowing archeolo-

gists to gain new insights, as well as to share their work with other researchers and the general public.

These techniques have been used to explore and create an interactive virtual reality model of an archeological site called Chavín de Huántar, which is in the Peruvian highlands. The image on the top of the previous page is a digital still photograph taken from the virtual reality model. Chavín de Huántar was occupied from about 1000 BCE and predates the classical Incan civilization by more than 1000 years. Highly detailed photos together with measurements of over 25,000 points allows a reconstruction of an accurate virtual model. The image on the top of the previous page is from the new temple at Chavín, which played a key role in the establishment of formalized religious authority in the New World.

Three-dimensional modeling and reconstruction have also been used in the reconstruction of artifacts from fragments. The images on the left below are fragments of pottery found at Petra, the famous archeological site in Jordan. On the right is a computer reconstruction of the original vessel, highlighting the position of one of the fragments.



Images of pottery fragments found at Petra, Jordan

To learn more see these references on the ◙ library

- Reconstructing objects from fragments at the SHAPE Laboratory at Brown University
- The Chavín de Huántar exploration (includes virtual reality tour of the site)



A computer reconstruction from the fragments in the previous photo

4

Assessing and Understanding Performance

Time discovers truth.

Seneca

4.1 **Introduction** 240

4.2 **CPU Performance and Its Factors** 246

4.3 **Evaluating Performance** 254

4.4 **Real Stuff: Two SPEC Benchmarks and the Performance of Recent Intel Processors** 259

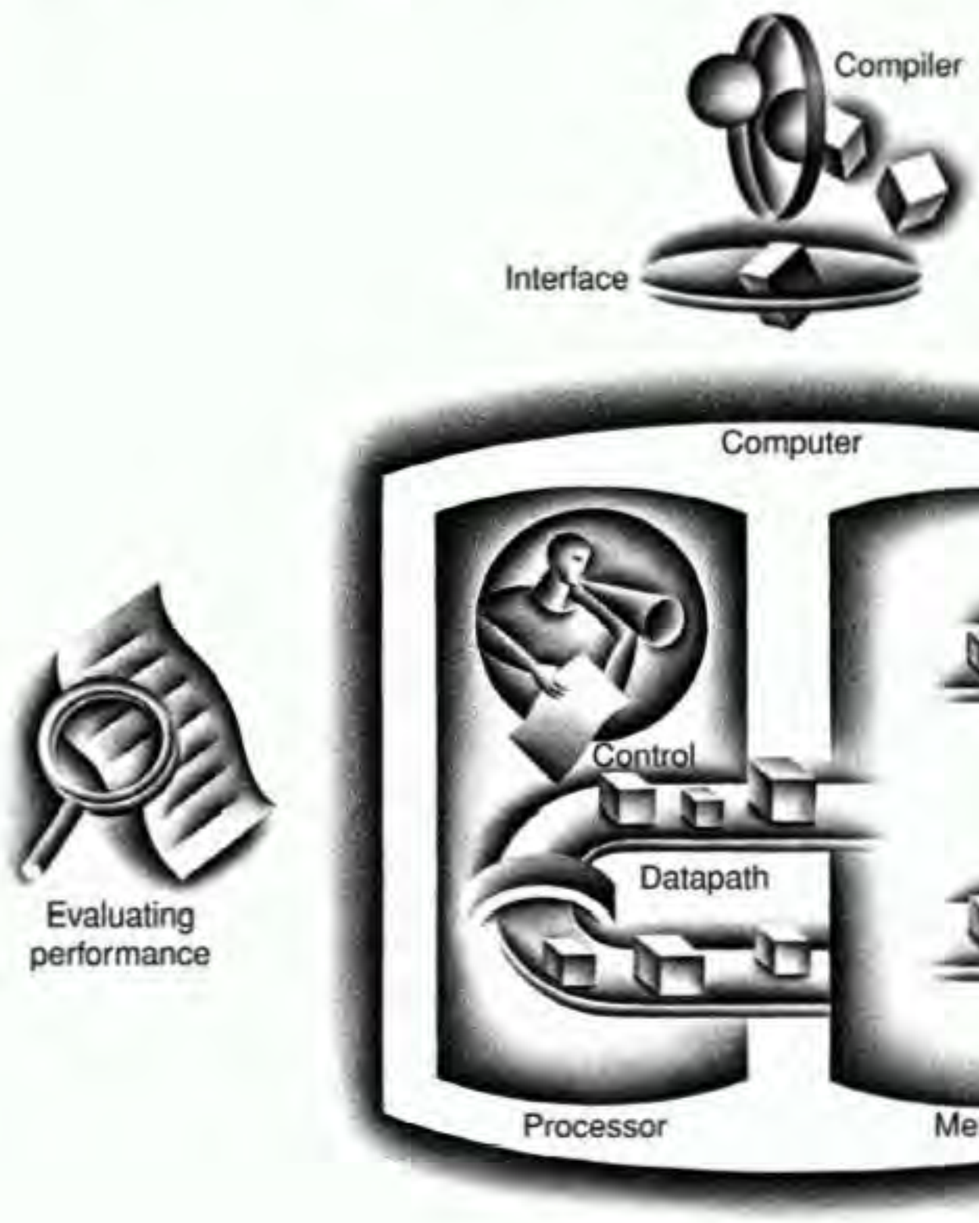
4.5 **Fallacies and Pitfalls** 266

4.6 **Concluding Remarks** 270

4.7 **Historical Perspective and Further Reading** 272

4.8 **Exercises** 272

The Five Classic Components of a Computer



4.1 Introduction

This chapter discusses how to measure, report, and summarize performance and describes the major factors that determine the performance of a computer. A primary reason for examining performance is that hardware performance is often key to the effectiveness of an entire system of hardware and software. Much of the material in this chapter, especially this section and the next, will be critical to understanding the next three chapters. The rest of the chapter provides important insights and principles that anyone seeking to evaluate the performance of a computer system should know. The material in Sections 4.3–4.5, however, is not necessary for the immediately following chapters and can be returned to later.

Assessing the performance of such a system 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. It is simply impossible to sit down with an instruction set manual and a significant software system and determine, simply by analysis, how fast the software will run on the computer. In fact, for different types of applications, different performance metrics may be appropriate, and different aspects of a computer system may be the most significant in determining overall performance.

Of course, in trying to choose among different computers, performance is almost always an important attribute. Accurately measuring and comparing different computers is critical to 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. In some cases, claims are made about computers that don't provide useful insight for any real applications. Hence, understanding how best to measure performance and the limitations of performance measurements is important in selecting a computer.

Our interest in performance, however, goes beyond issues of assessing performance only from the outside of a computer. To understand why a piece of software performs as it does, why one instruction set can be implemented to perform better than another, or how some hardware feature affects performance, we need to understand what determines the performance of a computer. For example, to improve the performance of a software system, we may need to understand what factors in the hardware contribute to the overall system performance and the relative importance of these factors. These factors may include how well the program uses the instructions of the computer, how well the underlying hardware

implements the instructions, and how well the memory and I/O systems perform. Understanding how to determine the performance impact of these factors is crucial to understanding the motivation behind the design of particular aspects of the computer, as we will see in the chapters that follow.

The rest of this section describes different ways in which performance can be determined. In Section 4.2, 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. Sections 4.3 and 4.4 describe how best to choose benchmarks to evaluate computers and how to accurately summarize the performance of a group of programs. Section 4.4 also describes one set of commonly used CPU benchmarks and examines measurements for a variety of Intel processors using those benchmarks. Finally, in Section 4.5, we'll examine some of the many pitfalls that have trapped designers and those who analyze and report performance.

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 4.1 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 were interested in transporting 450 passengers from one point to another, how-

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers x 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 4.1 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).

ever, 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 data center 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**. Data center managers are often interested in increasing *throughput*—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 desktop computers versus servers, and embedded computers require yet other metrics and applications. We will see examples of this in section 4.4 when we look at different SPEC benchmarks: one meant to measure CPU performance (SPEC CPU) and one meant to measure Web server performance (SPECweb99).

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.

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. (In Chapter 8, on input/output systems, we will discuss throughput-related measures.) 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."

Elaboration: Performance in embedded systems is often characterized by real-time constraints: that is, certain application-specific events must occur within a limited amount of time. There are two common types of real-time constraints: hard real time and soft real time. Hard real time defines a fixed bound on the time to respond to or process some event. For example, the embedded processor that controls an antilock brake system must respond within a hard limit from the time it receives a signal that the wheels are locked. In soft real-time systems, an average response or a response within a limited time to a large fraction of the events suffices. For example, handling video frames in a DVD playback system would be an example of a soft real-time constraint, since dropping a frame is permissible, if it happens very rarely. In embedded real-time applications, once the response-time performance constraint is met, designers often optimize throughput or try to reduce cost.

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. But 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.

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

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