

COMPUTER ORGANIZATION AND DESIGN

THE HARDWARE/SOFTWARE INTERFACE

FIFTH EDITION

DAVID A. PATTERSON
JOHN L. HENNESSY



MK
MORGAN KAUFMANN

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

“Textbook selection is often a frustrating act of compromise—pedagogy, content coverage, quality of exposition, level of rigor, cost. *Computer Organization and Design* is the rare book that hits all the right notes across the board, without compromise. It is not only the premier computer organization textbook, it is a shining example of what all computer science textbooks could and should be.”

—Michael Goldweber, *Xavier University*

“I have been using *Computer Organization and Design* for years, from the very first edition. The new Fifth Edition is yet another outstanding improvement on an already classic text. The evolution from desktop computing to mobile computing to Big Data brings new coverage of embedded processors such as the ARM, new material on how software and hardware interact to increase performance, and cloud computing. All this without sacrificing the fundamentals.”

—Ed Harcourt, *St. Lawrence University*

“To Millennials: *Computer Organization and Design* is the computer architecture book you should keep on your (virtual) bookshelf. The book is both old and new, because it develops venerable principles—Moore’s Law, abstraction, common case fast, redundancy, memory hierarchies, parallelism, and pipelining—but illustrates them with contemporary designs, e.g., ARM Cortex A8 and Intel Core i7.”

—Mark D. Hill, *University of Wisconsin-Madison*

“The new edition of *Computer Organization and Design* keeps pace with advances in emerging embedded and many-core (GPU) systems, where tablets and smartphones will quickly become our new desktops. This text acknowledges these changes, but continues to provide a rich foundation of the fundamentals in computer organization and design which will be needed for the designers of hardware and software that power this new class of devices and systems.”

—Dave Kaeli, *Northeastern University*

“The Fifth Edition of *Computer Organization and Design* provides more than an introduction to computer architecture. It prepares the reader for the changes necessary to meet the ever-increasing performance needs of mobile systems and big data processing at a time that difficulties in semiconductor scaling are making all systems power constrained. In this new era for computing, hardware and software must be co-designed and system-level architecture is as critical as component-level optimizations.”

—Christos Kozyrakis, *Stanford University*

“Patterson and Hennessy brilliantly address the issues in ever-changing computer hardware architectures, emphasizing on interactions among hardware and software components at various abstraction levels. By interspersing I/O and parallelism concepts with a variety of mechanisms in hardware and software throughout the book, the new edition achieves an excellent holistic presentation of computer architecture for the PostPC era. This book is an essential guide to hardware and software professionals facing energy efficiency and parallelization challenges in Tablet PC to cloud computing.”

—Jae C. Oh, *Syracuse University*

This page intentionally left blank

F I F T H E D I T I O N

Computer Organization and Design

THE HARDWARE / SOFTWARE INTERFACE

David A. Patterson has been teaching computer architecture at the University of California, Berkeley, since joining the faculty in 1977, where he holds the Pardee Chair of Computer Science. His teaching has been honored by the Distinguished Teaching Award from the University of California, the Karlstrom Award from ACM, and the Mulligan Education Medal and Undergraduate Teaching Award from IEEE. Patterson received the IEEE Technical Achievement Award and the ACM Eckert-Mauchly Award for contributions to RISC, and he shared the IEEE Johnson Information Storage Award for contributions to RAID. He also shared the IEEE John von Neumann Medal and the C & C Prize with John Hennessy. Like his co-author, Patterson is a Fellow of the American Academy of Arts and Sciences, the Computer History Museum, ACM, and IEEE, and he was elected to the National Academy of Engineering, the National Academy of Sciences, and the Silicon Valley Engineering Hall of Fame. He served on the Information Technology Advisory Committee to the U.S. President, as chair of the CS division in the Berkeley EECS department, as chair of the Computing Research Association, and as President of ACM. This record led to Distinguished Service Awards from ACM and CRA.

At Berkeley, Patterson led the design and implementation of RISC I, likely the first VLSI reduced instruction set computer, and the foundation of the commercial SPARC architecture. He was a leader of the Redundant Arrays of Inexpensive Disks (RAID) project, which led to dependable storage systems from many companies. He was also involved in the Network of Workstations (NOW) project, which led to cluster technology used by Internet companies and later to cloud computing. These projects earned three dissertation awards from ACM. His current research projects are Algorithm-Machine-People and Algorithms and Specializers for Provably Optimal Implementations with Resilience and Efficiency. The AMP Lab is developing scalable machine learning algorithms, warehouse-scale-computer-friendly programming models, and crowd-sourcing tools to gain valuable insights quickly from big data in the cloud. The ASPIRE Lab uses deep hardware and software co-tuning to achieve the highest possible performance and energy efficiency for mobile and rack computing systems.

John L. Hennessy is the tenth president of Stanford University, where he has been a member of the faculty since 1977 in the departments of electrical engineering and computer science. Hennessy is a Fellow of the IEEE and ACM; a member of the National Academy of Engineering, the National Academy of Science, and the American Philosophical Society; and a Fellow of the American Academy of Arts and Sciences. Among his many awards are the 2001 Eckert-Mauchly Award for his contributions to RISC technology, the 2001 Seymour Cray Computer Engineering Award, and the 2000 John von Neumann Award, which he shared with David Patterson. He has also received seven honorary doctorates.

In 1981, he started the MIPS project at Stanford with a handful of graduate students. After completing the project in 1984, he took a leave from the university to cofound MIPS Computer Systems (now MIPS Technologies), which developed one of the first commercial RISC microprocessors. As of 2006, over 2 billion MIPS microprocessors have been shipped in devices ranging from video games and palmtop computers to laser printers and network switches. Hennessy subsequently led the DASH (Director Architecture for Shared Memory) project, which prototyped the first scalable cache coherent multiprocessor; many of the key ideas have been adopted in modern multiprocessors. In addition to his technical activities and university responsibilities, he has continued to work with numerous start-ups both as an early-stage advisor and an investor.

F I F T H E D I T I O N

Computer Organization and Design

THE HARDWARE / SOFTWARE INTERFACE

David A. Patterson

University of California, Berkeley

John L. Hennessy

Stanford University

With contributions by

Perry Alexander

The University of Kansas

Peter J. Ashenden

Ashenden Designs Pty Ltd

Jason D. Bakos

University of South Carolina

Javier Bruguera

Universidade de Santiago de Compostela

Jichuan Chang

Hewlett-Packard

Matthew Farrens

University of California, Davis

David Kaeli

Northeastern University

Nicole Kaiyan

University of Adelaide

David Kirk

NVIDIA

James R. Larus

School of Computer and
Communications Science at EPFL

Jacob Leverich

Hewlett-Packard

Kevin Lim

Hewlett-Packard

John Nickolls

NVIDIA

John Oliver

Cal Poly, San Luis Obispo

Milos Prvulovic

Georgia Tech

Partha Ranganathan

Hewlett-Packard



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



Acquiring Editor: Todd Green
Development Editor: Nate McFadden
Project Manager: Lisa Jones
Designer: Russell Purdy

Morgan Kaufmann is an imprint of Elsevier
The Boulevard, Langford Lane, Kidlington, Oxford, OX5 1GB
225 Wyman Street, Waltham, MA 02451, USA

Copyright © 2014 Elsevier Inc. All rights reserved

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

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

Notices

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

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

Library of Congress Cataloging-in-Publication Data

Patterson, David A.

Computer organization and design: the hardware/software interface/David A. Patterson, John L. Hennessy. — 5th ed.

p. cm. — (The Morgan Kaufmann series in computer architecture and design)

Rev. ed. of: Computer organization and design/John L. Hennessy, David A. Patterson. 1998.

Summary: "Presents the fundamentals of hardware technologies, assembly language, computer arithmetic, pipelining, memory hierarchies and I/O"— Provided by publisher.

ISBN 978-0-12-407726-3 (pbk.)

1. Computer organization. 2. Computer engineering. 3. Computer interfaces. I. Hennessy, John L. II. Hennessy, John L. Computer organization and design. III. Title.

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-12-407726-3

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

Printed and bound in the United States of America

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



Working together
to grow libraries in
developing countries

www.elsevier.com • www.bookaid.org

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

A C K N O W L E D G M E N T S

Figures 1.7, 1.8 Courtesy of iFixit (www.ifixit.com).

Figure 1.9 Courtesy of Chipworks ([www\(chipworks.com](http://www(chipworks.com))).

Figure 1.13 Courtesy of Intel.

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

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

Figure 1.10.4 Courtesy of Cray Inc.

Figure 1.10.5 Courtesy of Apple Computer, Inc.

Figure 1.10.6 Courtesy of the Computer History Museum.

Figures 5.17.1, 5.17.2 Courtesy of Museum of Science, Boston.

Figure 5.17.4 Courtesy of MIPS Technologies, Inc.

Figure 6.15.1 Courtesy of NASA Ames Research Center.

Contents

Preface xv

CHAPTERS

1

Computer Abstractions and Technology 2

- 1.1 Introduction 3
- 1.2 Eight Great Ideas in Computer Architecture 11
- 1.3 Below Your Program 13
- 1.4 Under the Covers 16
- 1.5 Technologies for Building Processors and Memory 24
- 1.6 Performance 28
- 1.7 The Power Wall 40
- 1.8 The Sea Change: The Switch from Uniprocessors to Multiprocessors 43
- 1.9 Real Stuff: Benchmarking the Intel Core i7 46
- 1.10 Fallacies and Pitfalls 49
- 1.11 Concluding Remarks 52
-  1.12 Historical Perspective and Further Reading 54
- 1.13 Exercises 54

2

Instructions: Language of the Computer 60

- 2.1 Introduction 62
- 2.2 Operations of the Computer Hardware 63
- 2.3 Operands of the Computer Hardware 66
- 2.4 Signed and Unsigned Numbers 73
- 2.5 Representing Instructions in the Computer 80
- 2.6 Logical Operations 87
- 2.7 Instructions for Making Decisions 90
- 2.8 Supporting Procedures in Computer Hardware 96
- 2.9 Communicating with People 106
- 2.10 MIPS Addressing for 32-Bit Immediates and Addresses 111
- 2.11 Parallelism and Instructions: Synchronization 121
- 2.12 Translating and Starting a Program 123
- 2.13 A C Sort Example to Put It All Together 132
- 2.14 Arrays versus Pointers 141

	2.15 Advanced Material: Compiling C and Interpreting Java 145
	2.16 Real Stuff: ARMv7 (32-bit) Instructions 145
	2.17 Real Stuff: x86 Instructions 149
	2.18 Real Stuff: ARMv8 (64-bit) Instructions 158
	2.19 Fallacies and Pitfalls 159
	2.20 Concluding Remarks 161
	2.21 Historical Perspective and Further Reading 163
	2.22 Exercises 164

3

Arithmetic for Computers 176

3.1	Introduction 178
3.2	Addition and Subtraction 178
3.3	Multiplication 183
3.4	Division 189
3.5	Floating Point 196
3.6	Parallelism and Computer Arithmetic: Subword Parallelism 222
3.7	Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86 224
3.8	Going Faster: Subword Parallelism and Matrix Multiply 225
3.9	Fallacies and Pitfalls 229
3.10	Concluding Remarks 232
	3.11 Historical Perspective and Further Reading 236
	3.12 Exercises 237

4

The Processor 242

4.1	Introduction 244
4.2	Logic Design Conventions 248
4.3	Building a Datapath 251
4.4	A Simple Implementation Scheme 259
4.5	An Overview of Pipelining 272
4.6	Pipelined Datapath and Control 286
4.7	Data Hazards: Forwarding versus Stalling 303
4.8	Control Hazards 316
4.9	Exceptions 325
4.10	Parallelism via Instructions 332
4.11	Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines 344
4.12	Going Faster: Instruction-Level Parallelism and Matrix Multiply 351
	4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations 354

- 4.14 Fallacies and Pitfalls 355
- 4.15 Concluding Remarks 356
-  4.16 Historical Perspective and Further Reading 357
- 4.17 Exercises 357

5

Large and Fast: Exploiting Memory Hierarchy 372

- 5.1 Introduction 374
- 5.2 Memory Technologies 378
- 5.3 The Basics of Caches 383
- 5.4 Measuring and Improving Cache Performance 398
- 5.5 Dependable Memory Hierarchy 418
- 5.6 Virtual Machines 424
- 5.7 Virtual Memory 427
- 5.8 A Common Framework for Memory Hierarchy 454
- 5.9 Using a Finite-State Machine to Control a Simple Cache 461
-  5.10 Parallelism and Memory Hierarchies: Cache Coherence 466
-  5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks 470
-  5.12 Advanced Material: Implementing Cache Controllers 470
- 5.13 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Memory Hierarchies 471
- 5.14 Going Faster: Cache Blocking and Matrix Multiply 475
- 5.15 Fallacies and Pitfalls 478
- 5.16 Concluding Remarks 482
-  5.17 Historical Perspective and Further Reading 483
- 5.18 Exercises 483

6

Parallel Processors from Client to Cloud 500

- 6.1 Introduction 502
- 6.2 The Difficulty of Creating Parallel Processing Programs 504
- 6.3 SISD, MIMD, SIMD, SPMD, and Vector 509
- 6.4 Hardware Multithreading 516
- 6.5 Multicore and Other Shared Memory Multiprocessors 519
- 6.6 Introduction to Graphics Processing Units 524
- 6.7 Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors 531
- 6.8 Introduction to Multiprocessor Network Topologies 536
-  6.9 Communicating to the Outside World: Cluster Networking 539
- 6.10 Multiprocessor Benchmarks and Performance Models 540
- 6.11 Real Stuff: Benchmarking Intel Core i7 versus NVIDIA Tesla GPU 550

	6.12 Going Faster: Multiple Processors and Matrix Multiply	555
	6.13 Fallacies and Pitfalls	558
	6.14 Concluding Remarks	560
	6.15 Historical Perspective and Further Reading	563
	6.16 Exercises	563

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-9
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-48
B.8	Memory Elements: Flip-Flops, Latches, and Registers	B-50
B.9	Memory Elements: SRAMs and DRAMs	B-58
B.10	Finite-State Machines	B-67
B.11	Timing Methodologies	B-72
B.12	Field Programmable Devices	B-78
B.13	Concluding Remarks	B-79
B.14	Exercises	B-80

ONLINE CONTENT**Graphics and Computing GPUs C-2**

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

**Mapping Control to Hardware D-2**

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

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

- E.1 Introduction E-3
- E.2 Addressing Modes and Instruction Formats E-5
- E.3 Instructions: The MIPS Core Subset E-9
- E.4 Instructions: Multimedia Extensions of the Desktop/Server RISCs E-16
- E.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs E-19
- E.6 Instructions: Common Extensions to MIPS Core E-20
- E.7 Instructions Unique to MIPS-64 E-25
- E.8 Instructions Unique to Alpha E-27
- E.9 Instructions Unique to SPARC v9 E-29
- E.10 Instructions Unique to PowerPC E-32
- E.11 Instructions Unique to PA-RISC 2.0 E-34
- E.12 Instructions Unique to ARM E-36
- E.13 Instructions Unique to Thumb E-38
- E.14 Instructions Unique to SuperH E-39

E.15 Instructions Unique to M32R E-40
E.16 Instructions Unique to MIPS-16 E-40
E.17 Concluding Remarks E-43

 Glossary G-1

 Further Reading FR-1

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, energy, and, ultimately, the success of computer systems.

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

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

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

About the Other Book

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

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

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

Changes for the Fifth Edition

We had six major goals for the fifth edition of *Computer Organization and Design*: demonstrate the importance of understanding hardware with a running example; highlight major themes across the topics using margin icons that are introduced early; update examples to reflect changeover from PC era to PostPC era; spread the material on I/O throughout the book rather than isolating it into a single chapter; update the technical content to reflect changes in the industry since the publication of the fourth edition in 2009; and put appendices and optional sections online instead of including a CD to lower costs and to make this edition viable as an electronic book.

Before discussing the goals in detail, let's look at the table on the next page. It shows the hardware and software paths through the material. Chapters 1, 4, 5, and 6 are found on both paths, no matter what the experience or the focus. Chapter 1 discusses the importance of energy and how it motivates the switch from single core to multicore microprocessors and introduces the eight great ideas in computer architecture. Chapter 2 is likely to be review material for the hardware-oriented, but it is essential reading for the software-oriented, especially for those readers interested in learning more about compilers and object-oriented programming languages. Chapter 3 is for readers interested in constructing a datapath or in

Chapter or Appendix	Sections	Software focus	Hardware focus
1. Computer Abstractions and Technology	1.1 to 1.11		
	1.12 (History)		
2. Instructions: Language of the Computer	2.1 to 2.14		
	2.15 (Compilers & Java)		
	2.16 to 2.20		
	2.21 (History)		
E. RISC Instruction-Set Architectures	E.1 to E.17		
3. Arithmetic for Computers	3.1 to 3.5		
	3.6 to 3.8 (Subword Parallelism)		
	3.9 to 3.10 (Fallacies)		
	3.11 (History)		
B. The Basics of Logic Design	B.1 to B.13		
4. The Processor	4.1 (Overview)		
	4.2 (Logic Conventions)		
	4.3 to 4.4 (Simple Implementation)		
	4.5 (Pipelining Overview)		
	4.6 (Pipelined Datapath)		
	4.7 to 4.9 (Hazards, Exceptions)		
	4.10 to 4.12 (Parallel, Real Stuff)		
	4.13 (Verilog Pipeline Control)		
	4.14 to 4.15 (Fallacies)		
	4.16 (History)		
D. Mapping Control to Hardware	D.1 to D.6		
5. Large and Fast: Exploiting Memory Hierarchy	5.1 to 5.10		
	5.11 (Redundant Arrays of Inexpensive Disks)		
	5.12 (Verilog Cache Controller)		
	5.13 to 5.16		
	5.17 (History)		
	6.1 to 6.8		
6. Parallel Process from Client to Cloud	6.9 (Networks)		
	6.10 to 6.14		
	6.15 (History)		
	A.1 to A.11		
C. Graphics Processor Units	C.1 to C.13		

Read carefully

Read if have time

Reference

Review or read

Read for culture

learning more about floating-point arithmetic. Some will skip parts of Chapter 3, either because they don't need them or because they offer a review. However, we introduce the running example of matrix multiply in this chapter, showing how subword parallelism offers a fourfold improvement, so don't skip sections 3.6 to 3.8. Chapter 4 explains pipelined processors. Sections 4.1, 4.5, and 4.10 give overviews and Section 4.12 gives the next performance boost for matrix multiply for those with a software focus. Those with a hardware focus, however, will find that this chapter presents core material; they may also, depending on their background, want to read Appendix C on logic design first. The last chapter on multicores, multiprocessors, and clusters, is mostly new content and should be read by everyone. It was significantly reorganized in this edition to make the flow of ideas more natural and to include much more depth on GPUs, warehouse scale computers, and the hardware-software interface of network interface cards that are key to clusters.

The first of the six goals for this fifth edition was to demonstrate the importance of understanding modern hardware to get good performance and energy efficiency with a concrete example. As mentioned above, we start with subword parallelism in Chapter 3 to improve matrix multiply by a factor of 4. We double performance in Chapter 4 by unrolling the loop to demonstrate the value of instruction level parallelism. Chapter 5 doubles performance again by optimizing for caches using blocking. Finally, Chapter 6 demonstrates a speedup of 14 from 16 processors by using thread-level parallelism. All four optimizations in total add just 24 lines of C code to our initial matrix multiply example.

The second goal was to help readers separate the forest from the trees by identifying eight great ideas of computer architecture early and then pointing out all the places they occur throughout the rest of the book. We use (hopefully) easy to remember margin icons and highlight the corresponding word in the text to remind readers of these eight themes. There are nearly 100 citations in the book. No chapter has less than seven examples of great ideas, and no idea is cited less than five times. Performance via parallelism, pipelining, and prediction are the three most popular great ideas, followed closely by Moore's Law. The processor chapter (4) is the one with the most examples, which is not a surprise since it probably received the most attention from computer architects. The one great idea found in every chapter is performance via parallelism, which is a pleasant observation given the recent emphasis in parallelism in the field and in editions of this book.

The third goal was to recognize the generation change in computing from the PC era to the PostPC era by this edition with our examples and material. Thus, Chapter 1 dives into the guts of a tablet computer rather than a PC, and Chapter 6 describes the computing infrastructure of the cloud. We also feature the ARM, which is the instruction set of choice in the personal mobile devices of the PostPC era, as well as the x86 instruction set that dominated the PC Era and (so far) dominates cloud computing.

The fourth goal was to spread the I/O material throughout the book rather than have it in its own chapter, much as we spread parallelism throughout all the chapters in the fourth edition. Hence, I/O material in this edition can be found in

Sections 1.4, 4.9, 5.2, 5.5, 5.11, and 6.9. The thought is that readers (and instructors) are more likely to cover I/O if it's not segregated to its own chapter.

This is a fast-moving field, and, as is always the case for our new editions, an important goal is to update the technical content. The running example is the ARM Cortex A8 and the Intel Core i7, reflecting our PostPC Era. Other highlights include an overview the new 64-bit instruction set of ARMv8, a tutorial on GPUs that explains their unique terminology, more depth on the warehouse scale computers that make up the cloud, and a deep dive into 10 Gigabyte Ethernet cards.

To keep the main book short and compatible with electronic books, we placed the optional material as online appendices instead of on a companion CD as in prior editions.

Finally, we updated all the exercises in the book.

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

Changes for the Fifth Edition

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

textbooks.elsevier.com/9780124077263

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 cod5bugs@mfp.com or by low-tech mail using the address found on the copyright page.

This edition is the second break in the long-standing collaboration between Hennessy and Patterson, which started in 1989. The demands of running one of the world’s great universities meant that President Hennessy could no longer make the substantial commitment to create a new edition. The remaining author felt

once again like a tightrope walker without a safety net. Hence, the people in the acknowledgments and Berkeley colleagues played an even larger role in shaping the contents of this book. Nevertheless, this time around there is only one author to blame for the new material in what you are about to read.

Acknowledgments for the Fifth Edition

With every edition of this book, we are very fortunate to receive help from many readers, reviewers, and contributors. Each of these people has helped to make this book better.

Chapter 6 was so extensively revised that we did a separate review for ideas and contents, and I made changes based on the feedback from every reviewer. I'd like to thank **Christos Kozyrakis** of Stanford University for suggesting using the network interface for clusters to demonstrate the hardware-software interface of I/O and for suggestions on organizing the rest of the chapter; **Mario Flagsilk** of Stanford University for providing details, diagrams, and performance measurements of the NetFPGA NIC; and the following for suggestions on how to improve the chapter: **David Kaeli** of Northeastern University, **Partha Ranganathan** of HP Labs, **David Wood** of the University of Wisconsin, and my Berkeley colleagues **Siamak Faridani**, **Shoaib Kamil**, **Yunsup Lee**, **Zhangxi Tan**, and **Andrew Waterman**.

Special thanks goes to **Rimas Avizienis** of UC Berkeley, who developed the various versions of matrix multiply and supplied the performance numbers as well. As I worked with his father while I was a graduate student at UCLA, it was a nice symmetry to work with Rimas at UCB.

I also wish to thank my longtime collaborator **Randy Katz** of UC Berkeley, who helped develop the concept of great ideas in computer architecture as part of the extensive revision of an undergraduate class that we did together.

I'd like to thank **David Kirk**, **John Nickolls**, and their colleagues at NVIDIA (Michael Garland, John Montrym, Doug Voorhies, Lars Nyland, Erik Lindholm, Paulius Micikevicius, Massimiliano Fatica, Stuart Oberman, and Vasily Volkov) for writing the first in-depth appendix on GPUs. I'd like to express again my appreciation to **Jim Larus**, recently named Dean of the School of Computer and Communications Science at EPFL, for his willingness in contributing his expertise on assembly language programming, as well as for welcoming readers of this book with regard to using the simulator he developed and maintains.

I am also very grateful to **Jason Bakos** of the University of South Carolina, who updated and created new exercises for this edition, working from originals prepared for the fourth edition by **Perry Alexander** (The University of Kansas); **Javier Bruguera** (Universidade de Santiago de Compostela); **Matthew Farrens** (University of California, Davis); **David Kaeli** (Northeastern University); **Nicole Kaiyan** (University of Adelaide); **John Oliver** (Cal Poly, San Luis Obispo); **Milos Prvulovic** (Georgia Tech); and **Jichuan Chang**, **Jacob Leverich**, **Kevin Lim**, and **Partha Ranganathan** (all from Hewlett-Packard).

Additional thanks goes to **Jason Bakos** for developing the new lecture slides.

I am grateful to the many instructors who have answered the publisher's surveys, reviewed our proposals, and attended focus groups to analyze and respond to our plans for this edition. They include the following individuals: Focus Groups in 2012: Bruce Barton (Suffolk County Community College), Jeff Braun (Montana Tech), Ed Gehringer (North Carolina State), Michael Goldweber (Xavier University), Ed Harcourt (St. Lawrence University), Mark Hill (University of Wisconsin, Madison), Patrick Homer (University of Arizona), Norm Jouppi (HP Labs), Dave Kaeli (Northeastern University), Christos Kozyrakis (Stanford University), Zachary Kurmas (Grand Valley State University), Jae C. Oh (Syracuse University), Lu Peng (LSU), Milos Prvulovic (Georgia Tech), Partha Ranganathan (HP Labs), David Wood (University of Wisconsin), Craig Zilles (University of Illinois at Urbana-Champaign). Surveys and Reviews: Mahmoud Abou-Nasr (Wayne State University), Perry Alexander (The University of Kansas), Hakan Aydin (George Mason University), Hussein Badr (State University of New York at Stony Brook), Mac Baker (Virginia Military Institute), Ron Barnes (George Mason University), Douglas Blough (Georgia Institute of Technology), Kevin Bolding (Seattle Pacific University), Miodrag Bolic (University of Ottawa), John Bonomo (Westminster College), Jeff Braun (Montana Tech), Tom Briggs (Shippensburg University), Scott Burgess (Humboldt State University), Fazli Can (Bilkent University), Warren R. Carithers (Rochester Institute of Technology), Bruce Carlton (Mesa Community College), Nicholas Carter (University of Illinois at Urbana-Champaign), Anthony Cocchi (The City University of New York), Don Cooley (Utah State University), Robert D. Cupper (Allegheny College), Edward W. Davis (North Carolina State University), Nathaniel J. Davis (Air Force Institute of Technology), Molisa Derk (Oklahoma City University), Derek Eager (University of Saskatchewan), Ernest Ferguson (Northwest Missouri State University), Rhonda Kay Gaede (The University of Alabama), Etienne M. Gagnon (UQAM), Costa Gerousis (Christopher Newport University), Paul Gillard (Memorial University of Newfoundland), Michael Goldweber (Xavier University), Georgia Grant (College of San Mateo), Merrill Hall (The Master's College), Tyson Hall (Southern Adventist University), Ed Harcourt (St. Lawrence University), Justin E. Harlow (University of South Florida), Paul F. Hemler (Hampden-Sydney College), Martin Herboldt (Boston University), Steve J. Hodges (Cabrillo College), Kenneth Hopkinson (Cornell University), Dalton Hunkins (St. Bonaventure University), Baback Izadi (State University of New York—New Paltz), Reza Jafari, Robert W. Johnson (Colorado Technical University), Bharat Joshi (University of North Carolina, Charlotte), Nagarajan Kandasamy (Drexel University), Rajiv Kapadia, Ryan Kastner (University of California, Santa Barbara), E.J. Kim (Texas A&M University), Jihong Kim (Seoul National University), Jim Kirk (Union University), Geoffrey S. Knauth (Lycoming College), Manish M. Kochhal (Wayne State), Suzan Koknar-Tezel (Saint Joseph's University), Angkul Kongmunvattana (Columbus State University), April Kontostathis (Ursinus College), Christos Kozyrakis (Stanford University), Danny Krizanc (Wesleyan University), Ashok Kumar, S. Kumar (The University of Texas), Zachary Kurmas (Grand Valley State University), Robert N. Lea (University of Houston), Baoxin

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

A special thanks also goes to **Mark Smotherman** for making multiple passes to find technical and writing glitches that significantly improved the quality of this edition.

We wish to thank the extended Morgan Kaufmann family for agreeing to publish this book again under the able leadership of **Todd Green** and **Nate McFadden**: I certainly couldn't have completed the book without them. We also want to extend thanks to **Lisa Jones**, who managed the book production process, and **Russell Purdy**, who did the cover design. The new cover cleverly connects the PostPC Era content of this edition to the cover of the first edition.

The contributions of the nearly 150 people we mentioned here have helped make this fifth edition what I hope will be our best book yet. Enjoy!

David A. Patterson

This page intentionally left blank

1

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

Computer Abstractions and Technology

- 1.1 Introduction** 3
- 1.2 Eight Great Ideas in Computer Architecture** 11
- 1.3 Below Your Program** 13
- 1.4 Under the Covers** 16
- 1.5 Technologies for Building Processors and Memory** 24

1.6	Performance	28
1.7	The Power Wall	40
1.8	The Sea Change: The Switch from Uniprocessors to Multiprocessors	43
1.9	Real Stuff: Benchmarking the Intel Core i7	46
1.10	Fallacies and Pitfalls	49
1.11	Concluding Remarks	52
 1.12	Historical Perspective and Further Reading	54
1.13	Exercises	54

1.1

Introduction

Welcome to this book! We're delighted to have this opportunity to convey the excitement of the world of computer systems. This is not a dry and dreary field, where progress is glacial and where new ideas atrophy from neglect. No! Computers are the product of the incredibly vibrant information technology industry, all aspects of which are responsible for almost 10% of the gross national product of the United States, and whose economy has become dependent in part on the rapid improvements in information technology promised by Moore's Law. This unusual industry embraces innovation at a breath-taking rate. In the last 30 years, there have been a number of new computers whose introduction appeared to revolutionize the computing industry; these revolutions were cut short only because someone else built an even better computer.

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

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

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

- *Computers in automobiles:* Until microprocessors improved dramatically in price and performance in the early 1980s, computer control of cars was ludicrous. Today, computers reduce pollution, improve fuel efficiency via engine controls, and increase safety through blind spot warnings, lane departure warnings, moving object detection, and air bag inflation to protect occupants in a crash.
- *Cell phones:* Who would have dreamed that advances in computer systems would lead to more than half of the planet having mobile phones, allowing person-to-person communication to almost anyone anywhere in the world?
- *Human genome project:* The cost of computer equipment to map and analyze human DNA sequences was 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 15 to 25 years earlier. Moreover, costs continue to drop; you will soon be able to acquire your own genome, allowing medical care to be tailored to you.
- *World Wide Web:* Not in existence at the time of the first edition of this book, the web has transformed our society. For many, the web has replaced libraries and newspapers.
- *Search engines:* As the content of the web grew in size and in value, finding relevant information became increasingly important. Today, many people rely on search engines for such a large part of their lives that it would be a hardship to go without them.

Clearly, advances in this technology now affect almost every aspect of our society. Hardware advances have allowed programmers to create wonderfully useful software, which explains why computers are omnipresent. Today's science fiction suggests tomorrow's killer applications: already on their way are glasses that augment reality, the cashless society, and cars that can drive themselves.

Classes of Computing Applications and Their Characteristics

Although a common set of hardware technologies (see Sections 1.4 and 1.5) 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.

Personal computers (PCs) are possibly the best known form of computing, which readers of this book have likely used extensively. Personal computers emphasize delivery of good performance to single users at low cost and usually execute third-party software. This class of computing drove the evolution of many computing technologies, which is only about 35 years old!

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

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

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

personal computer (PC) A computer designed for use by an individual, usually incorporating a graphics display, a keyboard, and a mouse.

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

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

terabyte (TB) Originally 1,099,511,627,776 (2^{40}) bytes, although communications and secondary storage systems developers started using the term to mean 1,000,000,000,000 (10^{12}) bytes. To reduce confusion, we now use the term **tEBiByte (TiB)** for 2^{40} bytes, defining **terabyte (TB)** to mean 10^{12} bytes.

Figure 1.1 shows the full range of decimal and binary values and names.

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

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

FIGURE 1.1 The 2^x vs. 10^y bytes ambiguity was resolved by adding a binary notation for all the common size terms. In the last column we note how much larger the binary term is than its corresponding decimal term, which is compounded as we head down the chart. These prefixes work for bits as well as bytes, so *gigabit* (Gb) is 10^9 bits while *gibibits* (GiB) is 2^{30} bits.

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

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

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

Welcome to the PostPC Era

The continuing march of technology brings about generational changes in computer hardware that shake up the entire information technology industry. Since the last edition of the book we have undergone such a change, as significant in the past as the switch starting 30 years ago to personal computers. Replacing the

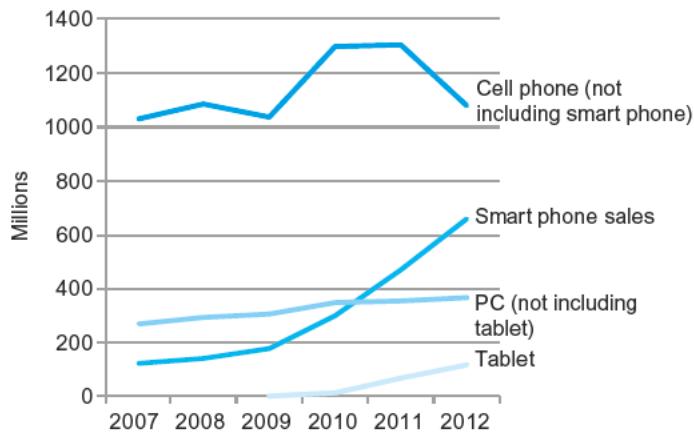


FIGURE 1.2 The number manufactured per year of tablets and smart phones, which reflect the PostPC era, versus personal computers and traditional cell phones. Smart phones represent the recent growth in the cell phone industry, and they passed PCs in 2011. Tablets are the fastest growing category, nearly doubling between 2011 and 2012. Recent PCs and traditional cell phone categories are relatively flat or declining.

PC is the **personal mobile device (PMD)**. PMDs are battery operated with wireless connectivity to the Internet and typically cost hundreds of dollars, and, like PCs, users can download software (“apps”) to run on them. Unlike PCs, they no longer have a keyboard and mouse, and are more likely to rely on a touch-sensitive screen or even speech input. Today’s PMD is a smart phone or a tablet computer, but tomorrow it may include electronic glasses. Figure 1.2 shows the rapid growth time of tablets and smart phones versus that of PCs and traditional cell phones.

Taking over from the traditional server is **Cloud Computing**, which relies upon giant datacenters that are now known as *Warehouse Scale Computers* (WSCs). Companies like Amazon and Google build these WSCs containing 100,000 servers and then let companies rent portions of them so that they can provide software services to PMDs without having to build WSCs of their own. Indeed, **Software as a Service (SaaS)** deployed via the cloud is revolutionizing the software industry just as PMDs and WSCs are revolutionizing the hardware industry. Today’s software developers will often have a portion of their application that runs on the PMD and a portion that runs in the Cloud.

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

Personal mobile devices (PMDs) are small wireless devices to connect to the Internet; they rely on batteries for power, and software is installed by downloading apps. Conventional examples are smart phones and tablets.

Cloud Computing refers to large collections of servers that provide services over the Internet; some providers rent dynamically varying numbers of servers as a utility.

Software as a Service (SaaS) delivers software and data as a service over the Internet, usually via a thin program such as a browser that runs on local client devices, instead of binary code that must be installed, and runs wholly on that device. Examples include web search and social networking.

last decade, advances in computer design and memory technology have greatly reduced the importance of small memory size in most applications other than those in embedded computing systems.

Programmers interested in performance now need to understand the issues that have replaced the simple memory model of the 1960s: the parallel nature of processors and the hierarchical nature of memories. Moreover, as we explain in Section 1.7, today's programmers need to worry about energy efficiency of their programs running either on the PMD or in the Cloud, which also requires understanding what is below your code. Programmers who seek to build competitive versions of software will therefore need to increase their knowledge of computer organization.

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

- How are programs written in a high-level language, such as C or Java, translated into the language of the hardware, and how does the hardware execute the resulting program? Comprehending these concepts forms the basis of understanding the aspects of both the hardware and software that affect program performance.
- What is the interface between the software and the hardware, and how does software instruct the hardware to perform needed functions? These concepts are vital to understanding how to write many kinds of software.
- What determines the performance of a program, and how can a programmer improve the performance? As we will see, this depends on the original program, the software translation of that program into the computer's language, and the effectiveness of the hardware in executing the program.
- What techniques can be used by hardware designers to improve performance? This book will introduce the basic concepts of modern computer design. The interested reader will find much more material on this topic in our advanced book, *Computer Architecture: A Quantitative Approach*.
- What techniques can be used by hardware designers to improve energy efficiency? What can the programmer do to help or hinder energy efficiency?
- What are the reasons for and the consequences of the recent switch from sequential processing to parallel processing? This book gives the motivation, describes the current hardware mechanisms to support parallelism, and surveys the new generation of “**multicore**” **microprocessors** (see Chapter 6).
- Since the first commercial computer in 1951, what great ideas did computer architects come up with that lay the foundation of modern computing?

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

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

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

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

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

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

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

Understanding Program Performance

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

To demonstrate the impact of the ideas in this book, we improve the performance of a C program that multiplies a matrix times a vector in a sequence of chapters. Each step leverages understanding how the underlying hardware really works in a modern microprocessor to improve performance by a factor of 200!

- In the category of *data level parallelism*, in Chapter 3 we use *subword parallelism via C intrinsics* to increase performance by a factor of 3.8.
- In the category of *instruction level parallelism*, in Chapter 4 we use *loop unrolling to exploit multiple instruction issue and out-of-order execution hardware* to increase performance by another factor of 2.3.
- In the category of *memory hierarchy optimization*, in Chapter 5 we use *cache blocking* to increase performance on large matrices by another factor of 2.5.
- In the category of *thread level parallelism*, in Chapter 6 we use *parallel for loops in OpenMP to exploit multicore hardware* to increase performance by another factor of 14.

Check Yourself

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

1. The number of embedded processors sold every year greatly outnumbers the number of PC and even PostPC 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 conventional 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

Eight Great Ideas In Computer Architecture

We now introduce eight great ideas that computer architects have been invented in the last 60 years of computer design. These ideas are so powerful they have lasted long after the first computer that used them, with newer architects demonstrating their admiration by imitating their predecessors. These great ideas are themes that we will weave through this and subsequent chapters as examples arise. To point out their influence, in this section we introduce icons and highlighted terms that represent the great ideas and we use them to identify the nearly 100 sections of the book that feature use of the great ideas.

Design for Moore's Law

The one constant for computer designers is rapid change, which is driven largely by Moore's Law. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an “up and to the right” Moore's Law graph to represent designing for rapid change.



Use Abstraction to Simplify Design

Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by Moore's Law. A major productivity technique for hardware and software is to use **abstractions** to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea.



ABSTRACTION

Make the Common Case Fast

Making the **common case fast** will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement (see Section 1.6). We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan!



COMMON CASE FAST



Performance via Parallelism

Since the dawn of computing, computer architects have offered designs that get more performance by performing operations in parallel. We'll see many examples of parallelism in this book. We use multiple jet engines of a plane as our icon for parallel performance.

Performance via Pipelining

A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: **pipelining**. For example, before fire engines, a "bucket brigade" would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. The townsfolk form a human chain to carry a water source to fire, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.

Performance via Prediction

Following the saying that it can be better to ask for forgiveness than to ask for permission, the final great idea is **prediction**. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. We use the fortune-teller's crystal ball as our prediction icon.

Hierarchy of Memories

Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a **hierarchy of memories**, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. As we shall see in Chapter 5, caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

Dependability via Redundancy

Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axels allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the flat tire can be fixed, thereby restoring redundancy!)

1.3

Below Your Program

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

Figure 1.3 shows that these layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of **systems software** sitting between the hardware and applications software.

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:

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

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

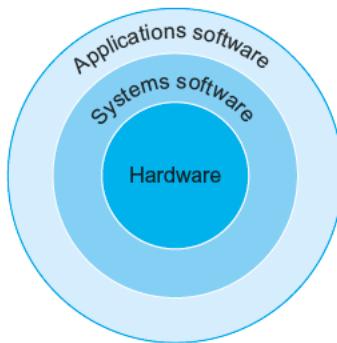


FIGURE 1.3 A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost. In complex applications, there are often multiple layers of application software as well. For example, a database system may run on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

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



systems software

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

operating system

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

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

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

From a High-Level Language to the Language of Hardware

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

1000110010100000

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

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

add A,B

and the assembler would translate this notation into

1000110010100000

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

Although a tremendous improvement, assembly language is still far from the notations a scientist might like to use to simulate fluid flow or that an accountant might use to balance the books. Assembly language requires the programmer to write one line for every instruction that the computer will follow, forcing the programmer to think like the computer.

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

instruction A command that computer hardware understands and obeys.

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

assembly language
A symbolic representation of machine instructions.

machine language
A binary representation of machine instructions.

The recognition that a program could be written to translate a more powerful language into computer instructions was one of the great breakthroughs in the early days of computing. Programmers today owe their productivity—and their sanity—to the creation of **high-level programming languages** and compilers that translate programs in such languages into instructions. Figure 1.4 shows the relationships among these programs and languages, which are more examples of the power of **abstraction**.



ABSTRACTION

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

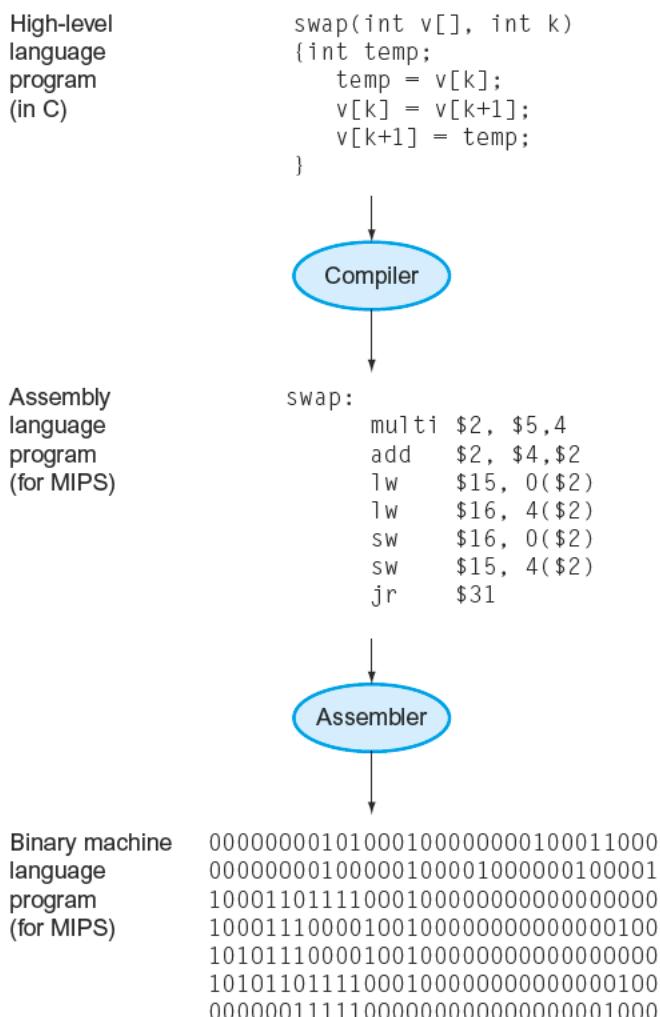


FIGURE 1.4 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages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

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

A + B

The compiler would compile it into this assembly language statement:

add A,B

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

High-level programming languages offer several important benefits. First, they allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols (see [Figure 1.4](#)). Moreover, they allow languages to be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on. There are also domain-specific languages for even narrower groups of users, such as those interested in simulation of fluids, for example.

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

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

1.4

Under the Covers

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

When we come to an important point in this book, a point so important that we hope you will remember it forever, we emphasize it by identifying it as a *Big Picture* item. We have about a dozen Big Pictures in this book, the first being the five components of a computer that perform the tasks of inputting, outputting, processing, and storing data.

Two key components of computers are **input devices**, such as the microphone, and **output devices**, such as the speaker. As the names suggest, input feeds the

input device

A mechanism through which the computer is fed information, such as a keyboard.

output device

A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.

computer, and output is the result of computation sent to the user. Some devices, such as wireless networks, provide both input and output to the computer.

Chapters 5 and 6 describe 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.

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

The BIG Picture

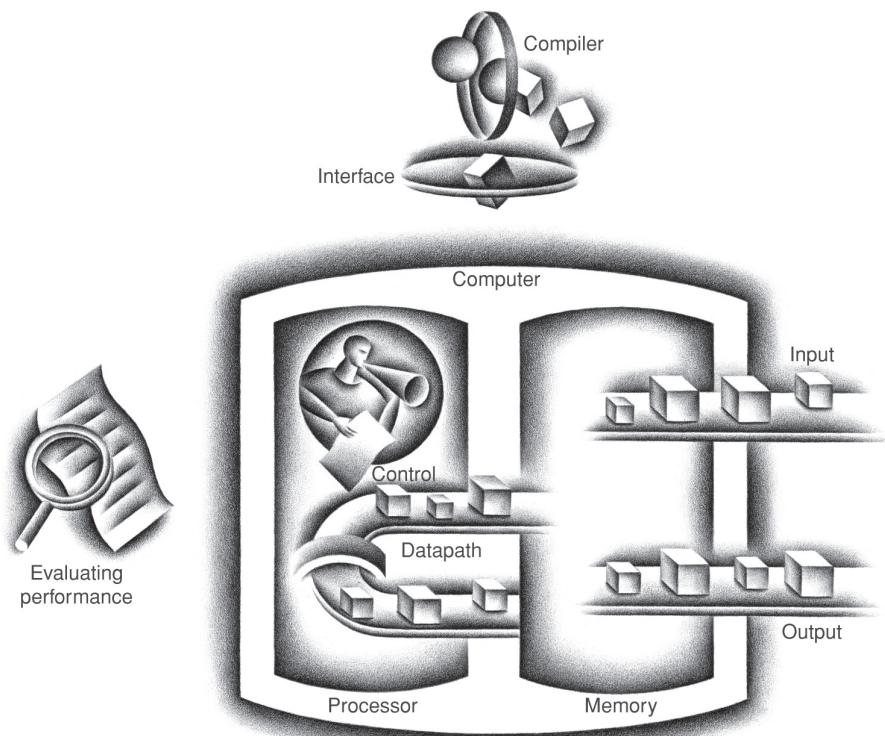


FIGURE 1.5 The organization of a computer, showing the five classic components. The processor gets instructions and data from memory. Input writes data to memory, and output reads data from memory. Control sends the signals that determine the operations of the datapath, memory, input, and output.

Through the Looking Glass

liquid crystal display

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

active matrix display

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

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

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

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

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

The image is composed of a matrix of picture elements, or **pixels**, which can be represented as a matrix of bits, called a *bit map*. Depending on the size of the screen and the resolution, the display matrix in a typical tablet ranges in size from 1024×768 to 2048×1536 . 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.

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

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.

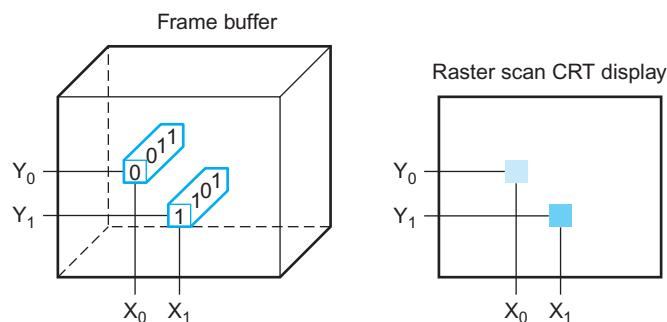


FIGURE 1.6 Each coordinate in the frame buffer on the left determines the shade of the corresponding coordinate for the raster scan CRT display on the right. Pixel (X_0, Y_0) contains the bit pattern 0011, which is a lighter shade on the screen than the bit pattern 1101 in pixel (X_1, Y_1) .

Touchscreen

While PCs also use LCD displays, the tablets and smartphones of the PostPC era have replaced the keyboard and mouse with touch sensitive displays, which has the wonderful user interface advantage of users pointing directly what they are interested in rather than indirectly with a mouse.

While there are a variety of ways to implement a touch screen, many tablets today use capacitive sensing. Since people are electrical conductors, if an insulator like glass is covered with a transparent conductor, touching distorts the electrostatic field of the screen, which results in a change in capacitance. This technology can allow multiple touches simultaneously, which allows gestures that can lead to attractive user interfaces.

Opening the Box

Figure 1.7 shows the contents of the Apple iPad 2 tablet computer. Unsurprisingly, of the five classic components of the computer, I/O dominates this reading device. The list of I/O devices includes a capacitive multitouch LCD display, front facing camera, rear facing camera, microphone, headphone jack, speakers, accelerometer, gyroscope, Wi-Fi network, and Bluetooth network. The datapath, control, and memory are a tiny portion of the components.

The small rectangles in Figure 1.8 contain the devices that drive our advancing technology, called **integrated circuits** and nicknamed **chips**. The A5 package seen in the middle of in Figure 1.8 contains two ARM processors that operate with a clock rate of 1 GHz. The *processor* is the active part of the computer, following the instructions of a program to the letter. It adds numbers, tests numbers, signals I/O devices to activate, and so on. Occasionally, people call the processor the **CPU**, for the more bureaucratic-sounding **central processor unit**.

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

The A5 package in Figure 1.8 also includes two memory chips, each with 2 gibibits of capacity, thereby supplying 512 MiB. The **memory** is where the programs are kept when they are running; it also contains the data needed by the running programs. The memory is built from DRAM chips. DRAM stands for **dynamic random access memory**. Multiple DRAMs are used together to contain the instructions and data of a program. In contrast to sequential access memories, such as magnetic tapes, the RAM portion of the term DRAM means that memory accesses take basically the same amount of time no matter what portion of the memory is read.

Descending into the depths of any component of the hardware reveals insights into the computer. Inside the processor is another type of memory—cache memory.

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

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.

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.

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

dynamic random access memory (DRAM) Memory built as an integrated circuit; it provides random access to any location. Access times are 50 nanoseconds and cost per gigabyte in 2012 was \$5 to \$10.



FIGURE 1.7 Components of the Apple iPad 2 A1395. The metal back of the iPad (with the reversed Apple logo in the middle) is in the center. At the top is the capacitive multitouch screen and LCD display. To the far right is the 3.8 V, 25 watt-hour, polymer battery, which consists of three Li-ion cell cases and offers 10 hours of battery life. To the far left is the metal frame that attaches the LCD to the back of the iPad. The small components surrounding the metal back in the center are what we think of as the computer; they are often L-shaped to fit compactly inside the case next to the battery. Figure 1.8 shows a close-up of the L-shaped board to the lower left of the metal case, which is the logic printed circuit board that contains the processor and the memory. The tiny rectangle below the logic board contains a chip that provides wireless communication: Wi-Fi, Bluetooth, and FM tuner. It fits into a small slot in the lower left corner of the logic board. Near the upper left corner of the case is another L-shaped component, which is a front-facing camera assembly that includes the camera, headphone jack, and microphone. Near the right upper corner of the case is the board containing the volume control and silent/screen rotation lock button along with a gyroscope and accelerometer. These last two chips combine to allow the iPad to recognize 6-axis motion. The tiny rectangle next to it is the rear-facing camera. Near the bottom right of the case is the L-shaped speaker assembly. The cable at the bottom is the connector between the logic board and the camera/volume control board. The board between the cable and the speaker assembly is the controller for the capacitive touchscreen. (Courtesy iFixit, www.ifixit.com)



FIGURE 1.8 The logic board of Apple iPad 2 in Figure 1.7. The photo highlights five integrated circuits. The large integrated circuit in the middle is the Apple A5 chip, which contains a dual ARM processor cores that run at 1 GHz as well as 512 MB of main memory inside the package. Figure 1.9 shows a photograph of the processor chip inside the A5 package. The similar sized chip to the left is the 32 GB flash memory chip for non-volatile storage. There is an empty space between the two chips where a second flash chip can be installed to double storage capacity of the iPad. The chips to the right of the A5 include power controller and I/O controller chips. (Courtesy iFixit, www.ifixit.com)

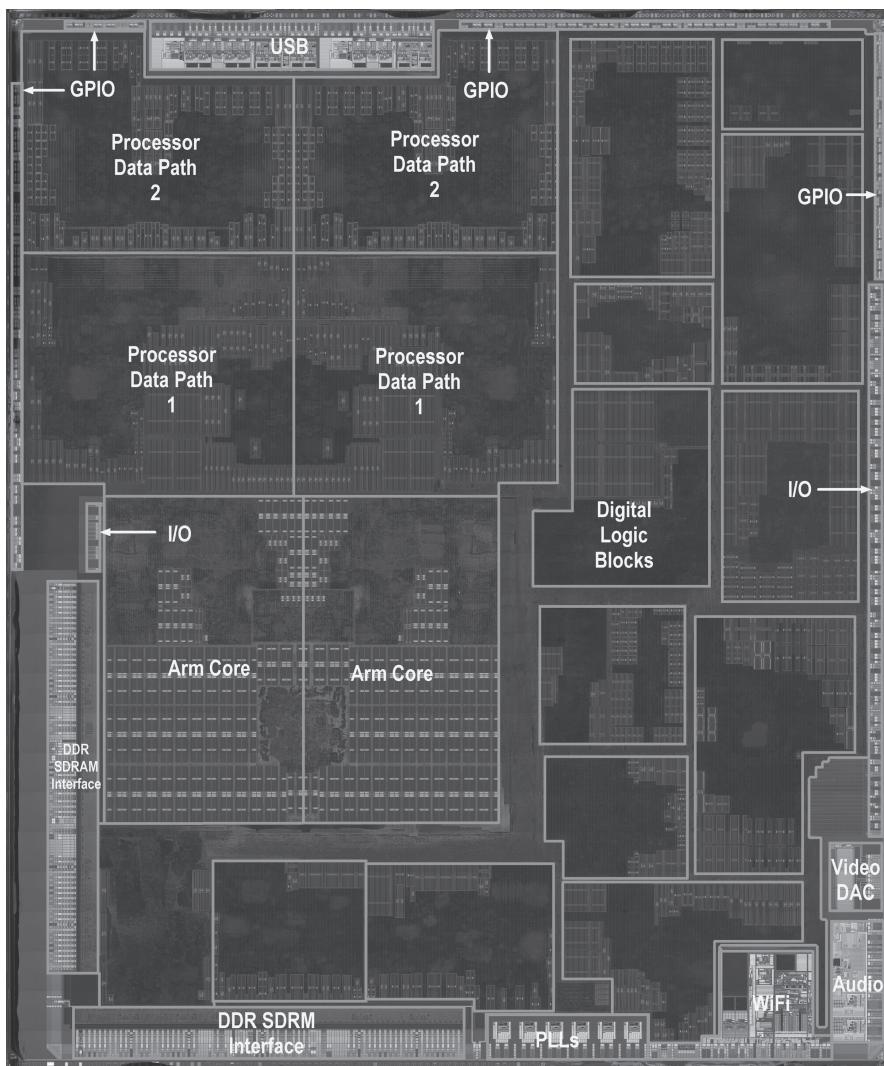


FIGURE 1.9 The processor integrated circuit inside the A5 package. The size of chip is 12.1 by 10.1 mm, and it was manufactured originally in a 45-nm process (see Section 1.5). It has two identical ARM processors or cores in the middle left of the chip and a PowerVR graphical processor unit (GPU) with four datapaths in the upper left quadrant. To the left and bottom side of the ARM cores are interfaces to main memory (DRAM). (Courtesy Chipworks, www.chipworks.com)

Cache memory consists of a small, fast memory that acts as a buffer for the DRAM memory. (The nontechnical definition of *cache* is a safe place for hiding things.) Cache is built using a different memory technology, **static random access memory (SRAM)**. SRAM is faster but less dense, and hence more expensive, than DRAM (see Chapter 5). SRAM and DRAM are two layers of the **memory hierarchy**.

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

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





ABSTRACTION

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

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

The BIG Picture

implementation
Hardware that obeys the architecture abstraction.

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

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

As mentioned above, one of the great ideas to improve design is abstraction. One of the most important **abstractions** is the interface between the hardware and the lowest-level software. Because of its importance, it is given a special name: the **instruction set architecture**, or simply **architecture**, of a computer. The instruction set architecture includes anything programmers need to know to make a binary machine language program work correctly, including instructions, I/O devices, and so on. Typically, the operating system will encapsulate the details of doing I/O, allocating memory, and other low-level system functions so that application programmers do not need to worry about such details. The combination of the basic instruction set and the operating system interface provided for application programmers is called the **application binary interface (ABI)**.

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

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

A Safe Place for Data

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

To distinguish between the volatile memory used to hold data and programs while they are running and this nonvolatile memory used to store data and programs between runs, the term **main memory** or **primary memory** is used for the former, and **secondary memory** for the latter. Secondary memory forms the next lower layer of the **memory hierarchy**. DRAMs have dominated main memory since 1975, but **magnetic disks** dominated secondary memory starting even earlier. Because of their size and form factor, personal Mobile Devices use **flash memory**, a nonvolatile semiconductor memory, instead of disks. Figure 1.8 shows the chip containing the flash memory of the iPad 2. While slower than DRAM, it is much cheaper than DRAM in addition to being nonvolatile. Although costing more per bit than disks, it is smaller, it comes in much smaller capacities, it is more rugged, and it is more power efficient than disks. Hence, flash memory is the standard secondary memory for PMDs. Alas, unlike disks and DRAM, flash memory bits wear out after 100,000 to 1,000,000 writes. Thus, file systems must keep track of the number of writes and have a strategy to avoid wearing out storage, such as by moving popular data. Chapter 5 describes disks and flash memory in more detail.

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 is connected to memory and I/O devices, networks interconnect whole computers, allowing computer users to extend the power of computing by including communication. Networks have become so popular that they are the backbone of current computer systems; a new personal mobile device or server without a network interface would be ridiculed. Networked computers have several major advantages:

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

Networks vary in length and performance, with the cost of communication increasing according to both the speed of communication and the distance that information travels. Perhaps the most popular type of network is *Ethernet*. It can be up to a kilometer long and transfer at up to 40 gigabits per second. Its length and speed make Ethernet useful to connect computers on the same floor of a building;



HIERARCHY

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

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of flash memory in PMDs and magnetic disks in servers.

magnetic disk Also called **hard disk**. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material. Because they are rotating mechanical devices, access times are about 5 to 20 milliseconds and cost per gigabyte in 2012 was \$0.05 to \$0.10.

flash memory A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks. Access times are about 5 to 50 microseconds and cost per gigabyte in 2012 was \$0.75 to \$1.00.

local area network

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

wide area network

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

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 web. They are typically based on optical fibers and are leased from telecommunication companies.

Networks have changed the face of computing in the last 30 years, both by becoming much more ubiquitous and by making dramatic increases in performance. In the 1970s, very few individuals had access to electronic mail, the Internet and web did not exist, and physically mailing magnetic tapes was the primary way to transfer large amounts of data between two locations. Local area networks were almost nonexistent, and the few existing wide area networks had limited capacity and restricted access.

As networking technology improved, it became much cheaper and had a much higher capacity. For example, the first standardized local area network technology, developed about 30 years ago, was a version of Ethernet that had a maximum capacity (also called bandwidth) of 10 million bits per second, typically shared by tens of, if not a hundred, computers. Today, local area network technology offers a capacity of from 1 to 40 gigabits per second, usually shared by at most a few computers. Optical communications technology has allowed similar growth in the capacity of wide area networks, from hundreds of kilobits to gigabits and from hundreds of computers connected to a worldwide network to millions of computers connected. This combination of dramatic rise in deployment of networking combined with increases in capacity have made network technology central to the information revolution of the last 30 years.

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

Check Yourself

- Semiconductor DRAM memory, flash memory, and disk storage differ significantly. For each technology, list its volatility, approximate relative access time, and approximate relative cost compared to DRAM.

1.5**Technologies for Building Processors and Memory**

Processors and memory have improved at an incredible rate, because computer designers have long embraced the latest in electronic technology to try to win the race to design a better computer. [Figure 1.10](#) shows the technologies that have

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
2013	Ultra large-scale integrated circuit	250,000,000,000

FIGURE 1.10 Relative performance per unit cost of technologies used in computers over time. Source: Computer Museum, Boston, with 2013 extrapolated by the authors. See  [Section 1.12](#).

been used over time, with an estimate of the relative performance per unit cost for each technology. 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. When Gordon Moore predicted the continuous doubling of resources, he was predicting the growth rate of the number of transistors per chip. To describe the tremendous increase in the number of transistors from hundreds to millions, the adjective *very large scale* is added to the term, creating the abbreviation *VLSI*, for **very large-scale integrated circuit**.

This rate of increasing integration has been remarkably stable. Figure 1.11 shows the growth in DRAM capacity since 1977. For decades, the industry has consistently quadrupled capacity every 3 years, resulting in an increase in excess of 16,000 times!

To understand how manufacture integrated circuits, we 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)

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

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

silicon A natural element that is a semiconductor.

semiconductor A substance that does not conduct electricity well.

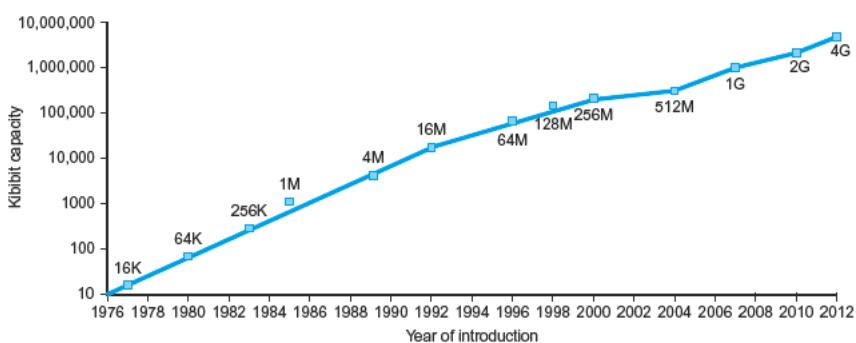


FIGURE 1.11 Growth of capacity per DRAM chip over time. The y-axis is measured in kibits (2^{10} bits). The DRAM industry quadrupled capacity almost every three years, a 60% increase per year, for 20 years. In recent years, the rate has slowed down and is somewhat closer to doubling every two years to three years.

- 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 crystal ingot

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

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

The manufacturing process for integrated circuits is critical to the cost of the chips and hence important to computer designers. Figure 1.12 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 inches thick. These wafers then go through a series of processing steps, during which patterns of chemicals are placed 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.

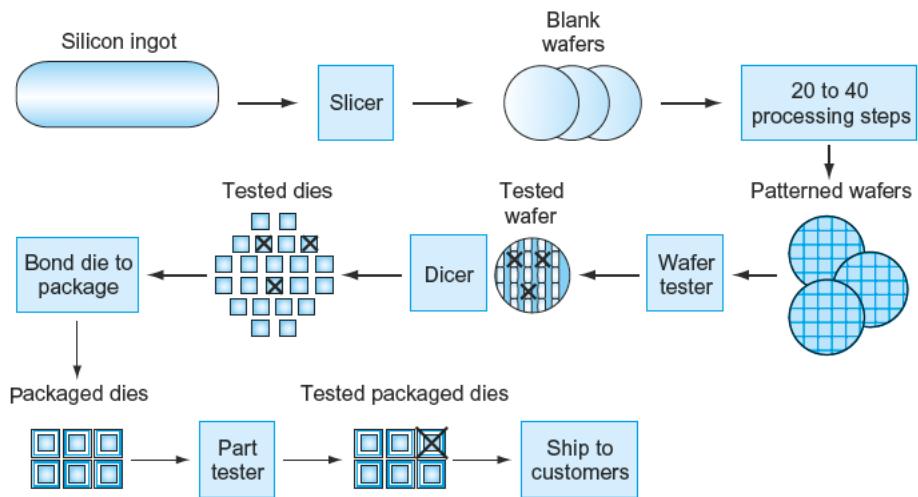


FIGURE 1.12 The chip manufacturing process. After being sliced from the silicon ingot, blank wafers are put through 20 to 40 steps to create patterned wafers (see Figure 1.13). These patterned wafers are then tested with a wafer tester, and a map of the good parts is made. Then, the wafers are diced into dies (see Figure 1.9). In this figure, one wafer produced 20 dies, of which 17 passed testing. (X means the die is bad.) The yield of good dies in this case was 17/20, or 85%. These good dies are then bonded into packages and tested one more time before shipping the packaged parts to customers. One bad packaged part was found in this final test.

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

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. The simplest way to cope with imperfection is to place many independent components on a single wafer. The patterned wafer is then chopped up, or *diced*, into these components,



FIGURE 1.13 A 12-inch (300 mm) wafer of Intel Core i7 (Courtesy Intel). The number of dies on this 300 mm (12 inch) wafer at 100% yield is 280, each 20.7 by 10.5 mm. The several dozen partially rounded chips at the boundaries of the wafer are useless; they are included because it's easier to create the masks used to pattern the silicon. This die uses a 32-nanometer technology, which means that the smallest features are approximately 32 nm in size, although they are typically somewhat smaller than the actual feature size, which refers to the size of the transistors as "drawn" versus the final manufactured size.

called **dies** and more informally known as **chips**. Figure 1.13 shows a photograph of a wafer containing microprocessors before they have been diced; earlier, Figure 1.9 shows an individual microprocessor die.

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

The cost of an integrated circuit rises quickly as the die size increases, due both to the lower yield and the smaller number of dies that fit on a wafer. To reduce the cost, using the next generation process shrinks a large die as it uses smaller sizes for both transistors and wires. This improves the yield and the die count per wafer. A 32-nanometer (nm) process was typical in 2012, which means essentially that the smallest feature size on the die is 32 nm.

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

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

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.

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

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

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

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

The first equation is straightforward to derive. The second is an approximation, since it does not subtract the area near the border of the round wafer that cannot accommodate the rectangular dies (see [Figure 1.13](#)). The final equation is based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.

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

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.

1.6

Performance

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

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

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

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

Defining Performance

When we say one computer has better performance than another, what do we mean? Although this question might seem simple, an analogy with passenger airplanes shows how subtle the question of performance can be. [Figure 1.14](#) lists 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 was the Concorde (retired from service in 2003), the plane with the longest range is the DC-8, and the plane with the largest capacity is the 747.

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

FIGURE 1.14 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).

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, however, the 747 would clearly be the fastest, as the last column of the figure shows. Similarly, we can define computer performance in several different ways.

If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred

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

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

to as **execution time**. Datacenter managers are often interested in increasing **throughput** or **bandwidth**—the total amount of work done in a given time. Hence, in most cases, we will need different performance metrics as well as different sets of applications to benchmark personal mobile devices, which are more focused on response time, versus servers, which are more focused on throughput.

EXAMPLE

ANSWER

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 web

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

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

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

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

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

$$\begin{aligned} \text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X \end{aligned}$$

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 as fast as Y, then the execution time on Y is n times as long as it is on X:

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

Relative Performance

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?

EXAMPLE

We know that A is n times as fast as B if

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

ANSWER

Thus the performance ratio is

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

and A is therefore 1.5 times as fast as 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 *as fast as* when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*, we usually say “improve performance” or “improve execution time” when we mean “increase performance” and “decrease execution time.”

Measuring Performance

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

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 over which the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to do accurately, because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems.

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

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

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

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

Understanding Program Performance

Different applications are sensitive to different aspects of the performance of a computer system. Many applications, especially those running on servers, depend as much on I/O performance, which, in turn, relies on both hardware and software. Total elapsed time measured by a wall clock is the measurement of interest. In

some application environments, the user may care about throughput, response time, or a complex combination of the two (e.g., maximum throughput with a worst-case response time). To improve the performance of a program, one must have a clear definition of what performance metric matters and then proceed to look for performance bottlenecks by measuring program execution and looking for the likely bottlenecks. In the following chapters, we will describe how to search for bottlenecks and improve performance in various parts of the system.

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

1. Suppose we know that an application that uses both personal mobile devices and the Cloud is limited by network performance. For the following changes, state whether only the throughput improves, both response time and throughput improve, or neither improves.
 - a. An extra network channel is added between the PMD and the Cloud, increasing the total network throughput and reducing the delay to obtain network access (since there are now two channels).
 - b. The networking software is improved, thereby reducing the network communication delay, but not increasing throughput.
 - c. More memory is added to the computer.
2. Computer C's performance is 4 times as fast as the performance of computer B, which runs a given application in 28 seconds. How long will computer C take to run that application?

CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as experienced by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU

clock cycle Also called **tick, clock tick, clock period, clock, or cycle**.

The time for one clock period, usually of the processor clock, which runs at a constant rate.

clock period The length of each clock cycle.

Check Yourself

execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock cycle time}}$$

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

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

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

EXAMPLE

Improving Performance

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

ANSWER

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

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

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

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

CPU time for B can be found using this equation:

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

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

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

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

Instruction Performance

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

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

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

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

Using the Performance Equation

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

EXAMPLE

ANSWER

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

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

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

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time} \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

Likewise, for B:

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

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

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

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

The Classic CPU Performance Equation

instruction count The number of instructions executed by the program.

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

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

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

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

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

Comparing Code Segments

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

EXAMPLE

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

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

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

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

ANSWER

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

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

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

This yields

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

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

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

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

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

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

The BIG Picture

Figure 1.15 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds per program:

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

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

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

FIGURE 1.15 The basic components of performance and how each is measured.

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

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

The above example shows the danger of using only one factor (instruction count) to assess performance. When comparing two computers, you must look at all three components, which combine to form execution time. If some of the factors are identical, like the clock rate in the above example, performance can be determined by comparing all the nonidentical factors. Since CPI varies by **instruction mix**, both instruction count and CPI must be compared, even if clock rates are identical. Several exercises at the end of this chapter ask you to evaluate a series of computer and compiler enhancements that affect clock rate, CPI, and instruction count. In [Section 1.10](#), we'll examine a common performance measurement that does not incorporate all the terms and can thus be misleading.

instruction mix

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

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

Understanding Program Performance

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

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

Elaboration: Although clock cycle time has traditionally been fixed, to save energy or temporarily boost performance, today's processors can vary their clock rates, so we would need to use the average clock rate for a program. For example, the Intel Core i7 will temporarily increase clock rate by about 10% until the chip gets too warm. Intel calls this *Turbo mode*.

Check Yourself

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

- $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$
- $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$
- $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

1.7

The Power Wall

Figure 1.16 shows the increase in clock rate and power of eight generations of Intel microprocessors over 30 years. Both clock rate and power increased rapidly for decades, and then flattened off recently. The reason they grew together is that they are correlated, and the reason for their recent slowing is that we have run into the practical power limit for cooling commodity microprocessors.

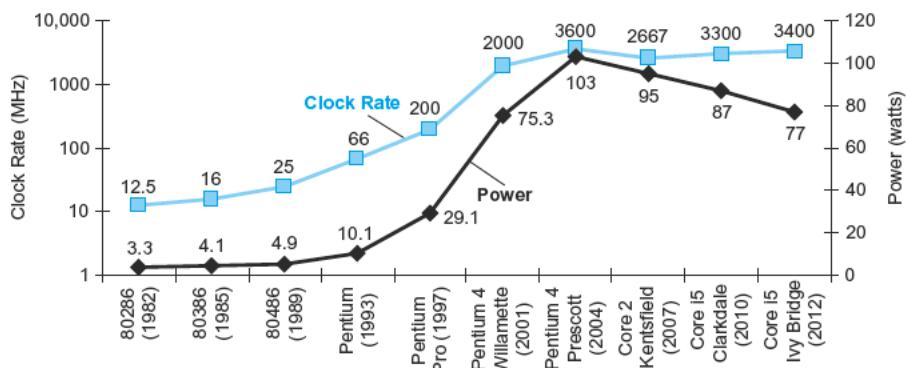


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

Although power provides a limit to what we can cool, in the PostPC Era the really critical resource is energy. Battery life can trump performance in the personal mobile device, and the architects of warehouse scale computers try to reduce the costs of powering and cooling 100,000 servers as the costs are high at this scale. Just as measuring time in seconds is a safer measure of program performance than a rate like MIPS (see Section 1.10), the energy metric joules is a better measure than a power rate like watts, which is just joules/second.

The dominant technology for integrated circuits is called CMOS (complementary metal oxide semiconductor). For CMOS, the primary source of energy consumption is so-called dynamic energy—that is, energy that is consumed when transistors switch states from 0 to 1 and vice versa. The dynamic energy depends on the capacitive loading of each transistor and the voltage applied:

$$\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$$

This equation is the energy of a pulse during the logic transition of $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$. The energy of a single transition is then

$$\text{Energy} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

$$\text{Power} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

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

With regard to [Figure 1.16](#), how could clock rates grow by a factor of 1000 while power grew by only a factor of 30? Energy and thus power can be reduced by lowering the voltage, which occurred with each new generation of technology, and power is a function of the voltage squared. Typically, the voltage was reduced about 15% per generation. In 20 years, voltages have gone from 5 V to 1 V, which is why the increase in power is only 30 times.

Relative Power

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

EXAMPLE

ANSWER

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

Thus the power ratio is

$$0.85^4 = 0.52$$

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

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

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

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

Elaboration: Although dynamic energy is the primary source of energy consumption in CMOS, static energy consumption occurs because of leakage current that flows even when a transistor is off. In servers, leakage is typically responsible for 40% of the energy 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 are being deployed to control leakage, but it's hard to lower voltage further.

Elaboration: Power is a challenge for integrated circuits 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 chip interconnect are used solely for power and ground distribution to portions of the chip. Second, power is dissipated as heat and must be removed. Server chips can burn more than 100 watts, and cooling the chip and the surrounding system is a major expense in Warehouse Scale Computers (see Chapter 6).

1.8

The Sea Change: The Switch from Uniprocessors to Multiprocessors

The power limit has forced a dramatic change in the design of microprocessors. Figure 1.17 shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to a factor of 1.2 per year.

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

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

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

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

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

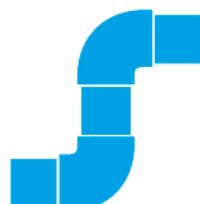


PARALLELISM

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

Forcing programmers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel had been the “third rail” of computer architecture, for companies in the past that depended on such a change in behavior failed (see [Section 6.15](#)). From this historical perspective, it’s startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

Hardware/ Software Interface



PIPELINING

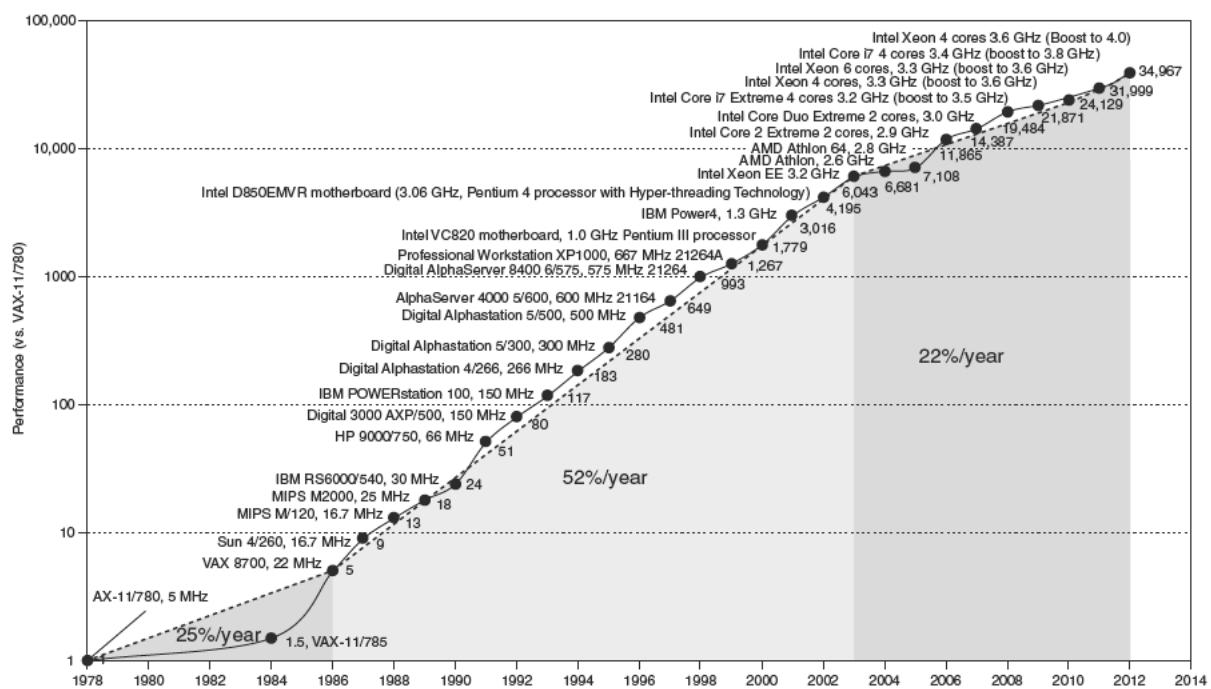


FIGURE 1.17 Growth In processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECInt benchmarks (see Section 1.10). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. The higher annual performance improvement of 52% since the mid-1980s meant performance was about a factor of seven higher in 2002 than it would have been had it stayed at 25%. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 22% per year.

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

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism.

As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *schedule* the sub-tasks. If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must *balance the*

load evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all of the other parts were completed. Thus, care must be taken to *reduce communication and synchronization overhead*. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

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

- *Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization.* Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Subword Parallelism.* Perhaps the simplest form of parallelism to build involves computing on elements in parallel, such as when multiplying two vectors. Subword parallelism takes advantage of the resources supplied by **Moore's Law** to provider wider arithmetic units that can operate on many operands simultaneously.
- *Chapter 4, Section 4.10: Parallelism via Instructions.* Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism, initially via **pipelining**. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions simultaneously and guessing on the outcomes of decisions, and executing instructions speculatively using **prediction**.
- *Chapter 5, Section 5.10: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.
- *Chapter 5, Section 5.11: Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks.* This section describes how using many disks in conjunction can offer much higher throughput, which was the original inspiration of *Redundant Arrays of Inexpensive Disks* (RAID). The real popularity of RAID proved to be to the much greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and dependability between the different RAID levels.





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

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



benchmark A program selected for use in comparing computer performance.

In addition to these sections, there is a full chapter on parallel processing. Chapter 6 goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors; and, finally, describes and evaluates four examples of multicore microprocessors using this model.

As mentioned above, Chapters 3 to 6 use matrix vector multiply as a running example to show how each type of parallelism can significantly increase performance.

➊ **Appendix C** describes an increasingly popular hardware component that is included with desktop computers, the *graphics processing unit* (GPU). Invented to accelerate graphics, GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs rely on **parallelism**.

➋ **Appendix C** describes the NVIDIA GPU and highlights parts of its parallel programming environment.

1.9

Real Stuff: Benchmarking the Intel Core i7

Each chapter has a section entitled “Real Stuff” that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying modern computers. For this first “Real Stuff” section, we look at how integrated circuits are manufactured and how performance and power are measured, with the Intel Core i7 as the example.

SPEC CPU Benchmark

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. The set of programs run would form a **workload**. To evaluate two computer systems, a user would simply compare the execution time of the workload on the two computers. Most users, however, are not in this situation. Instead, they must rely on other methods that measure the performance of a candidate computer, hoping that the methods will reflect how well the computer will perform with the user’s workload. This alternative is usually followed by evaluating the computer using a set of **benchmarks**—programs specifically chosen to measure performance. The benchmarks form a workload that the user hopes will predict the performance of the actual workload. As we noted above, to make the **common case fast**, you first need to know accurately which case is common, so benchmarks play a critical role in computer architecture.

SPEC (*System Performance Evaluation Cooperative*) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems. In 1989, SPEC originally created a benchmark

Description	Name	Instruction Count x 10 ⁹	CPI	Clock cycle time (seconds x 10 ⁻⁹)	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	—	—	—	—	—	—	25.7

FIGURE 1.18 SPECINTC2006 benchmarks running on a 2.66GHz Intel Core i7 920. As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number quoted as SPECINTC2006 is the geometric mean of the SPECratios.

set focusing on processor performance (now called SPEC89), which has evolved through five generations. The latest is SPEC CPU2006, which consists of a set of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006). The integer benchmarks vary from part of a C compiler to a chess program to a quantum computer simulation. The floating-point benchmarks include structured grid codes for finite element modeling, particle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics.

Figure 1.18 describes the SPEC integer benchmarks and their execution time on the Intel Core i7 and shows the factors that explain execution time: instruction count, CPI, and clock cycle time. Note that CPI varies by more than a factor of 5.

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

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

The formula for the geometric mean is

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

where $\text{Execution time ratio}_i$ is the execution time, normalized to the reference computer, for the i th program of a total of n in the workload, and

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

SPEC Power Benchmark

Given the increasing importance of energy and power, SPEC added a benchmark to measure power. It reports power consumption of servers at different workload levels, divided into 10% increments, over a period of time. [Figure 1.19](#) shows the results for a server using Intel Nehalem processors similar to the above.

Target Load %	Performance (ssj_ops)	Average Power (watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1922
$\sum \text{ssj_ops} / \sum \text{power} =$		2490

FIGURE 1.19 SPECpower_ssj2008 running on a dual socket 2.66 GHz Intel Xeon X5650 with 16 GB of DRAM and one 100 GB SSD disk.

SPECpower started with another SPEC benchmark for Java business applications (SPECJBB2005), which exercises the processors, caches, and main memory as well as the Java virtual machine, compiler, garbage collector, and pieces of the operating system. Performance is measured in throughput, and the units are business operations per second. Once again, to simplify the marketing of computers, SPEC

boils these numbers down to a single number, called “overall ssj_ops per watt.” The formula for this single summarizing metric is

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

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

1.10 Fallacies and Pitfalls

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 them *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 only true in a limited context. The purpose of these sections is to help you avoid making these mistakes in the computers you may design or use. Cost/performance fallacies and pitfalls have ensnared many a computer architect, including us. Accordingly, this section suffers no shortage of relevant examples. We start with a pitfall that traps many designers and reveals an important relationship in computer design.

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

The great idea of making the **common case fast** has a demoralizing corollary that has plagued designers of both hardware and software. It reminds us that the opportunity for improvement is affected by how much time the event consumes.

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

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

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

For this problem:

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

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

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



COMMON CASE FAST

Amdahl's Law

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

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

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

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

That is, there is *no amount* by which we can enhance-multiply to achieve a fivefold increase in performance, if multiply accounts for only 80% of the workload. The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used. In everyday life this concept also yields what we call the law of diminishing returns.

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

Amdahl's Law is also used to argue for practical limits to the number of parallel processors. We examine this argument in the Fallacies and Pitfalls section of Chapter 6.

Fallacy: Computers at low utilization use little power.

Power efficiency matters at low utilizations because server workloads vary. Utilization of servers in Google's warehouse scale computer, for example, is between 10% and 50% most of the time and at 100% less than 1% of the time. Even given five years to learn how to run the SPECpower benchmark well, the specially configured computer with the best results in 2012 still uses 33% of the peak power at 10% of the load. Systems in the field that are not configured for the SPECpower benchmark are surely worse.

Since servers' workloads vary but use a large fraction of peak power, Luiz Barroso and Urs Hölzle [2007] argue that we should redesign hardware to achieve "energy-proportional computing." If future servers used, say, 10% of peak power at 10% workload, we could reduce the electricity bill of datacenters and become good corporate citizens in an era of increasing concern about CO₂ emissions.

Fallacy: Designing for performance and designing for energy efficiency are unrelated goals.

Since energy is power over time, it is often the case that hardware or software optimizations that take less time save energy overall even if the optimization takes a bit more energy when it is used. One reason is that all of the rest of the computer is consuming energy while the program is running, so even if the optimized portion uses a little more energy, the reduced time can save the energy of the whole system.

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

We have already warned about the danger of predicting performance based on simply one of clock rate, instruction count, or CPI. Another common mistake

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

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

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

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

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

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

The CPI varied by a factor of 5 for SPEC CPU2006 on an Intel Core i7 computer in [Figure 1.18](#), so MIPS does as well. Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance!

Consider the following performance measurements for a program:

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

million instructions per second (MIPS)

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

Check Yourself

- Which computer has the higher MIPS rating?
- Which computer is faster?

1.11

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 1½ tons.

*Popular Mechanics,
March 1949*



ABSTRACTION

The **BIG**
Picture

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

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

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

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

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

The key hardware technology for modern processors is silicon. Equal in importance to an understanding of integrated circuit technology is an understanding of the expected rates of technological change, as predicted by **Moore's Law**. While silicon fuels the rapid advance of hardware, new ideas in the organization of computers have improved price/performance. Two of the key ideas are exploiting parallelism in the program, typically today via multiple processors, and exploiting locality of accesses to a **memory hierarchy**, typically via caches.

Energy efficiency has replaced die area as the most critical resource of microprocessor design. Conserving power while trying to increase performance has forced the hardware industry to switch to multicore microprocessors, thereby forcing the software industry to switch to programming parallel hardware. **Parallelism** is now required for performance.

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

Road Map for This Book

At the bottom of these abstractions are the five classic components of a computer: datapath, control, memory, input, and output (refer to Figure 1.5). These five components also serve as the framework for the rest of the chapters in this book:

- **Datapath:** Chapter 3, Chapter 4, Chapter 6, and [Appendix C](#)
- **Control:** Chapter 4, Chapter 6, and [Appendix C](#)
- **Memory:** Chapter 5
- **Input:** Chapters 5 and 6
- **Output:** Chapters 5 and 6

As mentioned above, Chapter 4 describes how processors exploit implicit parallelism, Chapter 6 describes the explicitly parallel multicore microprocessors that are at the heart of the parallel revolution, and [Appendix C](#) describes the highly parallel graphics processor chip. Chapter 5 describes how a memory hierarchy exploits locality. Chapter 2 describes instruction sets—the interface between compilers and the computer—and emphasizes the role of compilers and programming languages in using the features of the instruction set. Appendix A provides a reference for the instruction set of Chapter 2. Chapter 3 describes how computers handle arithmetic data. Appendix B introduces logic design.





Historical Perspective and Further Reading

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

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

The historical perspective for this chapter provides a background for some of the key ideas presented in this opening chapter. Its purpose is to give you the human story behind the technological advances and to place achievements in their historical context. By understanding the past, you may be better able to understand the forces that will shape computing in the future. Each Historical Perspective section online ends with suggestions for further reading, which are also collected separately online under the section “[Further Reading](#).” The rest of [Section 1.12](#) is found online.

1.13

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, <\$1.4> means you should have read Section 1.4, Under the Covers, to help you solve this exercise.

1.1 [2] <\$1.1> Aside from the smart cell phones used by a billion people, list and describe four other types of computers.

1.2 [5] <\$1.2> The eight great ideas in computer architecture are similar to ideas from other fields. Match the eight ideas from computer architecture, “Design for Moore’s Law”, “Use Abstraction to Simplify Design”, “Make the Common Case Fast”, “Performance via Parallelism”, “Performance via Pipelining”, “Performance via Prediction”, “Hierarchy of Memories”, and “Dependability via Redundancy” to the following ideas from other fields:

- a. Assembly lines in automobile manufacturing
- b. Suspension bridge cables
- c. Aircraft and marine navigation systems that incorporate wind information
- d. Express elevators in buildings

- e. Library reserve desk
- f. Increasing the gate area on a CMOS transistor to decrease its switching time
- g. Adding electromagnetic aircraft catapults (which are electrically-powered as opposed to current steam-powered models), allowed by the increased power generation offered by the new reactor technology
- h. Building self-driving cars whose control systems partially rely on existing sensor systems already installed into the base vehicle, such as lane departure systems and smart cruise control systems

1.3 [2] <§1.3> Describe the steps that transform a program written in a high-level language such as C into a representation that is directly executed by a computer processor.

1.4 [2] <§1.4> Assume a color display using 8 bits for each of the primary colors (red, green, blue) per pixel and a frame size of 1280×1024 .

- a. What is the minimum size in bytes of the frame buffer to store a frame?
- b. How long would it take, at a minimum, for the frame to be sent over a 100 Mbit/s network?

1.5 [4] <§1.6> Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.

- a. Which processor has the highest performance expressed in instructions per second?
- b. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions.
- c. We are trying to reduce the execution time by 30% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time reduction?

1.6 [20] <§1.6> Consider two different implementations of the same instruction set architecture. The instructions can be divided into four classes according to their CPI (class A, B, C, and D). P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3, and P2 with a clock rate of 3 GHz and CPIs of 2, 2, 2, and 2.

Given a program with a dynamic instruction count of $1.0E6$ instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which implementation is faster?

- a. What is the global CPI for each implementation?
- b. Find the clock cycles required in both cases.

1.7 [15] <§1.6> Compilers can have a profound impact on the performance of an application. Assume that for a program, compiler A results in a dynamic instruction count of 1.0E9 and has an execution time of 1.1 s, while compiler B results in a dynamic instruction count of 1.2E9 and an execution time of 1.5 s.

- a. Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.
- b. Assume the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler A's code versus the clock of the processor running compiler B's code?
- c. A new compiler is developed that uses only 6.0E8 instructions and has an average CPI of 1.1. What is the speedup of using this new compiler versus using compiler A or B on the original processor?

1.8 The Pentium 4 Prescott processor, released in 2004, had a clock rate of 3.6 GHz and voltage of 1.25 V. Assume that, on average, it consumed 10 W of static power and 90 W of dynamic power.

The Core i5 Ivy Bridge, released in 2012, had a clock rate of 3.4 GHz and voltage of 0.9 V. Assume that, on average, it consumed 30 W of static power and 40 W of dynamic power.

1.8.1 [5] <§1.7> For each processor find the average capacitive loads.

1.8.2 [5] <§1.7> Find the percentage of the total dissipated power comprised by static power and the ratio of static power to dynamic power for each technology.

1.8.3 [15] <§1.7> If the total dissipated power is to be reduced by 10%, how much should the voltage be reduced to maintain the same leakage current? Note: power is defined as the product of voltage and current.

1.9 Assume for arithmetic, load/store, and branch instructions, a processor has CPIs of 1, 12, and 5, respectively. Also assume that on a single processor a program requires the execution of 2.56E9 arithmetic instructions, 1.28E9 load/store instructions, and 256 million branch instructions. Assume that each processor has a 2 GHz clock frequency.

Assume that, as the program is parallelized to run over multiple cores, the number of arithmetic and load/store instructions per processor is divided by $0.7 \times p$ (where p is the number of processors) but the number of branch instructions per processor remains the same.

1.9.1 [5] <§1.7> Find the total execution time for this program on 1, 2, 4, and 8 processors, and show the relative speedup of the 2, 4, and 8 processor result relative to the single processor result.

1.9.2 [10] <§§1.6, 1.8> If the CPI of the arithmetic instructions was doubled, what would the impact be on the execution time of the program on 1, 2, 4, or 8 processors?

1.9.3 [10] <§§1.6, 1.8> To what should the CPI of load/store instructions be reduced in order for a single processor to match the performance of four processors using the original CPI values?

1.10 Assume a 15 cm diameter wafer has a cost of 12, contains 84 dies, and has 0.020 defects/cm². Assume a 20 cm diameter wafer has a cost of 15, contains 100 dies, and has 0.031 defects/cm².

1.10.1 [10] <\$1.5> Find the yield for both wafers.

1.10.2 [5] <\$1.5> Find the cost per die for both wafers.

1.10.3 [5] <\$1.5> If the number of dies per wafer is increased by 10% and the defects per area unit increases by 15%, find the die area and yield.

1.10.4 [5] <\$1.5> Assume a fabrication process improves the yield from 0.92 to 0.95. Find the defects per area unit for each version of the technology given a die area of 200 mm².

1.11 The results of the SPEC CPU2006 bzip2 benchmark running on an AMD Barcelona has an instruction count of 2.389E12, an execution time of 750 s, and a reference time of 9650 s.

1.11.1 [5] <§§1.6, 1.9> Find the CPI if the clock cycle time is 0.333 ns.

1.11.2 [5] <\$1.9> Find the SPECratio.

1.11.3 [5] <§§1.6, 1.9> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% without affecting the CPI.

1.11.4 [5] <§§1.6, 1.9> Find the increase in CPU time if the number of instructions of the benchmark is increased by 10% and the CPI is increased by 5%.

1.11.5 [5] <§§1.6, 1.9> Find the change in the SPECratio for this change.

1.11.6 [10] <\$1.6> Suppose that we are developing a new version of the AMD Barcelona processor with a 4 GHz clock rate. We have added some additional instructions to the instruction set in such a way that the number of instructions has been reduced by 15%. The execution time is reduced to 700 s and the new SPECratio is 13.7. Find the new CPI.

1.11.7 [10] <\$1.6> This CPI value is larger than obtained in 1.11.1 as the clock rate was increased from 3 GHz to 4 GHz. Determine whether the increase in the CPI is similar to that of the clock rate. If they are dissimilar, why?

1.11.8 [5] <\$1.6> By how much has the CPU time been reduced?

1.11.9 [10] <§1.6> For a second benchmark, libquantum, assume an execution time of 960 ns, CPI of 1.61, and clock rate of 3 GHz. If the execution time is reduced by an additional 10% without affecting the CPI and with a clock rate of 4 GHz, determine the number of instructions.

1.11.10 [10] <§1.6> Determine the clock rate required to give a further 10% reduction in CPU time while maintaining the number of instructions and with the CPI unchanged.

1.11.11 [10] <§1.6> Determine the clock rate if the CPI is reduced by 15% and the CPU time by 20% while the number of instructions is unchanged.

1.12 Section 1.10 cites as a pitfall the utilization of a subset of the performance equation as a performance metric. To illustrate this, consider the following two processors. P1 has a clock rate of 4 GHz, average CPI of 0.9, and requires the execution of 5.0E9 instructions. P2 has a clock rate of 3 GHz, an average CPI of 0.75, and requires the execution of 1.0E9 instructions.

1.12.1 [5] <§§1.6, 1.10> One usual fallacy is to consider the computer with the largest clock rate as having the largest performance. Check if this is true for P1 and P2.

1.12.2 [10] <§§1.6, 1.10> Another fallacy is to consider that the processor executing the largest number of instructions will need a larger CPU time. Considering that processor P1 is executing a sequence of 1.0E9 instructions and that the CPI of processors P1 and P2 do not change, determine the number of instructions that P2 can execute in the same time that P1 needs to execute 1.0E9 instructions.

1.12.3 [10] <§§1.6, 1.10> A common fallacy is to use MIPS (millions of instructions per second) to compare the performance of two different processors, and consider that the processor with the largest MIPS has the largest performance. Check if this is true for P1 and P2.

1.12.4 [10] <§1.10> Another common performance figure is MFLOPS (millions of floating-point operations per second), defined as

$$\text{MFLOPS} = \text{No. FP operations} / (\text{execution time} \times 1\text{E}6)$$

but this figure has the same problems as MIPS. Assume that 40% of the instructions executed on both P1 and P2 are floating-point instructions. Find the MFLOPS figures for the programs.

1.13 Another pitfall cited in Section 1.10 is expecting to improve the overall performance of a computer by improving only one aspect of the computer. Consider a computer running a program that requires 250 s, with 70 s spent executing FP instructions, 85 s executed L/S instructions, and 40 s spent executing branch instructions.

1.13.1 [5] <§1.10> By how much is the total time reduced if the time for FP operations is reduced by 20%?

1.13.2 [5] <§1.10> By how much is the time for INT operations reduced if the total time is reduced by 20%?

1.13.3 [5] <§1.10> Can the total time can be reduced by 20% by reducing only the time for branch instructions?

1.14 Assume a program requires the execution of 50×106 FP instructions, 110×106 INT instructions, 80×106 L/S instructions, and 16×106 branch instructions. The CPI for each type of instruction is 1, 1, 4, and 2, respectively. Assume that the processor has a 2 GHz clock rate.

1.14.1 [10] <§1.10> By how much must we improve the CPI of FP instructions if we want the program to run two times faster?

1.14.2 [10] <§1.10> By how much must we improve the CPI of L/S instructions if we want the program to run two times faster?

1.14.3 [5] <§1.10> By how much is the execution time of the program improved if the CPI of INT and FP instructions is reduced by 40% and the CPI of L/S and Branch is reduced by 30%?

1.15 [5] <§1.8> When a program is adapted to run on multiple processors in a multiprocessor system, the execution time on each processor is comprised of computing time and the overhead time required for locked critical sections and/or to send data from one processor to another.

Assume a program requires $t = 100$ s of execution time on one processor. When run p processors, each processor requires t/p s, as well as an additional 4 s of overhead, irrespective of the number of processors. Compute the per-processor execution time for 2, 4, 8, 16, 32, 64, and 128 processors. For each case, list the corresponding speedup relative to a single processor and the ratio between actual speedup versus ideal speedup (speedup if there was no overhead).

§1.1, page 10: Discussion questions: many answers are acceptable.

§1.4, page 24: DRAM memory: volatile, short access time of 50 to 70 nanoseconds, and cost per GB is \$5 to \$10. Disk memory: nonvolatile, access times are 100,000 to 400,000 times slower than DRAM, and cost per GB is 100 times cheaper than DRAM. Flash memory: nonvolatile, access times are 100 to 1000 times slower than DRAM, and cost per GB is 7 to 10 times cheaper than DRAM.

§1.5, page 28: 1, 3, and 4 are valid reasons. Answer 5 can be generally true because high volume can make the extra investment to reduce die size by, say, 10% a good economic decision, but it doesn't have to be true.

§1.6, page 33: 1. a: both, b: latency, c: neither. 7 seconds.

§1.6, page 40: b.

§1.10, page 51: a. Computer A has the higher MIPS rating. b. Computer B is faster.

Answers to Check Yourself

2

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

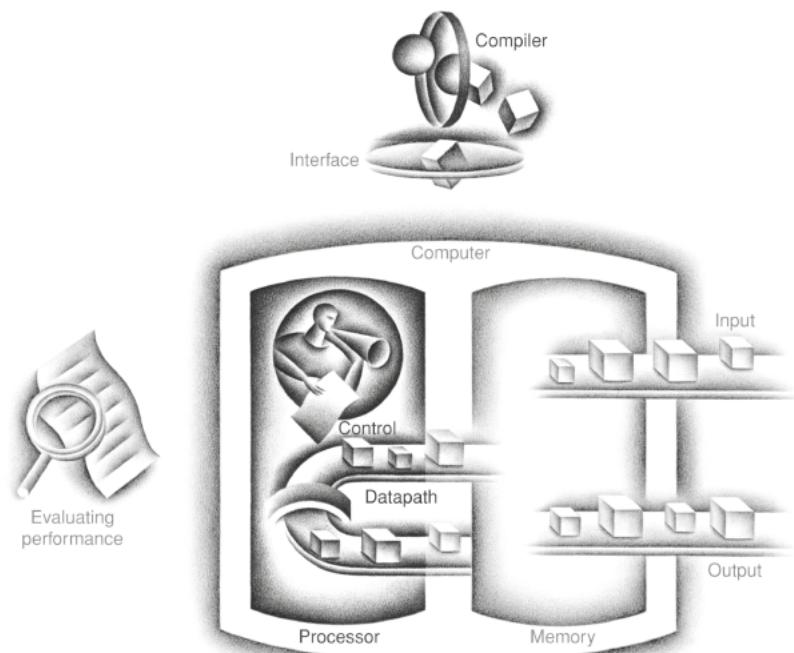
Charles V, Holy Roman Emperor
(1500–1558)

Instructions: Language of the Computer

- 2.1 Introduction** 62
- 2.2 Operations of the Computer Hardware** 63
- 2.3 Operands of the Computer Hardware** 66
- 2.4 Signed and Unsigned Numbers** 73
- 2.5 Representing Instructions in the
Computer** 80
- 2.6 Logical Operations** 87
- 2.7 Instructions for Making Decisions** 90

- 2.8 Supporting Procedures in Computer Hardware** 96
- 2.9 Communicating with People** 106
- 2.10 MIPS Addressing for 32-Bit Immediates and Addresses** 111
- 2.11 Parallelism and Instructions: Synchronization** 121
- 2.12 Translating and Starting a Program** 123
- 2.13 A C Sort Example to Put It All Together** 132
- 2.14 Arrays versus Pointers** 141
- 2.15 Advanced Material: Compiling C and Interpreting Java** 145
- 2.16 Real Stuff: ARMv7 (32-bit) Instructions** 145
- 2.17 Real Stuff: x86 Instructions** 149
- 2.18 Real Stuff: ARMv8 (64-bit) Instructions** 158
- 2.19 Fallacies and Pitfalls** 159
- 2.20 Concluding Remarks** 161
- 2.21 Historical Perspective and Further Reading** 163
- 2.22 Exercises** 164

The Five Classic Components of a Computer



2.1

Introduction

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

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

You might think that the languages of computers would be as diverse as those of people, but in reality computer languages are quite similar, more like regional dialects than like independent languages. Hence, once you learn one, it is easy to pick up others.

The chosen instruction set comes from MIPS Technologies, and is an elegant example of the instruction sets designed since the 1980s. To demonstrate how easy it is to pick up other instruction sets, we will take a quick look at three other popular instruction sets.

1. ARMv7 is similar to MIPS. More than 9 billion chips with ARM processors were manufactured in 2011, making it the most popular instruction set in the world.
2. The second example is the Intel x86, which powers both the PC and the cloud of the PostPC Era.
3. The third example is ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, as we shall see, this 2013 instruction set is closer to MIPS than it is to ARMv7.

This similarity of instruction sets occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all computers must provide. Moreover, computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy. 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.

The “simplicity of the equipment” is as valuable a consideration for today’s computers as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in hardware and the relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language;  [Section 2.15](#) 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.

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

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.

2.2

Operations of the Computer Hardware

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

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

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

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

The following sequence of instructions adds the four variables:

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

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

The words to the right of the sharp symbol (#) on each line above are *comments* for the human reader, so the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one

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

Burks, Goldstine, and von Neumann, 1947

MIPS operands

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

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	l1 \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2+20]=\$s1; \$s1=0 \text{ or } 1$	Store word as 2nd half of atomic swap
Logical	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
	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	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Shift left by constant
Conditional branch	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Shift right by constant
	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	set less than immediate unsigned	sltiu \$s1,\$s2,20	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant unsigned
	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	$\$ra = \text{PC} + 4$; go to 10000	For procedure call

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

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 three underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

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

Compiling Two C Assignment Statements into MIPS

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.

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
```

EXAMPLE

ANSWER

Compiling a Complex C Assignment into MIPS

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?

EXAMPLE

ANSWER

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

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

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

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

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

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

Check Yourself

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

1. Java
2. C
3. MIPS assembly language

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

2.3

Operands of the Computer Hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called *registers*. Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name **word** in the MIPS architecture.

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

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS. (See [Section 2.21](#) 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 three underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

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

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

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

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

Compiling a C Assignment Using Registers

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

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

EXAMPLE

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

ANSWER

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

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

Memory Operands

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

Recall the five components of a computer introduced in Chapter 1 and repeated on page 61. The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

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

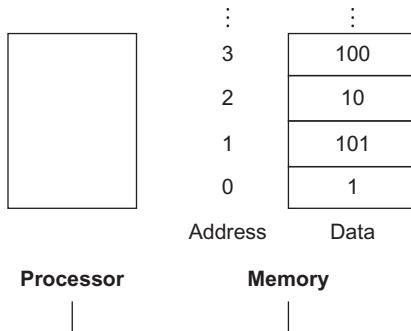


FIGURE 2.2 Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since MIPS actually uses byte addressing, with each word representing four bytes. [Figure 2.3](#) shows the memory addressing for sequential word addresses.

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

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.

Compiling an Assignment When an Operand Is in Memory

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];
```

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]
```

(We'll be making 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] (contained in \$t0) and put the sum in the register corresponding to g (associated with \$s1):

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

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

EXAMPLE**ANSWER**

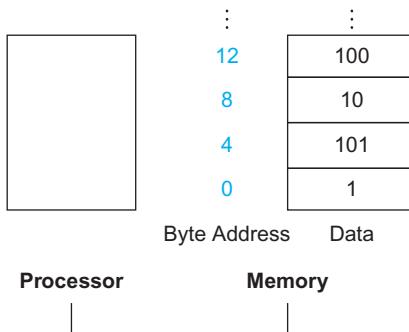
In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

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

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

**Hardware/
Software
Interface**

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

**FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words.**

The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

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. Since the order matters only if you access the identical data both as a word and as four bytes, few need to be aware of the endianess. (Appendix A shows the two options to number bytes in a word.)

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register \$s3 must be 4×8 , or 32*, so that the load address will select $A[8]$ and not $A[8/4]$. (See the related pitfall on page 160 of Section 2.19.)

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*.

Hardware/ Software Interface

As the addresses in loads and stores are binary numbers, we can see why the DRAM for main memory comes in binary sizes rather than in decimal sizes. That is, in gibibytes (2^{30}) or tebibytes (2^{40}), not in gigabytes (10^9) or terabytes (10^{12}); see Figure 1.1.

Compiling Using Load and Store

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];
```

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 in the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select *A*[8], and the add instruction places the sum in \$t0:

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

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

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

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

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

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

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

Thus, registers take less time to access and have higher throughput than memory, making data in registers both faster to access and simpler to use. Accessing registers also uses less energy than accessing memory. To achieve highest performance and conserve energy, an instruction set architecture must have a sufficient number of registers, and compilers must use registers efficiently.

EXAMPLE

ANSWER

Hardware/ Software Interface

Constant or Immediate Operands

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

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

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

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

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

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

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

The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, the move operation is just an add instruction where one operand is zero. Hence, MIPS dedicates a register \$zero to be hard-wired to the value zero. (As you might expect, it is register number 0.) Using frequency to justify the inclusions of constants is another example of the great idea of making the **common case fast**.



COMMON CASE FAST

Check Yourself

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

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

Elaboration: Although the MIPS registers in this book are 32 bits wide, there is a 64-bit version of the MIPS instruction set with 32 64-bit registers. To keep them straight, they are officially called MIPS-32 and MIPS-64. In this chapter, we use a subset of MIPS-32.  [Appendix E](#) shows the differences between MIPS-32 and MIPS-64. Sections 2.16 and 2.18 show the much more dramatic difference between the 32-bit address ARMv7 and its 64-bit successor, ARMv8.

Elaboration: 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.

Elaboration: 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.

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

2.4

Signed and Unsigned Numbers

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

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

A single digit of a binary number is thus the “atom” of computing, since all information is composed of **binary digits** or *bits*. This fundamental building block can be one of two values, which can be thought of as several alternatives: high or low, on or off, true or false, or 1 or 0.

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

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

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

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

represents

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

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

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

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

(32 bits wide)

least significant bit The rightmost bit in a MIPS word.

most significant bit The leftmost bit in a MIPS word.

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).

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 $2^{32} - 1$ ($4,294,967,295_{\text{ten}}$):

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}
...									...	
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}

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here x_i means the i th bit of x):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \dots + (x1 \times 2^1) + (x0 \times 2^0)$$

For reasons we will shortly see, these positive numbers are called unsigned numbers.

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.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred.

It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude representation was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

0000 0000 0000 0000 0000 0000 0000 0000 _{two}	= 0 _{ten}
0000 0000 0000 0000 0000 0000 0001 _{two}	= 1 _{ten}
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 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...0000_{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...0001_{two}) down to -1_{ten} (1111...1111_{two}).

Two's complement does have one negative number, -2,147,483,648_{ten}, that has no corresponding positive number. Such imbalance was also a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer and the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \dots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

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:

$$\begin{aligned} & (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^0) \\ &= -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\ &= -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}} \\ &= -4_{\text{ten}} \end{aligned}$$

We'll see a shortcut to simplify conversion from negative to positive soon.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware/ Software Interface

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* (`l b`) treats the byte as a signed number and thus sign-extends to fill the 24 left-most bits of the register, while *load byte unsigned* (`l bu`) works with unsigned integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `l bu` is used practically exclusively for byte loads.

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.

Hardware/ Software Interface

Let's examine two useful shortcuts when working with two's complement numbers. The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111 \dots 111_{\text{two}}$, which represents -1 . Since $x + \bar{x} = -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$. (We use the notation \bar{x} to mean invert every bit in x from 0 to 1 and vice versa.)

Negation Shortcut

Negate 2_{ten} , and then check the result by negating -2_{ten} .

$$2_{\text{ten}} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}}$$

EXAMPLE

Negating this number by inverting the bits and adding one,

$$\begin{array}{r} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{two}} \\ + \hspace{1cm} 1_{\text{two}} \\ \hline = \hspace{1cm} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} \\ = \hspace{1cm} -2_{\text{ten}} \end{array}$$

ANSWER

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

is first inverted and then incremented:

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} \\ + \hspace{1cm} 1_{\text{two}} \\ \hline = \hspace{1cm} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} \\ = \hspace{1cm} 2_{\text{ten}} \end{array}$$

Our next shortcut tells us how to convert a binary number represented in n bits to a number represented with more than n bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32,768_{\text{ten}}$ (-2^{15}) to $32,767_{\text{ten}}$ ($2^{15} - 1$). 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 nonsign bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension*.

EXAMPLE

Sign Extension Shortcut

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

ANSWER

The 16-bit binary version of the number 2 is

$$0000 \ 0000 \ 0000 \ 0010_{\text{two}} = 2_{\text{ten}}$$

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:

$$0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0010_{\text{two}} = 2_{\text{ten}}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000 \ 0000 \ 0000 \ 0010_{\text{two}}$$

becomes

$$\begin{array}{r} 1111 \ 1111 \ 1111 \ 1101_{\text{two}} \\ + \qquad \qquad \qquad 1_{\text{two}} \\ \hline \\ = 1111 \ 1111 \ 1111 \ 1110_{\text{two}} \end{array}$$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_{\text{two}} = -2_{\text{ten}}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the unanimous choice since 1965 has been two's complement.

Elaboration: For signed decimal numbers, we used “-” to represent negative because there are no limits to the size of a decimal number. Given a fixed word size, binary and hexadecimal (see Figure 2.4) bit strings can encode the sign; hence we do not normally use “+” or “-” with binary or hexadecimal notation.

What is the decimal value of this 64-bit two's complement number?

- 1) -4_{ten}
 - 2) -8_{ten}
 - 3) -16_{ten}
 - 4) 18,446,744,073,709,551,609_{ten}

Check Yourself

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n -bit number and its n -bit negative is 2^n ; hence, the negation or complement of a number x is $2^n - x$, or its “two’s complement.”

A third alternative representation to two's complement and sign and magnitude is called **one's complement**. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, or \bar{x} . This relation helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: 00 ... 00_{two} is positive 0 and 11 ... 11_{two} is negative 0. The most negative number, 10 ... 000_{two}, represents $-2,147,483,647$ _{ten}, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point in Chapter 3, is to represent the most negative value by $00 \dots 000_{\text{two}}$ and the most positive value by $11 \dots 11_{\text{two}}$, with 0 typically having the value $10 \dots 00_{\text{two}}$. This is called a **biased notation**, since it biases the number such that the number plus the bias has a non-negative representation.

one's complement

A notation that represents the most negative value by $10\dots000_{\text{two}}$ and the most positive value by $01\dots11_{\text{two}}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive ($00\dots00_{\text{two}}$) and one negative ($11\dots11_{\text{two}}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

biased notation

A notation that represents the most negative value by $00 \dots 000_{\text{two}}$ and the most positive value by $11 \dots 11_{\text{two}}$, with 0 typically having the value $10 \dots 00_{\text{two}}$, thereby biasing the number such that the number plus the bias has a non-negative representation.

2.5

Representing Instructions In the Computer

We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions.

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.

Since registers are referred to in 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.

EXAMPLE

Translating a MIPS Assembly Instruction into a Machine Instruction

Let's do the next step in the refinement of the MIPS language as an example. We'll show the real MIPS language version of the instruction represented symbolically as

`add $t0,$s1,$s2`

first as a combination of decimal numbers and then of binary numbers.

ANSWER

The decimal representation is

0	17	18	8	0	32
---	----	----	---	---	----

Each of these segments of an instruction is called a *field*. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation ($17 = \$s1$), and the third field gives the other source operand for the addition ($18 = \$s2$). The fourth field contains the number of the register that is to receive the sum ($8 = \$t0$). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register $\$s1$ to register $\$s2$ and places the sum in register $\$t0$.

This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this MIPS instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all MIPS instructions are 32 bits long.

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

It would appear that you would now be reading and writing long, tedious strings of binary numbers. We avoid that tedium by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, **hexadecimal** (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 2.4 converts between hexadecimal and binary.

instruction format

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

machine

language Binary representation used for communication within a computer system.

hexadecimal Numbers in base 16.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0 _{hex}	0000 _{two}	4 _{hex}	0100 _{two}	8 _{hex}	1000 _{two}	c _{hex}	1100 _{two}
1 _{hex}	0001 _{two}	5 _{hex}	0101 _{two}	9 _{hex}	1001 _{two}	d _{hex}	1101 _{two}
2 _{hex}	0010 _{two}	6 _{hex}	0110 _{two}	a _{hex}	1010 _{two}	e _{hex}	1110 _{two}
3 _{hex}	0011 _{two}	7 _{hex}	0111 _{two}	b _{hex}	1011 _{two}	f _{hex}	1111 _{two}

FIGURE 2.4 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Because we frequently deal with different number bases, to avoid confusion we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation 0x_{nnnn} for hexadecimal numbers.

Binary to Hexadecimal and Back

Convert the following hexadecimal and binary numbers into the other base:

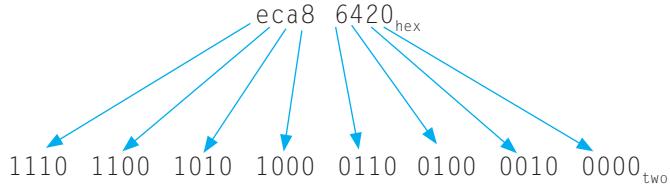
eca8 6420_{hex}

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

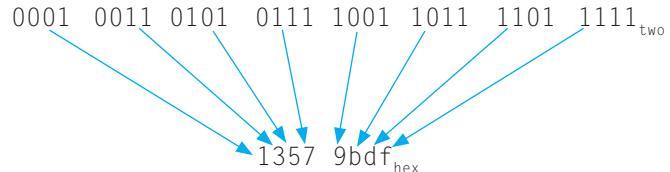
EXAMPLE

ANSWER

Using Figure 2.4, the answer is just a table lookup one way:



And then the other direction:



MIPS Fields

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

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Here is the meaning of each name of the fields in MIPS instructions:

- **op**: Basic operation of the instruction, traditionally called the **opcode**.
- **rs**: The first register source operand.
- **rt**: The second register source operand.
- **rd**: The register destination operand. It gets the result of the operation.
- **shamt**: Shift amount. (Section 2.6 explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
- **funct**: Function. This field, often called the *function code*, selects the specific variant of the operation in the op field.

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 3: Good design demands good compromises.

The compromise chosen by the MIPS designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. For example, the format above is called *R-type* (for register) or *R-format*. A second type of instruction format is called *I-type* (for immediate) or *I-format* and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$. We see that more than 32 registers would be difficult in this format, as the rs and rt fields would each need another bit, making it harder to fit everything in one word.

Let's look at the load word instruction from page 71:

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

Here, 19 (for \$s3) is placed in the rs field, 8 (for \$t0) is placed in the rt field, and 32 is placed in the address field. Note that the meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the *destination* register, which receives the result of the load.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first three fields of the R-type and I-type formats are the same size and have the same names; the length of the fourth field in I-type is equal to the sum of the lengths of the last three fields of R-type.

In case you were wondering, the formats are distinguished by the values in the first field: each format is assigned a distinct set of values in the first field (op) so that the hardware knows whether to treat the last half of the instruction as three fields (R-type) or as a single field (I-type). Figure 2.5 shows the numbers used in each field for the MIPS instructions covered so far.

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

FIGURE 2.5 MIPS instruction encoding. In the table above, “reg” means a register number between 0 and 31, “address” means a 16-bit address, and “n.a.” (not applicable) means this field does not appear in this format. Note that add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

EXAMPLE**Translating MIPS Assembly Language into Machine Language**

We can now take an example all the way from what the programmer writes to what the computer executes. If \$t1 has the base of the array A and \$s2 corresponds to h, the assignment statement

$A[300] = h + A[300];$

is compiled into

```
lw    $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw    $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

ANSWER

For convenience, let's first represent the machine language instructions using decimal numbers. From [Figure 2.5](#), we can determine the three machine language instructions:

Op	rs	rt	rd	address/ shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

The `lw` instruction is identified by 35 (see [Figure 2.5](#)) in the first field (op). The base register 9 (\$t1) is specified in the second field (rs), and the destination register 8 (\$t0) is specified in the third field (rt). The offset to select $A[300]$ ($1200 = 300 \times 4$) is found in the final field (address).

The `add` instruction that follows is specified with 0 in the first field (op) and 32 in the last field (funct). The three register operands (18, 8, and 8) are found in the second, third, and fourth fields and correspond to \$s2, \$t0, and \$t0.

The `sw` instruction is identified with 43 in the first field. The rest of this final instruction is identical to the `lw` instruction.

Since $1200_{\text{ten}} = 0000\ 0100\ 1011\ 0000_{\text{two}}$, the binary equivalent to the decimal form is:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Note the similarity of the binary representations of the first and last instructions. The only difference is in the third bit from the left, which is highlighted here.

The desire to keep all instructions the same size is in conflict with the desire to have as many registers as possible. Any increase in the number of registers uses up at least one more bit in every register field of the instruction format. Given these constraints and the design principle that smaller is faster, most instruction sets today have 16 or 32 general purpose registers.

Hardware/ Software Interface

Figure 2.6 summarizes the portions of MIPS machine language described in this section. As we shall see in Chapter 4, the similarity of the binary representations of related instructions simplifies hardware design. These similarities are another example of regularity in the MIPS architecture.

MIPS machine language								
Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

The BIG Picture

Today's computers are built on two key principles:

1. Instructions are represented as numbers.
2. Programs are stored in memory to be read or written, just like data.

These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 2.7 shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such “binary compatibility” often leads industry to align around a small number of instruction set architectures.

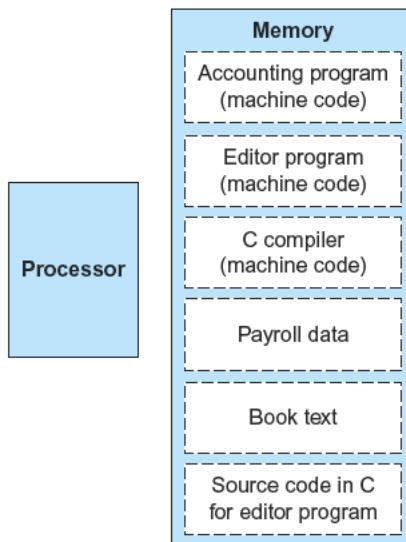


FIGURE 2.7 The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

What MIPS instruction does this represent? Choose from one of the four options below.

Check Yourself

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub \$t0, \$t1, \$t2
2. add \$t2, \$t0, \$t1
3. sub \$t2, \$t1, \$t0
4. sub \$t2, \$t0, \$t1

2.6 Logical Operations

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see Section 2.9). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.8 shows logical operations in C, Java, and MIPS.

“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

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

FIGURE 2.8 C and Java logical operators and their corresponding MIPS Instructions. MIPS implements NOT using a NOR with one operand being zero.

The first class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register \$s0 contained

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{\text{two}} = 9_{\text{ten}}$$

and the instruction to shift left by 4 was executed, the new value would be:

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

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (`sll`) and *shift right logical* (`srl`). The following instruction performs the operation above, assuming that the original value was in register `$s0` and the result should go in register `$t2`:

```
sll    $t2,$s0,4  # reg $t2 = reg $s0 << 4 bits
```

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of `sll` is 0 in both the op and funct fields, rd contains 10 (register `$t2`), rt contains 16 (register `$s0`), and shamt contains 4. The rs field is unused and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by i bits gives the same result as multiplying by 2^i , just as shifting a decimal number by i digits is equivalent to multiplying by 10^i . For example, the above `sll` shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and $9 \times 16 = 144$, the value of the second bit pattern.

Another useful operation that isolates fields is **AND**. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register `$t2` contains

```
0000 0000 0000 0000 0000 1101 1100 0000two
```

and register `$t1` contains

```
0000 0000 0000 0000 0011 1100 0000 0000two
```

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 0000two
```

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.

AND A logical bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in *both* operands.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers \$t1 and \$t2 are unchanged from the preceding example, the result of the MIPS instruction

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

is this value in register \$t0:

```
0000 0000 0000 0000 0011 1101 1100 0000two
```

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. Using our prior notation, it calculates \bar{x} .

In keeping with the three-operand format, the designers of MIPS decided to include the instruction **NOR** (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT: A NOR 0 = NOT (A OR 0) = NOT (A).

If the register \$t1 is unchanged from the preceding example and register \$t3 has the value 0, the result of the MIPS instruction

```
nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

is this value in register \$t0:

```
1111 1111 1111 1111 1100 0011 1111 1111two
```

Figure 2.8 above shows the relationship between the C and Java operators and the MIPS instructions. Constants are useful in AND and OR logical operations as well as in arithmetic operations, so MIPS also provides the instructions *and immediate* (`andi`) and *or immediate* (`ori`). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version of NOR.

Elaboration: The full MIPS instruction set also includes exclusive or (XOR), which sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same. C allows *bit fields* or *fields* to be defined within words, both allowing objects to be packed within a word and to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: `and`, `or`, `sll`, and `srl`.

Elaboration: Logical AND immediate and logical OR immediate put Os into the upper 16 bits to form a 32-bit constant, unlike add immediate, which does sign extension.

Which operations can isolate a field in a word?

1. AND
2. A shift left followed by a shift right

OR A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand.

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

NOR A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in *both* operands.

Check Yourself

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation.... This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Burks, Goldstine, and von Neumann, 1947

EXAMPLE

ANSWER

2.7

Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
beq register1, register2, L1
```

This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic *beq* stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled L1 if the value in register1 does *not* equal the value in register2. The mnemonic *bne* stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

Compiling If-then-else into Conditional Branches

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;
```

Figure 2.9 shows a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (*beq*). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label *Else* is defined below) and so we use the branch if registers are *not* equal instruction (*bne*):

```
bne $s3,$s4,Else    # go to Else if i != j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2      # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as *j* (the label *Exit* is defined below).

```
j Exit      # go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label *Else* to this instruction. We also show the label *Exit* that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:sub $s0,$s1,$s2  # f = g - h (skipped if i = j)
Exit:
```

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.12).

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.

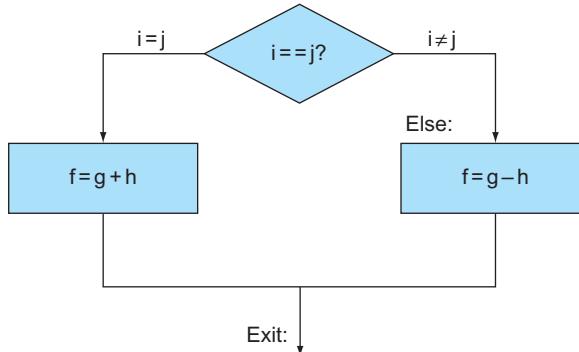


FIGURE 2.9 Illustration of the options in the if statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Hardware/ Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

EXAMPLE

Compiling a **while** Loop in C

Here is a traditional loop in C:

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

Assume that *i* and *k* correspond to registers \$s3 and \$s5 and the base of the array *save* is in \$s6. What is the MIPS assembly code corresponding to this C segment?

ANSWER

The first step is to load *save[i]* into a temporary register. Before we can load *save[i]* into a temporary register, we need to have its address. Before we can add *i* to the base of array *save* to form the address, we must multiply the index *i* by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 88 in the prior section). We need to add the label *Loop* to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll $t1,$s3,2      # Temp reg $t1 = i * 4
```

To get the address of *save[i]*, we need to add *\$t1* and the base of *save* in \$s6:

```
add $t1,$t1,$s6      # $t1 = address of save[i]
```

Now we can use that address to load *save[i]* into a temporary register:

```
lw $t0,0($t1)      # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if *save[i]* \neq *k*:

```
bne $t0,$s5, Exit    # go to Exit if save[i]  $\neq$  k
```

The next instruction adds 1 to *i*:

```
addi $s3,$s3,1      # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
j      Loop      # go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

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.

Hardware/ Software Interface

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than*, or `slt`. For example,

```
slt      $t0, $s3, $s4      # $t0 = 1 if $s3 < $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

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

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).

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.

Hardware/ Software Interface

Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

Hardware/ Software Interface

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slt i`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

EXAMPLE

Signed versus Unsigned Comparison

Suppose register `$s0` has the binary number

1111 1111 1111 1111 1111 1111 1111 1111_{two}

and that register `$s1` has the binary number

0000 0000 0000 0000 0000 0000 0000 0001_{two}

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt      $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

ANSWER

The value in register `$s0` represents -1_{ten} if it is an integer and $4,294,967,295_{\text{ten}}$ if it is an unsigned integer. The value in register `$s1` represents 1_{ten} in either case. Then register `$t0` has the value 1, since $-1_{\text{ten}} < 1_{\text{ten}}$, and register `$t1` has the value 0, since $4,294,967,295_{\text{ten}} > 1_{\text{ten}}$.

Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y .

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if $\$s1 \geq \$t2$ or if $\$s1$ is negative.

The checking code just uses `u` to do both checks:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0  
beq $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

EXAMPLE

ANSWER

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`j r`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction. We'll see an even more popular use of `j r` in the next section.

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

Hardware/ Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

Check Yourself

Elaboration: If you have heard about *delayed branches*, covered in Chapter 4, don't worry: the MIPS assembler makes them invisible to the assembly language programmer.

- I. C has many statements for decisions and loops, while MIPS has few. Which of the following do or do not explain this imbalance? Why?
 1. More decision statements make code easier to read and understand.
 2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
 3. More decision statements mean fewer lines of code, which generally reduces coding time.
 4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.
- II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while MIPS doesn't?
 1. Logical operations AND and OR implement & and |, while conditional branches implement && and ||.
 2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
 3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.



ABSTRACTION

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

2.8

Supporting Procedures In Computer Hardware

A **procedure** or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results. We describe the equivalent to procedures in Java in [Section 2.15](#), but Java needs everything from a computer that C needs. Procedures are one way to implement **abstraction** in software.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a “need to know” basis, so the spy can’t make assumptions about his employer.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- \$a0-\$a3: four argument registers in which to pass parameters
- \$v0-\$v1: two value registers in which to return values
- \$ra: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register \$ra. The **jump-and-link instruction** (`jal`) is simply written

```
jal ProcedureAddress
```

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This “link,” stored in register \$ra (register 31), is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

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

```
jr    $ra
```

jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (\$ra in MIPS).

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register \$ra.

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

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

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

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register \$sp.

push Add element to stack.

pop Remove element from stack.

EXAMPLE

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 and \$v1, and returns control to the caller using `jr $ra`.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *PC* in the MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction actually saves PC + 4 in register \$ra to link to the following instruction to set up the procedure return.

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 above.

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The **stack pointer** is adjusted by one word for each register that is saved or restored. MIPS software reserves register 29 for the stack pointer, giving it the obvious name \$sp. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a **push**, and removing data from the stack is called a **pop**.

By historical precedent, stacks “grow” from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a C Procedure That Doesn’t Call Another Procedure

Let’s turn the example on page 65 from Section 2.2 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

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

What is the compiled MIPS assembly code?

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:

ANSWER

`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 68, which uses two temporary registers. Thus, we need to save three registers: \$s0, \$t0, and \$t1. We “push” the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw  $t1, 8($sp)      # save register $t1 for use afterwards
sw  $t0, 4($sp)      # save register $t0 for use afterwards
sw  $s0, 0($sp)      # save register $s0 for use afterwards
```

[Figure 2.10](#) shows the stack before, during, and after the procedure call.

The next three statements correspond to the body of the procedure, which follows the example on page 68:

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

To return the value of *f*, we copy it into a return value register:

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Before returning, we restore the three old values of the registers we saved by “popping” them from the stack:

```
lw $s0, 0($sp)    # restore register $s0 for caller
lw $t0, 4($sp)    # restore register $t0 for caller
lw $t1, 8($sp)    # restore register $t1 for caller
addi $sp,$sp,12   # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr $ra    # jump back to calling routine
```

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- \$t0-\$t9: temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- \$s0-\$s7: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

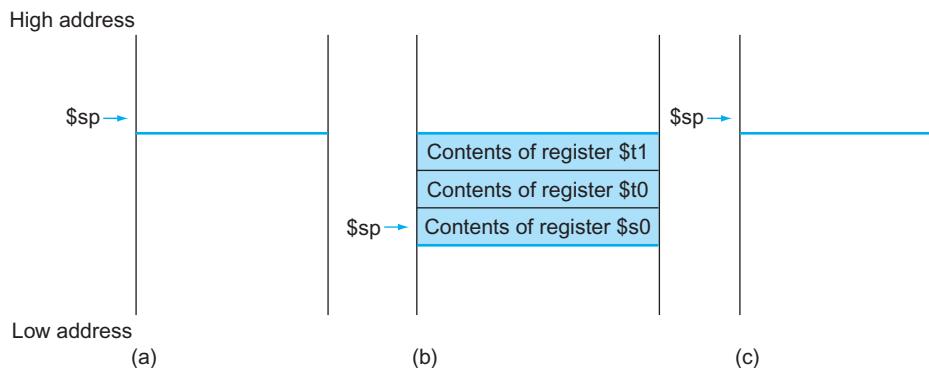


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

This simple convention reduces register spilling. In the example above, since the caller does not expect registers $\$t0$ and $\$t1$ to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore $\$s0$, since the callee must assume that the caller needs its value.

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 $\text{jal } A$. Then suppose that procedure A calls procedure B via $\text{jal } B$ 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?

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:

ANSWER

```
fact:
    addi  $sp, $sp, -8 # adjust stack for 2 items
    sw    $ra, 4($sp) # save the return address
    sw    $a0, 0($sp) # save the argument n
```

The first time *fact* is called, *sw* saves an address in the program that called *fact*. The next two instructions test whether *n* is less than 1, going to L1 if *n* ≥ 1 .

```
slti  $t0,$a0,1      # test for n < 1
beq   $t0,$zero,L1  # if n >= 1, go to L1
```

If *n* is less than 1, *fact* returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in \$v0. It then pops the two saved values off the stack and jumps to the return address:

```
addi  $v0,$zero,1 # return 1
addi  $sp,$sp,8   # pop 2 items off stack
jr   $ra           # return to caller
```

Before popping two items off the stack, we could have loaded \$a0 and \$ra. Since \$a0 and \$ra don't change when *n* is less than 1, we skip those instructions.

If *n* is not less than 1, the argument *n* is decremented and then *fact* is called again with the decremented value:

```
L1: addi $a0,$a0,-1  # n >= 1: argument gets (n - 1)
     jal fact          # call fact with (n - 1)
```

The next instruction is where `fact` returns. Now the old return address and old argument are restored, along with the stack pointer:

```
lw    $a0, 0($sp)  # return from jal: restore argument n
lw    $ra, 4($sp)  # restore the return address
addi $sp, $sp, 8   # adjust stack pointer to pop 2 items
```

Next, the value register `$v0` gets the product of old argument `$a0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul  $v0,$a0,$v0  # return n * fact (n - 1)
```

Finally, `fact` jumps again to the return address:

```
jr   $ra           # return to the caller
```

Hardware/ Software Interface

global pointer The register that is reserved to point to the static area.

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Examples include integers and characters (see Section 2.9). C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*. The rest are automatic. To simplify access to static data, MIPS software reserves another register, called the **global pointer**, or `$gp`.

Figure 2.11 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above `$sp` is preserved simply by making sure the callee does not write above `$sp`; `$sp` is

Preserved	Not preserved
Saved registers: <code>\$s0-\$s7</code>	Temporary registers: <code>\$t0-\$t9</code>
Stack pointer register: <code>\$sp</code>	Argument registers: <code>\$a0-\$a3</code>
Return address register: <code>\$ra</code>	Return value registers: <code>\$v0-\$v1</code>
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.11 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

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.

Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.12 shows the state of the stack before, during, and after the procedure call.

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

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.

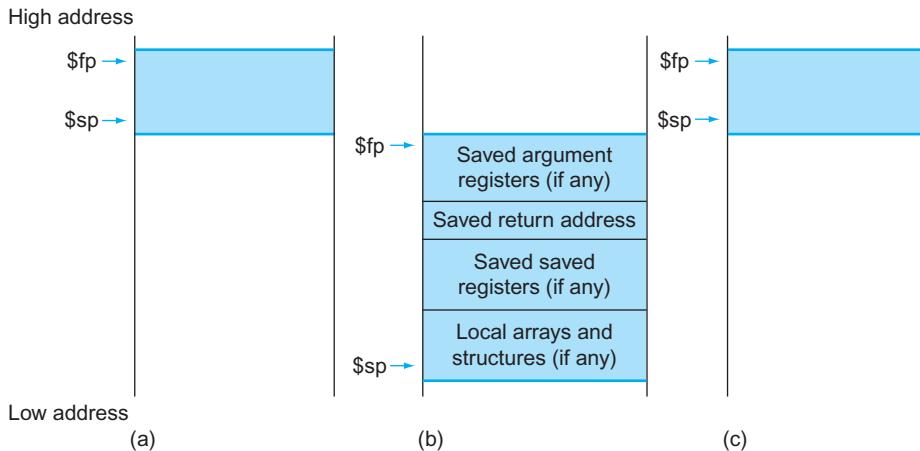


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer ($\$fp$) points to the first word of the frame, often a saved argument register, and the stack pointer ($\$sp$) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in $\$sp$ on a call, and $\$sp$ is restored using $\$fp$. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Allocating Space for New Data on the Heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. Figure 2.13 shows the MIPS convention for allocation of memory. The stack starts in the high end of memory and grows down. The first part of the low end of memory is reserved, followed by the home of the MIPS machine code, traditionally called the **text segment**. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be a fixed length and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap*, and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

text segment The segment of a UNIX object file that contains the machine language code for routines in the source file.

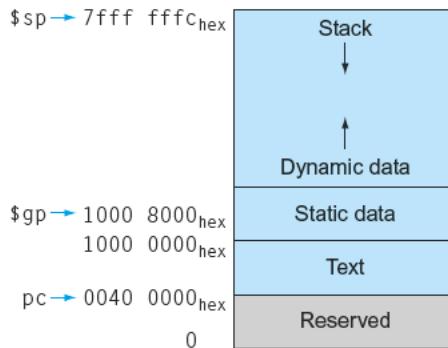


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to 7fff fffc_{hex} and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 0000_{hex}. The static data starts at 1000 0000_{hex}. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, \$gp, is set to an address to make it easy to access data. It is initialized to 1000 8000_{hex} so that it can access from 1000 0000_{hex} to 1000 ffff_{hex} using the positive and negative 16-bit offsets from \$gp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

C allocates and frees space on the heap with explicit functions. `malloc()` allocates space on the heap and returns a pointer to it, and `free()` releases space on the heap to which the pointer points. Memory allocation is controlled by programs in C, and it is the source of many common and difficult bugs. Forgetting to free space leads to a “memory leak,” which eventually uses up so much memory that the operating system may crash. Freeing space too early leads to “dangling pointers,” which can cause pointers to point to things that the program never intended. Java uses automatic memory allocation and garbage collection just to avoid such bugs.

Figure 2.14 summarizes the register conventions for the MIPS assembly language. This convention is another example of making the **common case fast**: most procedures can be satisfied with up to 4 arguments, 2 registers for a return value, 8 saved registers, and 10 temporary registers without ever going to memory.



COMMON CASE FAST

Name	Register number	Usage	Preserved on call?
\$zero	0	The constant value 0	n.a.
\$v0-\$v1	2-3	Values for results and expression evaluation	no
\$a0-\$a3	4-7	Arguments	no
\$t0-\$t7	8-15	Temporaries	no
\$s0-\$s7	16-23	Saved	yes
\$t8-\$t9	24-25	More temporaries	no
\$gp	28	Global pointer	yes
\$sp	29	Stack pointer	yes
\$fp	30	Frame pointer	yes
\$ra	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called \$at, is reserved for the assembler (see Section 2.12), and registers 26–27, called \$k0–\$k1, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

Elaboration: What if there are more than four parameters? The MIPS convention is to place the extra parameters on the stack just above the frame pointer. The procedure then expects the first four parameters to be in registers \$a0 through \$a3 and the rest in memory, addressable via the frame pointer.

As mentioned in the caption of Figure 2.12, the frame pointer is convenient because all references to variables in the stack within a procedure will have the same offset. The frame pointer is not necessary, however. The GNU MIPS C compiler uses a frame pointer, but the C compiler from MIPS does not; it treats register 30 as another save register (\$s8).

Elaboration: Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with recursive procedure calls. For example, consider a procedure used to accumulate a sum:

```
int sum (int n, int acc) {
    if (n >0)
        return sum(n - 1, acc + n);
    else
        return acc;
}
```

Consider the procedure call `sum(3,0)`. This will result in recursive calls to `sum(2,3)`, `sum(1,5)`, and `sum(0,6)`, and then the result 6 will be returned four

times. This recursive call of sum is referred to as a *tail call*, and this example use of tail recursion can be implemented very efficiently (assume \$a0 = n and \$a1 = acc):

```
sum: slti $t0, $a0, 1          # test if n <= 0
      bne $t0, $zero, sum_exit # go to sum_exit if n <= 0
      add$ a1, $a1, $a0         # add n to acc
      addi$ a0, $a0, -1         # subtract 1 from n
      j sum                      # go to sum
sum_exit:
      add$ v0, $a1, $zero        # return value acc
      jr $ra                      # return to caller
```

Check Yourself

!(@ | = > (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).

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

1. C programmers manage data explicitly, while it’s automatic in Java.
2. C leads to more pointer bugs and memory leak bugs than does Java.

2.9

Communicating with People

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today offer 8-bit bytes to represent characters, with the *American Standard Code for Information Interchange* (ASCII) being the representation that nearly everyone follows. Figure 2.15 summarizes ASCII.

ASCII value	Character										
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.

ASCII versus Binary Numbers

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

EXAMPLE

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. Beyond the expansion in storage, the hardware to add, subtract, multiply, and divide such decimal numbers is difficult and would consume more energy. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

ANSWER

A series of instructions can extract a byte from a word, so load word and store word are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, MIPS provides instructions to move bytes. *Load byte* (`l b`) loads a byte from memory, placing it in the rightmost 8 bits of a register. *Store byte* (`s b`) 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. (As we shall see, Java uses the first option.)

EXAMPLE**Compiling a String Copy Procedure, Showing How to Use C Strings**

The procedure `strcpy` copies string `y` to string `x` using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```

What is the MIPS assembly code?

ANSWER

Below is the basic MIPS assembly code segment. Assume that base addresses for arrays `x` and `y` are found in `$a0` and `$a1`, while `i` is in `$s0`. `strcpy` adjusts the stack pointer and then saves the saved register `$s0` on the stack:

```
strcpy:
    addi  $sp,$sp,-4  # adjust stack for 1 more item
    sw    $s0, 0($sp)  # save $s0
```

To initialize `i` to 0, the next instruction sets `$s0` to 0 by adding 0 to 0 and placing that sum in `$s0`:

```
add    $s0,$zero,$zero # i = 0 + 0
```

This is the beginning of the loop. The address of `y[i]` is first formed by adding `i` to `y[]`:

```
L1: add    $t1,$s0,$a1  # address of y[i] in $t1
```

Note that we don't have to multiply `i` by 4 since `y` is an array of *bytes* and not of words, as in prior examples.

To load the character in `y[i]`, we use `load byte unsigned`, which puts the character into `$t2`:

```
lbu   $t2, 0($t1)  # $t2 = y[i]
```

A similar address calculation puts the address of `x[i]` in `$t3`, and then the character in `$t2` is stored at that address.

```

add    $t3,$s0,$a0  # address of x[i] in $t3
sb    $t2, 0($t3)  # x[i] = y[i]

```

Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:

```
beq    $t2,$zero,L2 # if y[i] == 0, go to L2
```

If not, we increment *i* and loop back:

```

addi   $s0, $s0,1   # i = i + 1
j      L1           # go to L1

```

If we don't loop back, it was the last character of the string; we restore *\$s0* and the stack pointer, and then return.

```

L2: lw     $s0, 0($sp)  # y[i] == 0: end of string.
               # 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 2.14 for an explanation of arrays versus pointers.

Since the procedure `strcpy` above is a leaf procedure, the compiler could allocate *i* to a temporary register and avoid saving and restoring *\$s0*. Hence, instead of thinking of the *\$t* registers as being just for temporaries, we can think of them as registers that the callee should use whenever convenient. When a compiler finds a leaf procedure, it exhausts all temporary registers before using registers it must save.

Characters and Strings in Java

Unicode is a universal encoding of the alphabets of most human languages. Figure 2.16 gives a list of Unicode alphabets; there are almost as many *alphabets* in Unicode as there are useful *symbols* in ASCII. To be more inclusive, Java uses Unicode for characters. By default, it uses 16 bits to represent a character.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURE 2.16 Example alphabets in Unicode. Unicode version 4.0 has more than 160 “blocks,” which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16 or 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. Like *load byte*, *load half* (`lh`) treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register, while *load halfword unsigned* (`lhu`) works with unsigned integers. Thus, `lhu` is the more popular of the two. *Store half* (`sh`) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence

```
lhu $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp) # Write halfword (16 bits) to destination
```

Strings are a standard Java class with special built-in support and predefined methods for concatenation, comparison, and conversion. Unlike C, Java includes a word that gives the length of the string, similar to Java arrays.

Elaboration: MIPS software tries to keep the stack aligned to word addresses, allowing the program to always use `lw` and `sw` (which must be aligned) to access the stack. This convention means that a char variable allocated on the stack occupies 4 bytes, even though it needs less. However, a C string variable or an array of bytes *will* pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

Elaboration: Reflecting the international nature of the web, most web pages today use Unicode instead of ASCII.

I. Which of the following statements about characters and strings in C and Java are true?

1. A string in C takes about half the memory as the same string in Java.
2. Strings are just an informal name for single-dimension arrays of characters in C and Java.
3. Strings in C and Java use null (0) to mark the end of a string.
4. Operations on strings, like length, are faster in C than in Java.

II. Which type of variable that can contain $1,000,000,000_{\text{ten}}$ takes the most memory space?

1. `int` in C
2. `string` in C
3. `string` in Java

Check Yourself

2.10

MIPS Addressing for 32-bit Immediates and Addresses

Although keeping all MIPS instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches and jumps.

32-Bit Immediate Operands

Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger. The MIPS instruction set includes the instruction *load upper immediate* (*lui*) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. [Figure 2.17](#) shows the operation of *lui*.

EXAMPLE

ANSWER

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
```

First, we would load the upper 16 bits, which is 61 in decimal, using *lui*:

```
lui $s0, 61    # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register \$s0 afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to insert the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register \$s0 is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

The machine language version of *lui \$t0, 255 # \$t0 is register 8*:

001111	00000	01000	0000 0000 1111 1111
--------	-------	-------	---------------------

Contents of register \$t0 after executing *lui \$t0, 255*:

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

FIGURE 2.17 The effect of the *lui* instruction. The instruction *lui* transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

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 need is a reason for the register \$at (assembler temporary), which is reserved for the assembler.

Hence, the symbolic representation of the MIPS machine language is no longer limited by the hardware, but by whatever the creator of an assembler chooses to include (see Section 2.12). We stick close to the hardware to explain the architecture of the computer, noting when we use the enhanced language of the assembler that is not found in the processor.

Hardware/ Software Interface

Elaboration: Creating 32-bit constants needs care. The instruction addi copies the left-most bit of the 16-bit immediate field of the instruction into the upper 16 bits of a word. *Logical or immediate* from Section 2.6 loads 0s into the upper 16 bits and hence is used by the assembler in conjunction with lui to create 32-bit constants.

Addressing in Branches and Jumps

The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the *J-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
j 10000 # go to location 10000
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

2	10000
6 bits	26 bits

where the value of the jump opcode is 2 and the jump address is 10000.

Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

```
bne $s0,$s1,Exit # go to Exit if $s0 ≠ $s1
```

is assembled into this instruction, leaving only 16 bits for the branch address:

5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^{16} , which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch address}$$

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the *program counter* (PC) contains the address of the current instruction, we can branch within $\pm 2^{15}$ words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if* statements are much smaller than 2^{16} words, so the PC is the ideal choice.

This form of branch addressing is called **PC-relative addressing**. As we shall see in Chapter 4, it is convenient for the hardware to increment the PC early to point to the next instruction. Hence, the MIPS address is actually relative to the address of the following instruction ($\text{PC} + 4$) as opposed to the current instruction (PC). It is yet another example of making the **common case fast**, which in this case is addressing nearby instructions.

Like most recent computers, MIPS uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch. On the other hand, jump-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 16-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address. Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.

Elaboration: Since the PC is 32 bits, 4 bits must come from somewhere else for jumps. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker (Section 2.12) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

PC-relative addressing An addressing regime in which the address is the sum of the *program counter* (PC) and a constant in the instruction.



COMMON CASE FAST

Showing Branch Offset in Machine Language

The *while* loop on pages 92–93 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?

The assembled instructions and their addresses are:

EXAMPLE

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8		0	
80012	5	8	21		2	
80016	8	19	19		1	
80020	2			20000		
80024	...					

ANSWER

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

Most conditional branches are to a nearby location, but occasionally they branch 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.

EXAMPLE

Branching Far Away

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:
```

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

MIPS Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes**. Figure 2.18 shows how operands are identified for each addressing mode. The MIPS addressing modes are the following:

1. *Immediate addressing*, where the operand is a constant within the instruction itself
2. *Register addressing*, where the operand is a register
3. *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
4. *PC-relative addressing*, where the branch 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

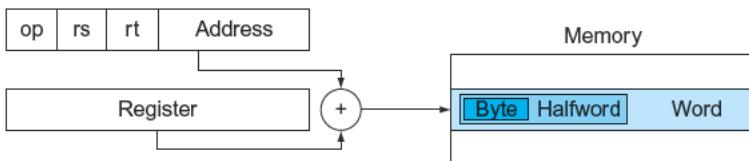
1. Immediate addressing



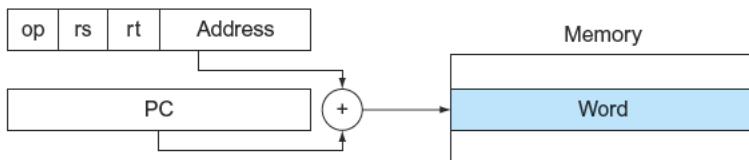
2. Register addressing



3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing

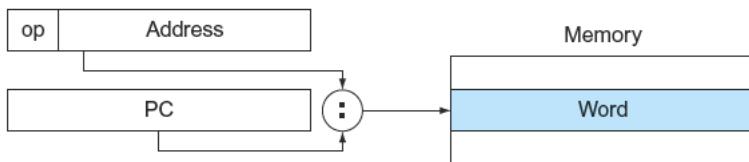


FIGURE 2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction 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. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (addi) and register (add) addressing.

Although we show MIPS as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions (see [Appendix E](#) and [Section 2.18](#)). These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in such a way that is able to move software compatibly upward to the next generation of architecture.

**Hardware/
Software
Interface**

Decoding Machine Language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at “core dump.” [Figure 2.19](#) 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.

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 2.19](#), 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 2.20](#):

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

The bottom portion of [Figure 2.19](#) 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, and 16 for rd (shamt is unused). [Figure 2.14](#) shows that these numbers represent registers \$a1, \$t7, and \$s0. Now we can reveal the assembly instruction:

add \$s0,\$a1,\$t7

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. unsigned	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	F1Pt						
3(011)								
4(100)	load byte	load half	lw1	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	sw1	store word			swr	
6(110)	load linked word	lwcl						
7(111)	store cond. word	swcl						
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	sraw
1(001)	jump register 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.	set l.t. unsigned				
6(110)								
7(111)								

FIGURE 2.19 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 floating point value in row 2, column 1 is defined in Figure 3.18 in Chapter 3. Bltz/gez is the opcode for four instructions found in Appendix A: bltz, bgez, bltzal, and bgezal. This chapter 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 are 32 bits long
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 2.20 MIPS instruction formats.

Figure 2.20 shows all the MIPS instruction formats. Figure 2.1 on page 64 shows the MIPS assembly language revealed in this chapter. The remaining hidden portion of MIPS instructions deals mainly with arithmetic and real numbers, which are covered in the next chapter.

Check Yourself

- I. 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
- II. 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
- III. 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

2.11

Parallelism and Instructions: Synchronization

Parallel execution is easier when tasks are independent, but often they need to cooperate. Cooperation usually means some tasks are writing new values that others must read. To know when a task is finished writing so that it is safe for another to read, the tasks need to synchronize. If they don't synchronize, there is a danger of a **data race**, where the results of the program can change depending on how events happen to occur.

For example, recall the analogy of the eight reporters writing a story on page 44 of Chapter 1. Suppose one reporter needs to read all the prior sections before writing a conclusion. Hence, he or she must know when the other reporters have finished their sections, so that there is no danger of sections being changed afterwards. That is, they had better synchronize the writing and reading of each section so that the conclusion will be consistent with what is printed in the prior sections.

In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. In this section, we focus on the implementation of *lock* and *unlock* synchronization operations. Lock and unlock can be used straightforwardly to create regions where only a single processor can operate, called a *mutual exclusion*, as well as to implement more complex synchronization mechanisms.

The critical ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to *atomically* read and modify a memory location. That is, nothing else can interpose itself between the read and the write of the memory location. Without such a capability, the cost of building basic synchronization primitives will be high and will increase unreasonably as the processor count increases.

There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky.

Let's start with one such hardware primitive and show how it can be used to build a basic synchronization primitive. One typical operation for building synchronization operations is the *atomic exchange* or *atomic swap*, which interchanges a value in a register for a value in memory.

To see how to use this to build a basic synchronization primitive, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had already claimed access, and 0 otherwise. In the latter



PARALLELISM

data race Two memory accesses form a data race if they are from different threads to same location, at least one is a write, and they occur one after another.

case, the value is also changed to 1, preventing any competing exchange in another processor from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: this race is broken, since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the hardware. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

Implementing a single atomic memory operation introduces some challenges in the design of the processor, since it requires both a memory read and a write in a single, uninterruptible instruction.

An alternative is to have a pair of instructions in which the second instruction returns a value showing whether the pair of instructions was executed as if the pair were atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair.

In MIPS this pair of instructions includes a special load called a *load linked* and a special store called a *store conditional*. These instructions are used in sequence: if the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. The store conditional is defined to both store the value of a (presumably different) register in memory *and* to change the value of that register to a 1 if it succeeds and to a 0 if it fails. Since the load linked returns the initial value, and the store conditional returns 1 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of \$s1:

```
again: addi $t0,$zero,1      ;copy locked value
      ll    $t1,0($s1)       ;load linked
      sc    $t0,0($s1)       ;store conditional
      beq   $t0,$zero,again  ;branch if store fails
      add   $s4,$zero,$t1     ;put load value in $s4
```

Any time a processor intervenes and modifies the value in memory between the `ll` and `sc` instructions, the `sc` returns 0 in `$t0`, causing the code sequence to try again. At the end of this sequence the contents of `$s4` and the memory location specified by `$s1` have been atomically exchanged.

Elaboration: Although it was presented for multiprocessor synchronization, atomic exchange is also useful for the operating system in dealing with multiple processes in a single processor. To make sure nothing interferes in a single processor, the store conditional also fails if the processor does a context switch between the two instructions (see Chapter 5).

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives, such as *atomic compare and swap* or *atomic fetch-and-increment*, which are used in some parallel programming models. These involve more instructions between the `ll` and the `sc`, but not too many.

Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing which instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the `sc` because of repeated page faults. In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

When do you use primitives like load linked and store conditional?

1. When cooperating threads of a parallel program need to synchronize to get proper behavior for reading and writing shared data
2. When cooperating processes on a uniprocessor need to synchronize for reading and writing shared data

Check Yourself

2.12

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 2.21 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.

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 million-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 well as an assembly language expert, and sometimes even better for large programs.

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

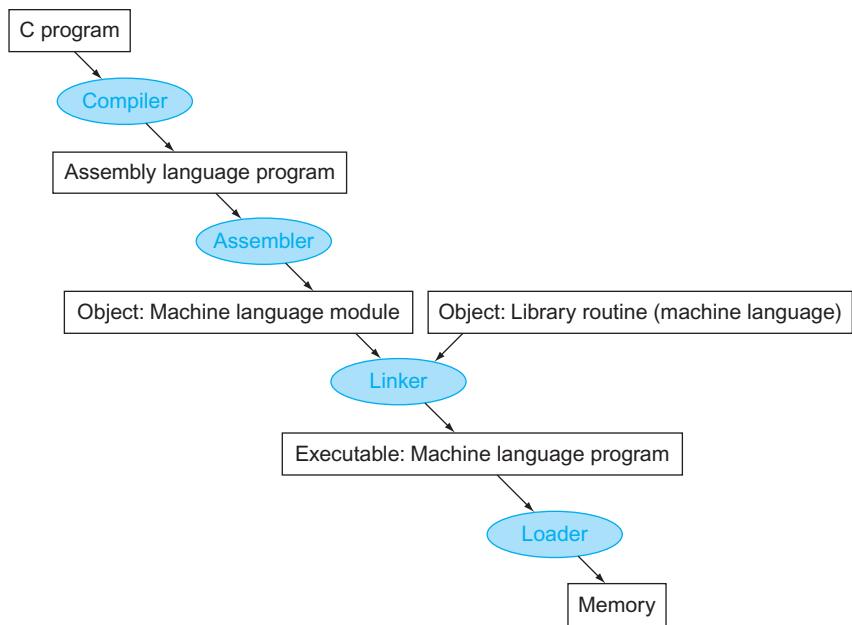


FIGURE 2.21 A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. 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.

Assembler

Since assembly language is an 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**.

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 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
```

pseudoinstruction

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

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 95. 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 be sure to get best performance, however, study the real MIPS instructions found in [Figures 2.1 and 2.19](#).

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.

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 symbols and addresses.

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 both *static data*, which is allocated throughout the program, and *dynamic data*, which can grow or shrink as needed by the program. See [Figure 2.13](#).)
- 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.

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

- 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.

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 is useful 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 2.13](#) on page 104 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 would 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, that still have unresolved addresses and hence result in object files.

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

executable file

A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A “stripped executable” does not contain that information. Relocation information may be included for the loader.

Linking Object Files

EXAMPLE

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 2.13](#) on page 104, 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
	$0040\ 0000_{hex}$	<code>lw \$a0, 8000_{hex} (\$gp)</code>
	$0040\ 0004_{hex}$	<code>jal 40 0100_{hex}</code>

	$0040\ 0100_{hex}$	<code>sw \$a1, 8020_{hex} (\$gp)</code>
	$0040\ 0104_{hex}$	<code>jal 40 0000_{hex}</code>

Data segment	Address	
	$1000\ 0000_{hex}$	(X)

	$1000\ 0020_{hex}$	(Y)

[Figure 2.13](#) also shows 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 `jal`s 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 2.13](#) 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 $40\ 0000_{hex}$. 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).

Elaboration: Recall that MIPS instructions are word aligned, so `jal` drops the right two bits to increase the instruction's address range. Thus, it uses 26 bits to create a 28-bit byte address. Hence, the actual address in the lower 26 bits of the `jal` instruction in this example is $10\ 0040_{\text{hex}}$, rather than $40\ 0100_{\text{hex}}$.

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. The **loader** 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 all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program; for example, the standard C library is 2.5 MB.

Virtually every problem in computer science can be solved by another level of indirection.

David Wheeler

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.

dynamically linked libraries (DLLs) Library routines that are linked to a program during execution.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus only 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.

Like many innovations in our field, this trick relies on a level of indirection. Figure 2.22 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

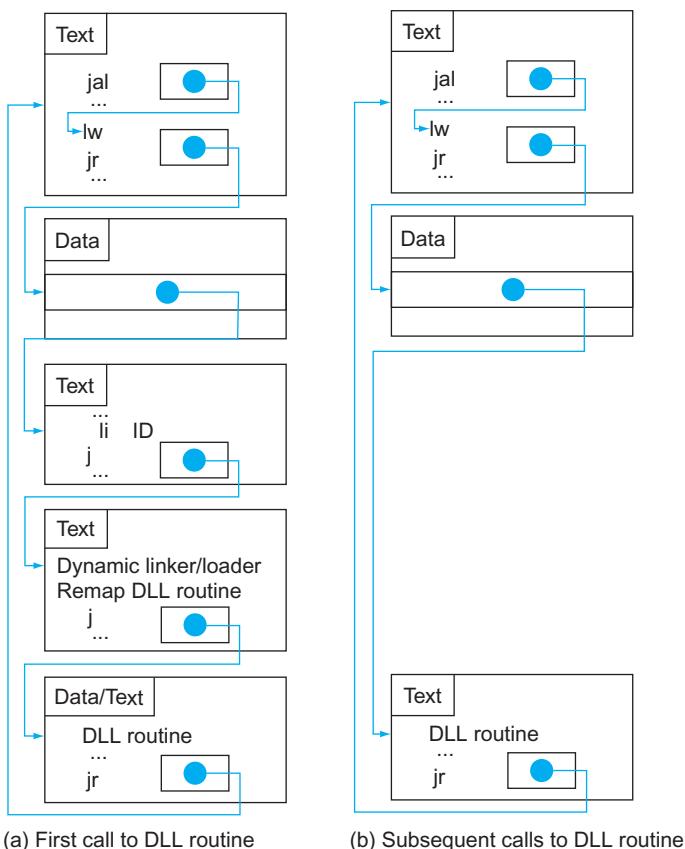


FIGURE 2.22 **Dynamically linked library via lazy procedure linkage.** (a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 5, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

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, the call to the library routine 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 dynamically linked libraries, and it is also the default when 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 run safely on any computer, even if it might slow execution time.

Figure 2.23 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 (see [Section 2.15](#)). This instruction set is designed to be close to the Java language so that this compilation 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.

Java bytecode

Instruction from an instruction set designed to interpret Java programs.

Java Virtual Machine (JVM)

The program that interprets Java bytecodes.

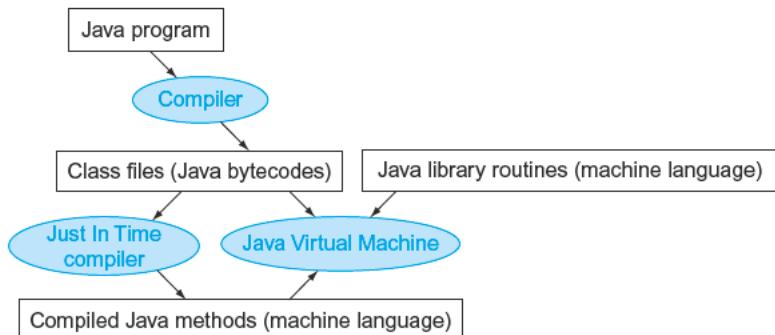


FIGURE 2.23 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the *Java Virtual Machine (JVM)*. The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

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.

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

The downside of interpretation is lower 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.

As computers get faster so that compilers can do more, and as researchers invent better ways to compile Java on the fly, the performance gap between Java and C or C++ is closing.  [Section 2.15](#) goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

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.

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

2.13

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.

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

FIGURE 2.24 A C procedure that swaps two locations in memory. This subsection uses this procedure in a sorting example.

The Procedure swap

Let's start with the code for the procedure `swap` in Figure 2.24. 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 pages 98–99, 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 100). This register allocation corresponds to the variable declarations in the first part of the `swap` procedure in Figure 2.24.

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 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 via a shift left by 2:

```
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 2.25](#) each block of code with its purpose in the procedure.

Procedure body
<pre>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)</pre>

Procedure return
<pre>jr \$ra # return to calling routine</pre>

FIGURE 2.25 MIPS assembly code of the procedure swap In [Figure 2.24](#).

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 2.26](#) shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

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

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`.

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 124.) 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 $i \geq n$. The set on less than instruction sets register \$t0 to 1 if $\$s0 < \$a1$ and to 0 otherwise. Since we want to test if $\$s0 \geq \$a1$, we branch if register \$t0 is 0. This test takes two instructions:

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

Voila! (The exercises explore 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, $j \geq 0$.

The second test exits if $v[j] > v[j + 1]$ is *not* true, or exits if $v[j] \leq v[j + 1]$. 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 $v[j] \leq v[j + 1]$ is the same as $v[j + 1] \geq v[j]$, so the two instructions of the exit test are

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

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

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

Combining the pieces, 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 \geq $t3
            beq $t0, $zero, exit2 # go to exit2 if $t4 \geq $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 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
```

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 2.27](#), 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, nine lines of the `sort` procedure in C became 35 lines in the MIPS assembly language.

Elaboration: One optimization that works with this example is *procedure inlining*. 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 5.

Saving registers		
	<pre>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</pre>	
Procedure body		
Move parameters	<pre>move \$s2, \$a0 # copy parameter \$a0 into \$s2 (save \$a0) move \$s3, \$a1 # copy parameter \$a1 into \$s3 (save \$a1)</pre>	
Outer loop	<pre> move \$s0, \$zero# i = 0 for1tst: slt \$t0, \$s0,\$s3 #reg\$t0=0if\$s0<\$s3(i<n) beq \$t0, \$zero, exit1#go to exit1 if \$s0 < \$s3 (i < n)</pre>	
Inner loop	<pre> addi \$s1, \$s0, -1# j = i - 1 for2tst: slti \$t0, \$s1,0 #reg\$t0=1if\$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</pre>	
Pass parameters and call	<pre> 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 2.25</pre>	
Inner loop	<pre> addi \$s1, \$s1, -1# j -= 1 j for2tst # jump to test of inner loop</pre>	
Outer loop	<pre>exit2: addi \$s0, \$s0, 1 # i += 1 j for1tst # jump to test of outer loop</pre>	
Restoring registers		
	<pre>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</pre>	
Procedure return		
	<pre>jr \$ra # return to calling routine</pre>	

FIGURE 2.27 MIPS assembly version of procedure sort In Figure 2.26.

Understanding Program Performance

Figure 2.28 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 2.29 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 JIT compiler makes Java 2.1 times *faster* than the unoptimized C and within a factor of 1.13 of the highest optimized C code. (Section 2.15 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 a performance increases by 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 times faster than the unoptimized C code versus 2.41 times faster).

Elaboration: 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.

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 2.28 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. 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	01	2.37	1.50	1562
	Compiler	02	2.38	1.50	1555
	Compiler	03	2.41	1.91	1955
Java	Interpreter	-	0.12	0.05	1050
	JIT compiler	-	2.13	0.29	338

FIGURE 2.29 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 in Figure 2.28. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

2.14 Arrays versus Pointers

A challenge for any new C 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 2.30 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`.

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]
```

```

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 2.30 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 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.

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; ANSI C requires a test of size before the loop, but we'll skip that legality here.)

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 uses 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:

```
sll $t1,$a1,2          # $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
      sll $t1,$a1,2          # $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):

<pre> 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 </pre>	<pre> 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 </pre>
--	---

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 moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from 6 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). [Section 2.15](#) describes these two and many other optimizations.

Elaboration: As mentioned earlier, a 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.

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

Understanding Program Performance



Advanced Material: Compiling C and Interpreting Java

This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section is for readers interested in seeing how an **object oriented language** like Java executes on a MIPS architecture. It shows the Java byte-codes used for interpretation and the MIPS code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java Virtual Machine and JIT compilers.

The rest of [Section 2.15](#) can be found online.

object oriented language

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

2.16

Real Stuff: ARMv7 (32-bit) Instructions

ARM is the most popular instruction set architecture for embedded devices, with more than 9 billion devices in 2011 using ARM, and recent growth has been 2 billion per year. Standing originally for the Acorn RISC Machine, later changed to Advanced RISC Machine, ARM came out the same year as MIPS and followed similar philosophies. [Figure 2.31](#) lists the similarities. The principal difference is that MIPS has more registers and ARM has more addressing modes.

There is a similar core of instruction sets for arithmetic-logical and data transfer instructions for MIPS and ARM, as [Figure 2.32](#) shows.

Addressing Modes

[Figure 2.33](#) shows the data addressing modes supported by ARM. Unlike MIPS, ARM does not reserve a register to contain 0. Although MIPS has just three simple data addressing modes (see [Figure 2.18](#)), ARM has nine, including fairly complex calculations. For example, ARM has an addressing mode that can shift one register

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

FIGURE 2.31 Similarities in ARM and MIPS Instruction sets.

	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl ¹	sllv, sll
	Shift right logical	lsr ¹	srlv, srl
	Shift right arithmetic	asr ¹	sra, sra
	Compare	cmp, cmn, tst, teq	slt/i, slt/iu
Data transfer	Load byte signed	ldr sb	lb
	Load byte unsigned	ldr b	lbu
	Load halfword signed	ldr sh	lh
	Load halfword unsigned	ldr h	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

FIGURE 2.32 ARM register-register and data transfer instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. If there are several choices of instructions equivalent to the MIPS core, they are separated by commas. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 1 are just a variation of a move instruction, such as lsr¹. Note that ARM has no divide instruction.

by any amount, add it to the other registers to form the address, and then update one register with this new address.

Addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

FIGURE 2.33 Summary of data addressing modes. ARM has separate register indirect and register + offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.

Compare and Conditional Branch

MIPS uses the contents of registers to evaluate conditional branches. ARM uses the traditional four condition code bits stored in the program status word: *negative*, *zero*, *carry*, and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in a pipelined implementation. ARM uses conditional branches to test condition codes to determine all possible unsigned and signed relations.

CMP subtracts one operand from the other and the difference sets the condition codes. *Compare negative* (CMN) adds one operand to the other, and the sum sets the condition codes. TST performs logical AND on the two operands to set all condition codes but overflow, while TEQ uses exclusive OR to set the first three condition codes.

One unusual feature of ARM is that every instruction has the option of executing conditionally, depending on the condition codes. Every instruction starts with a 4-bit field that determines whether it will act as a no operation instruction (nop) or as a real instruction, depending on the condition codes. Hence, conditional branches are properly considered as conditionally executing the unconditional branch instruction. Conditional execution allows avoiding a branch to jump over a single instruction. It takes less code space and time to simply conditionally execute one instruction.

Figure 2.34 shows the instruction formats for ARM and MIPS. The principal differences are the 4-bit conditional execution field in every instruction and the smaller register field, because ARM has half the number of registers.

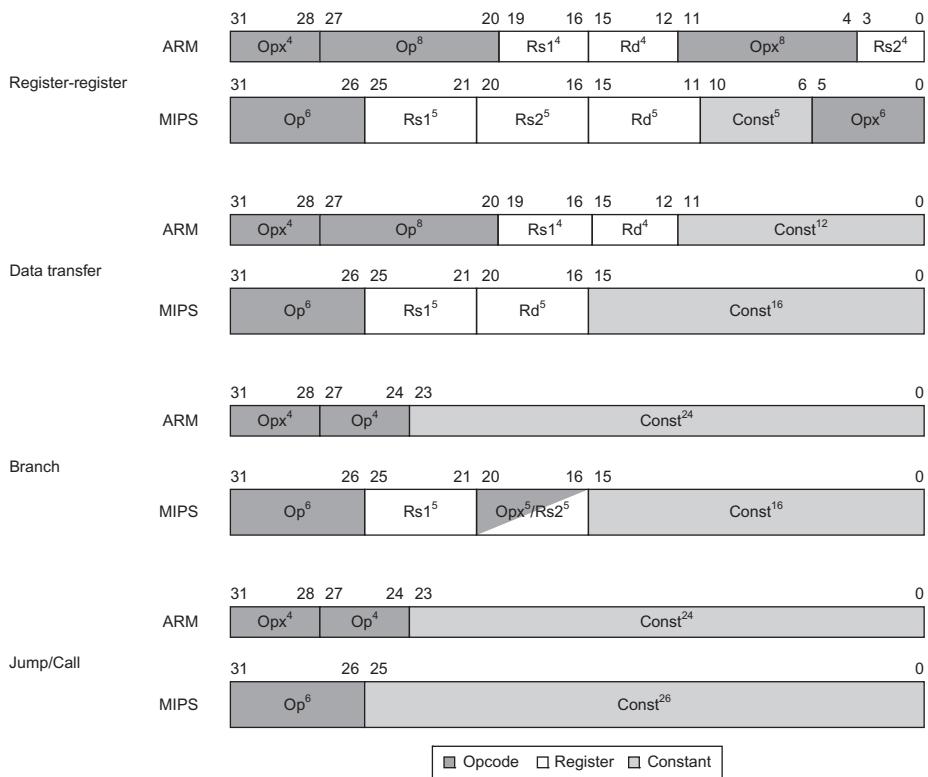


FIGURE 2.34 Instruction formats, ARM and MIPS. The differences result from whether the architecture has 16 or 32 registers.

Unique Features of ARM

Figure 2.35 shows a few arithmetic-logical instructions not found in MIPS. Since ARM does not have a dedicated register for 0, it has separate opcodes to perform some operations that MIPS can do with \$zero. In addition, ARM has support for multiword arithmetic.

ARM's 12-bit immediate field has a novel interpretation. The eight least-significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first four bits of the field multiplied by two. One advantage is that this scheme can represent all powers of two in a 32-bit word. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study.

Operand shifting is not limited to immediates. The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right.

Name	Definition	ARM	MIPS
Load immediate	Rd = Imm	mov	addi \$0,
Not	Rd = $\sim(Rs1)$	mvn	nor \$0,
Move	Rd = Rs1	mov	or \$0,
Rotate right	$Rd = Rs \text{ i} \gg i$ $Rd_{0..i-1} = Rs_{31-i..31}$	ror	
And not	Rd = Rs1 & $\sim(Rs2)$	bic	
Reverse subtract	Rd = Rs2 – Rs1	rsb, rsc	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	sbc	—

FIGURE 2.35 ARM arithmetic/logical instructions not found in MIPS.

ARM also has instructions to save groups of registers, called *block loads and stores*. Under control of a 16-bit mask within the instructions, any of the 16 registers can be loaded or stored into memory in a single instruction. These instructions can save and restore registers on procedure entry and return. These instructions can also be used for block memory copy, and today block copies are the most important use of such instructions.

2.17 Real Stuff: x86 Instructions

Designers of instruction sets sometimes provide more powerful operations than those found in ARM and 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.

The path toward operation complexity is thus fraught with peril. Section 2.19 demonstrates the pitfalls of complexity.

Evolution of the Intel x86

ARM and MIPS were the vision of single small groups in 1985; the pieces of these architectures fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the x86; it is the product of several independent groups who evolved the architecture over 35 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important x86 milestones.

*Beauty is altogether in
the eye of the beholder.*

Margaret Wolfe
Hungerford, *Molly Bawn*, 1877

general-purpose register (GPR)

A register that can be used for addresses or for data with virtually any instruction.

- **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.
- **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 2.21](#) and Section 3.7).
- **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 5), 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 5). 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 6) 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 6). 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 includes 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 boosted the floating-point performance of the Pentium 4, the first microprocessor to include SSE2 instructions.

- **2003:** A company other than Intel enhanced the x86 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 x86 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 four to ten new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to x86 (*legacy mode*) plus a mode that restricts user programs to x86 but allows operating systems to use AMD64 (*compatibility 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 Section 2.11). AMD added SSE3 in subsequent chips and the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.
- **2006:** Intel announces 54 new instructions as part of the SSE4 instruction set extensions. These extensions perform tweaks like sum of absolute differences, dot products for arrays of structures, sign or zero extension of narrow data to wider sizes, population count, and so on. They also added support for virtual machines (see Chapter 5).
- **2007:** AMD announces 170 instructions as part of SSE5, including 46 instructions of the base instruction set that adds three operand instructions like MIPS.
- **2011:** Intel ships the Advanced Vector Extension that expands the SSE register width from 128 to 256 bits, thereby redefining about 250 instructions and adding 128 new instructions.

This history illustrates the impact of the “golden handcuffs” of compatibility on the x86, as the existing software base at each step was too important to jeopardize with significant architectural changes.

Whatever the artistic failures of the x86, keep in mind that this instruction set largely drove the PC generation of computers and still dominates the cloud portion of the PostPC Era. Manufacturing 350M x86 chips per year may seem small compared to 9 billion ARMv7 chips, but many companies would love to control such a market. 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 x86 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, 32-bit, and 64-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

x86 Registers and Data Addressing Modes

The registers of the 80386 show the evolution of the instruction set ([Figure 2.36](#)). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits, 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 and ARMv7 twice as many.

[Figure 2.37](#) shows the arithmetic, logical, and data transfer instructions are two-operand instructions. There are two important differences here. The x86 arithmetic and logical instructions must have one operand act as both a source and a destination; ARMv7 and MIPS allow 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 ARMv7 and MIPS.

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 2.38](#) shows the x86 addressing modes and which GPRs cannot be used with each mode, as well as how to get the same effect using MIPS instructions.

x86 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 x86. (AMD64 adds 64-

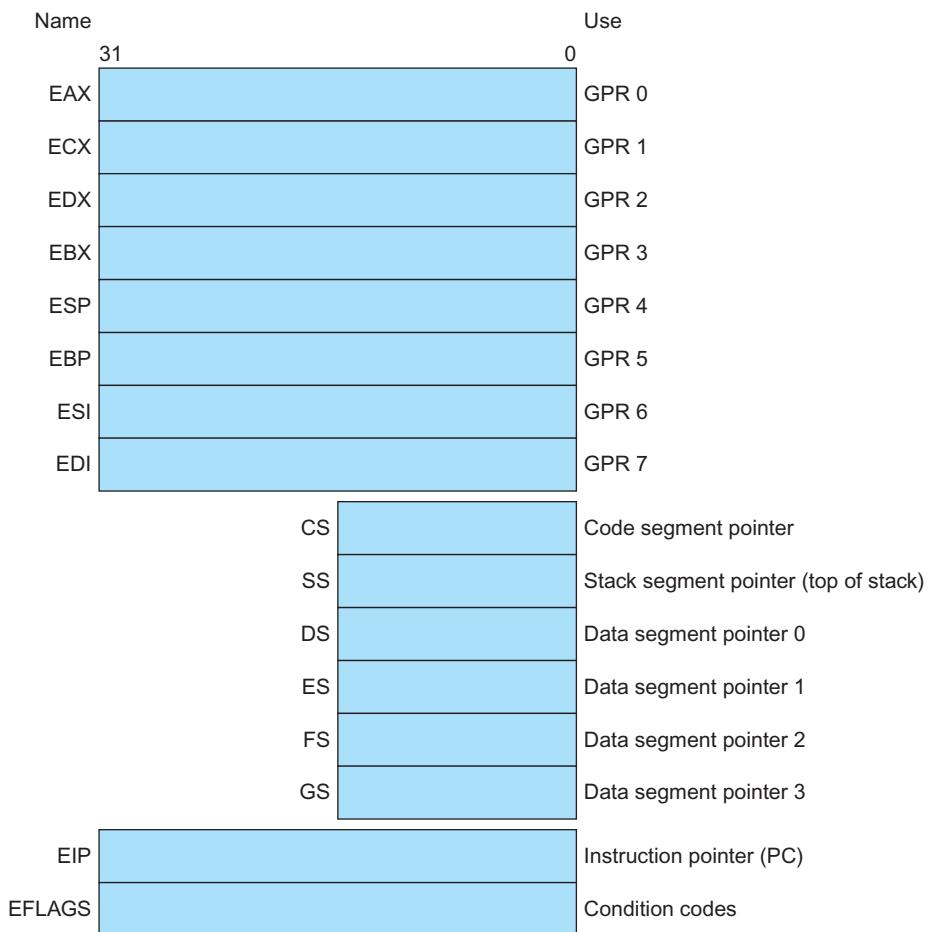


FIGURE 2.36 The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

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

FIGURE 2.37 Instruction types for the arithmetic, logical, and data transfer instructions. The x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediate may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.36 (not EIP or EFLAGS).

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

FIGURE 2.38 x86 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in ARM or MIPS, is included to avoid the multiplies by 4 (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.25 and 2.27). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. A 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.)

bit addresses and data, called *quad words*; we'll stick to the 80386 in this section.) 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 provided a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominates 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 synchronization (see Section 2.11), 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 x86 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 2.39](#) shows some typical x86 instructions and their functions.

Conditional branches on the x86 are based on *condition codes* or *flags*, like ARMv7. 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. PC-

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 2.39 Some typical x86 instructions and their functions. A list of frequent operations appears in [Figure 2.40](#). The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

relative branch addresses must be specified in the number of bytes, since unlike ARMv7 and MIPS, 80386 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the x86 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the fallacy on page 159).

[Figure 2.40](#) lists some of the integer x86 instructions. Many of the instructions are available in both byte and word formats.

x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 80386 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 15 bytes.

[Figure 2.41](#) shows the instruction format for several of the example instructions in [Figure 2.39](#). The opcode byte usually contains a bit saying whether the operand is 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

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
move	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 eight 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
lod\$	Loads a byte, word, or doubleword of a string into the EAX register

FIGURE 2.40 Some typical operations on the x86. 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.

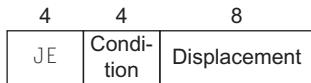
of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled “sc, index, base.”

Figure 2.42 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to understand fully 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.

x86 Conclusion

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the x86 is more difficult to build than computers like ARMv7 and MIPS, but the large market meant in the PC Era that AMD and Intel could afford more resources

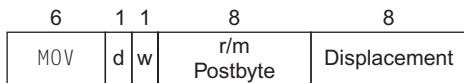
a. JE EIP + displacement



b. CALL



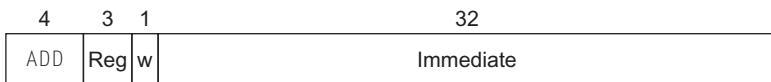
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



FIGURE 2.41 Typical x86 instruction formats. Figure 2.42 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a 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 15 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.

to help overcome the added complexity. What the x86 lacks in style, it made up for in market size, making it beautiful from the right perspective.

Its saving grace is that the most frequently used x86 architectural components are not too difficult to implement, as AMD and Intel have demonstrated by rapidly improving performance of integer programs since 1978. To get that performance,

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	= <i>(sib)</i>	SI+disp8	(<i>sib</i>)+disp8	SI+disp8	(<i>sib</i>)+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 2.42 The encoding of the first address specifier of the x86: 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 1) r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; 2) r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and 3) r/m = 4 in 32-bit mode when mod does not equal 3, where (*sib*) means use the scaled index mode shown in Figure 2.38. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

compilers must avoid the portions of the architecture that are hard to implement fast.

In the PostPC Era, however, despite considerable architectural and manufacturing expertise, x86 has not yet been competitive in the personal mobile device.

2.18

Real Stuff: ARMv8 (64-bit) Instructions

Of the many potential problems in an instruction set, the one that is almost impossible to overcome is having too small a memory address. While the x86 was successfully extended first to 32-bit addresses and then later to 64-bit addresses, many of its brethren were left behind. For example, the 16-bit address MOStek 6502 powered the Apple II, but even given this headstart with the first commercially successful personal computer, its lack of address bits condemned it to the dustbin of history.

ARM architects could see the writing on the wall of their 32-bit address computer, and began design of the 64-bit address version of ARM in 2007. It was finally revealed in 2013. Rather than some minor cosmetic changes to make all the registers 64 bits wide, which is basically what happened to the x86, ARM did a complete overhaul. The good news is that if you know MIPS it will be very easy to pick up ARMv8, as the 64-bit version is called.

First, as compared to MIPS, ARM dropped virtually all of the unusual features of v7:

- There is no conditional execution field, as there was in nearly every instruction in v7.

- The immediate field is simply a 12 bit constant, rather than essentially an input to a function that produces a constant as in v7.
- ARM dropped Load Multiple and Store Multiple instructions.
- The PC is no longer one of the registers, which resulted in unexpected branches if you wrote to it.

Second, ARM added missing features that are useful in MIPS

- V8 has 32 general-purpose registers, which compiler writers surely love. Like MIPS, one register is hardwired to 0, although in load and store instructions it instead refers to the stack pointer.
- Its addressing modes work for all word sizes in ARMv8, which was not the case in ARMv7.
- It includes a divide instruction, which was omitted from ARMv7.
- It adds the equivalent of MIPS branch if equal and branch if not equal.

As the philosophy of the v8 instruction set is much closer to MIPS than it is to v7, our conclusion is that the main similarity between ARMv7 and ARMv8 is the name.

2.19 Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel x86 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 as fast. A third version, which uses the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times as fast than the complex move instruction.

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

At one time compilers for programming languages produced naïve 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 understand the concepts in Chapters 4 and 5 thoroughly (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is another 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 old 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 does.

Even if writing by hand resulted in faster code, the dangers of writing in assembly language are the 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 or Java. 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.

Fallacy: The importance of commercial binary compatibility means successful instruction sets don't change.

While backwards binary compatibility is sacrosanct, [Figure 2.43](#) shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 35-year lifetime!

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 2.12](#), the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

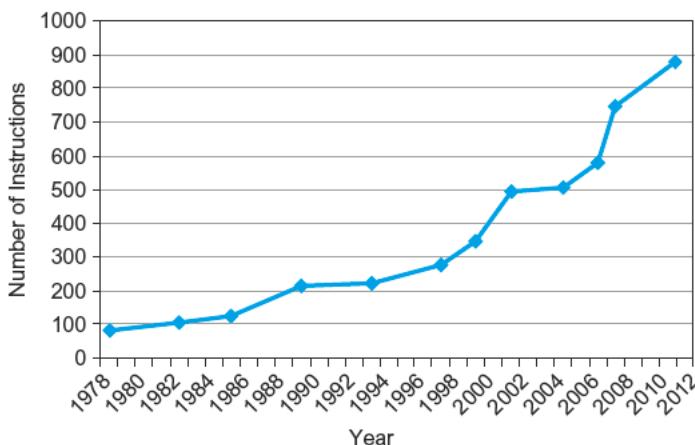


FIGURE 2.43 Growth of x86 Instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

2.20 Concluding Remarks

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. As illustrated in this chapter, three 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. *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.

Less is more.

Robert Browning,
Andrea del Sarto, 1855



COMMON CASE FAST

We also saw the great idea of making the **common cast fast** applied to instruction sets as well as computer architecture. Examples of making the common MIPS case fast include PC-relative addressing for conditional branches and immediate addressing for larger constant operands.

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 2.44](#) lists the MIPS instructions we have covered

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
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I			
load byte unsigned	lbu	I	branch greater than or equal	bge	I
store byte	sb	I			
load linked	ll	I			
store conditional	sc	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			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

FIGURE 2.44 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) describes the full MIPS architecture. [Figure 2.1](#) shows more details of the MIPS architecture revealed in this chapter. The information given here is also found in Columns 1 and 2 of the MIPS Reference Data Card at the front of the book.

so far, both real and pseudoinstructions. Hiding details from the higher level is another example of the great idea of **abstraction**.

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

- Arithmetic instructions correspond to the operations found in assignment statements.
- Transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- Conditional branches are used in *if* statements and in loops.
- Unconditional jumps are used in procedure calls and returns and for *case/switch* statements.



These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.45 shows the popularity of each class of instructions for SPEC CPU2006. The varying popularity of instructions plays an important role in the chapters about datapath, control, and pipelining.

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statement s	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statement s	12%	4%
Conditional branch	beq, bne,slt, slti, sltiu	<i>If</i> statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, returns, and <i>case/switch</i> statements	2%	0%

FIGURE 2.45 MIPS Instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS Instructions executed by category for the average Integer and floating point SPEC CPU2006 benchmarks. Figure 3.26 in Chapter 3 shows average percentage of the individual MIPS instructions executed.

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



Historical Perspective and Further Reading

This section surveys the history of *instruction set architectures* (ISAs) 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 ARM and the x86. 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 [Section 2.21](#) is found online.

2.22

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 [5] <§2.2> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables `f`, `g`, `h`, and `i` are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.

`f = g + (h - 5);`

2.2 [5] <§2.2> For the following MIPS assembly instructions above, what is a corresponding C statement?

`add f, g, h`
`add f, i, f`

2.3 [5] <§§2.2, 2.3> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, i, and j are assigned to registers \$s0, \$s1, \$s2, \$s3, and \$s4, respectively. Assume that the base address of the arrays A and B are in registers \$s6 and \$s7, respectively.

```
B[8] = A[i-j];
```

2.4 [5] <§§2.2, 2.3> For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers \$s0, \$s1, \$s2, \$s3, and \$s4, respectively. Assume that the base address of the arrays A and B are in registers \$s6 and \$s7, respectively.

```
sll    $t0, $s0, 2      # $t0 = f * 4
add   $t0, $s6, $t0     # $t0 = &A[f]
sll    $t1, $s1, 2      # $t1 = g * 4
add   $t1, $s7, $t1     # $t1 = &B[g]
lw    $s0, 0($t0)       # f = A[f]
addi  $t2, $t0, 4
lw    $t0, 0($t2)
add   $t0, $t0, $s0
sw    $t0, 0($t1)
```

2.5 [5] <§§2.2, 2.3> For the MIPS assembly instructions in Exercise 2.4, rewrite the assembly code to minimize the number of MIPS instructions (if possible) needed to carry out the same function.

2.6 The table below shows 32-bit values of an array stored in memory.

Address	Data
24	2
38	4
32	3
36	6
40	1

2.6.1 [5] <§§2.2, 2.3> For the memory locations in the table above, write C code to sort the data from lowest to highest, placing the lowest value in the smallest memory location shown in the figure. Assume that the data shown represents the C variable called `Array`, which is an array of type `int`, and that the first number in the array shown is the first element in the array. Assume that this particular machine is a byte-addressable machine and a word consists of four bytes.

2.6.2 [5] <§§2.2, 2.3> For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of `Array` is stored in register `$s6`.

2.7 [5] <§2.3> Show how the value `0xabcdef12` would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

2.8 [5] <§2.4> Translate `0xabcdef12` into decimal.

2.9 [5] <§§2.2, 2.3> Translate the following C code to MIPS. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively. Assume that the elements of the arrays `A` and `B` are 4-byte words:

`B[8] = A[i] + A[j];`

2.10 [5] <§§2.2, 2.3> Translate the following MIPS code to C. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. Assume that the base address of the arrays `A` and `B` are in registers `$s6` and `$s7`, respectively.

```
addi $t0, $s6, 4
add $t1, $s6, $0
sw $t1, 0($t0)
lw $t0, 0($t0)
add $s0, $t1, $t0
```

2.11 [5] <§§2.2, 2.5> For each MIPS instruction, show the value of the opcode (OP), source register (RS), and target register (RT) fields. For the I-type instructions, show the value of the immediate field, and for the R-type instructions, show the value of the destination register (RD) field.

2.12 Assume that registers \$s0 and \$s1 hold the values 0x80000000 and 0xD0000000, respectively.

2.12.1 [5] <§2.4> What is the value of \$t0 for the following assembly code?

```
add $t0, $s0, $s1
```

2.12.2 [5] <§2.4> Is the result in \$t0 the desired result, or has there been overflow?

2.12.3 [5] <§2.4> For the contents of registers \$s0 and \$s1 as specified above, what is the value of \$t0 for the following assembly code?

```
sub $t0, $s0, $s1
```

2.12.4 [5] <§2.4> Is the result in \$t0 the desired result, or has there been overflow?

2.12.5 [5] <§2.4> For the contents of registers \$s0 and \$s1 as specified above, what is the value of \$t0 for the following assembly code?

```
add $t0, $s0, $s1  
add $t0, $t0, $s0
```

2.12.6 [5] <§2.4> Is the result in \$t0 the desired result, or has there been overflow?

2.13 Assume that \$s0 holds the value 128_{ten}.

2.13.1 [5] <§2.4> For the instruction add \$t0, \$s0, \$s1, what is the range(s) of values for \$s1 that would result in overflow?

2.13.2 [5] <§2.4> For the instruction sub \$t0, \$s0, \$s1, what is the range(s) of values for \$s1 that would result in overflow?

2.13.3 [5] <§2.4> For the instruction sub \$t0, \$s1, \$s0, what is the range(s) of values for \$s1 that would result in overflow?

2.14 [5] <§§2.2, 2.5> Provide the type and assembly language instruction for the following binary value: 0000 0010 0001 0000 1000 0000 0010 0000_{two}

2.15 [5] <§§2.2, 2.5> Provide the type and hexadecimal representation of following instruction: sw \$t1, 32(\$t2)

2.16 [5] <§2.5> Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

op=0, rs=3, rt=2, rd=3, shamt=0, funct=34

2.17 [5] <§2.5> Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

op=0x23, rs=1, rt=2, const=0x4

2.18 Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.

2.18.1 [5] <§2.5> How would this affect the size of each of the bit fields in the R-type instructions?

2.18.2 [5] <§2.5> How would this affect the size of each of the bit fields in the I-type instructions?

2.18.3 [5] <§§2.5, 2.10> How could each of the two proposed changes decrease the size of an MIPS assembly program? On the other hand, how could the proposed change increase the size of an MIPS assembly program?

2.19 Assume the following register contents:

\$t0 = 0xFFFFFFFF, \$t1 = 0x12345678

2.19.1 [5] <§2.6> For the register values shown above, what is the value of \$t2 for the following sequence of instructions?

sll \$t2, \$t0, 4
or \$t2, \$t2, \$t1

2.19.2 [5] <§2.6> For the register values shown above, what is the value of \$t2 for the following sequence of instructions?

sll \$t2, \$t0, 4
andi \$t2, \$t2, -1

2.19.3 [5] <§2.6> For the register values shown above, what is the value of \$t2 for the following sequence of instructions?

srl \$t2, \$t0, 3
andi \$t2, \$t2, 0xFFFF

2.20 [5] <§2.6> Find the shortest sequence of MIPS instructions that extracts bits 16 down to 11 from register \$t0 and uses the value of this field to replace bits 31 down to 26 in register \$t1 without changing the other 26 bits of register \$t1.

2.21 [5] <§2.6> Provide a minimal set of MIPS instructions that may be used to implement the following pseudoinstruction:

```
not $t1, $t2      // bit-wise invert
```

2.22 [5] <§2.6> For the following C statement, write a minimal sequence of MIPS assembly instructions that does the identical operation. Assume \$t1 = A, \$t2 = B, and \$s1 is the base address of C.

```
A = C[0] << 4;
```

2.23 [5] <§2.7> Assume \$t0 holds the value 0x00101000. What is the value of \$t2 after the following instructions?

```
slt $t2, $0, $t0
bne $t2, $0, ELSE
j DONE
ELSE: addi $t2, $t2, 2
DONE:
```

2.24 [5] <§2.7> Suppose the program counter (PC) is set to 0x2000 0000. Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as 0x4000 0000? Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?

2.25 The following instruction is not included in the MIPS instruction set:

```
rpt $t2, loop # if(R[rs]>0) R[rs]=R[rs]-1, PC=PC+4+BranchAddr
```

2.25.1 [5] <§2.7> If this instruction were to be implemented in the MIPS instruction set, what is the most appropriate instruction format?

2.25.2 [5] <§2.7> What is the shortest sequence of MIPS instructions that performs the same operation?

2.26 Consider the following MIPS loop:

```

LOOP: slt $t2, $0, $t1
      beq $t2, $0, DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j LOOP
      
```

DONE:

2.26.1 [5] <§2.7> Assume that the register \$t1 is initialized to the value 10. What is the value in register \$s2 assuming \$s2 is initially zero?

2.26.2 [5] <§2.7> For each of the loops above, write the equivalent C code routine. Assume that the registers \$s1, \$s2, \$t1, and \$t2 are integers A, B, i, and temp, respectively.

2.26.3 [5] <§2.7> For the loops written in MIPS assembly above, assume that the register \$t1 is initialized to the value N. How many MIPS instructions are executed?

2.27 [5] <§2.7> Translate the following C code to MIPS assembly code. Use a minimum number of instructions. Assume that the values of a, b, i, and j are in registers \$s0, \$s1, \$t0, and \$t1, respectively. Also, assume that register \$s2 holds the base address of the array D.

```

for(i=0; i<a; i++)
    for(j=0; j<b; j++)
        D[4*j] = i + j;
    
```

2.28 [5] <§2.7> How many MIPS instructions does it take to implement the C code from Exercise 2.27? If the variables a and b are initialized to 10 and 1 and all elements of D are initially 0, what is the total number of MIPS instructions that is executed to complete the loop?

2.29 [5] <§2.7> Translate the following loop into C. Assume that the C-level integer i is held in register \$t1, \$s2 holds the C-level integer called result, and \$s0 holds the base address of the integer MemArray.

```

addi $t1, $0, $0
LOOP: lw $s1, 0($s0)
      add $s2, $s2, $s1
      addi $s0, $s0, 4
      
```

```
addi $t1, $t1, 1  
slti $t2, $t1, 100  
bne $t2, $s0, LOOP
```

2.30 [5] <§2.7> Rewrite the loop from Exercise 2.29 to reduce the number of MIPS instructions executed.

2.31 [5] <§2.8> Implement the following C code in MIPS assembly. What is the total number of MIPS instructions needed to execute the function?

```
int fib(int n){  
    if (n==0)  
        return 0;  
    else if (n == 1)  
        return 1;  
    else  
        return fib(n-1) + fib(n-2);
```

2.32 [5] <§2.8> Functions can often be implemented by compilers “in-line.” An in-line function is when the body of the function is copied into the program space, allowing the overhead of the function call to be eliminated. Implement an “in-line” version of the C code above in MIPS assembly. What is the reduction in the total number of MIPS assembly instructions needed to complete the function? Assume that the C variable n is initialized to 5.

2.33 [5] <§2.8> For each function call, show the contents of the stack after the function call is made. Assume the stack pointer is originally at address 0x7fffffc, and follow the register conventions as specified in Figure 2.11.

2.34 Translate function f into MIPS assembly language. If you need to use registers \$t0 through \$t7, use the lower-numbered registers first. Assume the function declaration for func is “int func(int a, int b);”. The code for function f is as follows:

```
int f(int a, int b, int c, int d){  
    return func(func(a,b),c+d);  
}
```

2.35 [5] <§2.8> Can we use the tail-call optimization in this function? If no, explain why not. If yes, what is the difference in the number of executed instructions in *f* with and without the optimization?

2.36 [5] <§2.8> Right before your function *f* from Exercise 2.34 returns, what do we know about contents of registers \$t5, \$s3, \$ra, and \$sp? Keep in mind that we know what the entire function *f* looks like, but for function *func* we only know its declaration.

2.37 [5] <§2.9> Write a program in MIPS assembly language to convert an ASCII number string containing positive and negative integer decimal strings, to an integer. Your program should expect register \$a0 to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register \$v0. If a non-digit character appears anywhere in the string, your program should stop with the value -1 in register \$v0. For example, if register \$a0 points to a sequence of three bytes 50ten, 52ten, 0ten (the null-terminated string “24”), then when the program stops, register \$v0 should contain the value 24_{ten}.

2.38 [5] <§2.9> Consider the following code:

```
lbu $t0, 0($t1)  
sw $t0, 0($t2)
```

Assume that the register \$t1 contains the address 0x1000 0000 and the register \$t2 contains the address 0x1000 0010. Note the MIPS architecture utilizes big-endian addressing. Assume that the data (in hexadecimal) at address 0x1000 0000 is: 0x11223344. What value is stored at the address pointed to by register \$t2?

2.39 [5] <§2.10> Write the MIPS assembly code that creates the 32-bit constant 0010 0000 0000 0001 0100 1001 0010 0100_{two} and stores that value to register \$t1.

2.40 [5] <§§2.6, 2.10> If the current value of the PC is 0x00000000, can you use a single jump instruction to get to the PC address as shown in Exercise 2.39?

2.41 [5] <§§2.6, 2.10> If the current value of the PC is 0x00000600, can you use a single branch instruction to get to the PC address as shown in Exercise 2.39?

2.42 [5] <§§2.6, 2.10> If the current value of the PC is 0xFFFFf000, can you use a single branch instruction to get to the PC address as shown in Exercise 2.39?

2.43 [5] <§2.11> Write the MIPS assembly code to implement the following C code:

```
lock(lk);
shvar=max(shvar,x);
unlock(lk);
```

Assume that the address of the lk variable is in \$a0, the address of the shvar variable is in \$a1, and the value of variable x is in \$a2. Your critical section should not contain any function calls. Use ll/sc instructions to implement the lock() operation, and the unlock() operation is simply an ordinary store instruction.

2.44 [5] <§2.11> Repeat Exercise 2.43, but this time use ll/sc to perform an atomic update of the shvar variable directly, without using lock() and unlock(). Note that in this problem there is no variable lk.

2.45 [5] <§2.11> Using your code from Exercise 2.43 as an example, explain what happens when two processors begin to execute this critical section at the same time, assuming that each processor executes exactly one instruction per cycle.

2.46 Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

2.46.1 [5] <§2.19> Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

2.46.2 [5] <§2.19> Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times?

2.47 Assume that for a given program 70% of the executed instructions are arithmetic, 10% are load/store, and 20% are branch.

2.47.1 [5] <§2.19> Given this instruction mix and the assumption that an arithmetic instruction requires 2 cycles, a load/store instruction takes 6 cycles, and a branch instruction takes 3 cycles, find the average CPI.

2.47.2 [5] <§2.19> For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

2.47.3 [5] <§2.19> For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Answers to Check Yourself

§2.2, page 66: MIPS, C, Java

§2.3, page 72: 2) Very slow

§2.4, page 79: 2) -8_{ten}

§2.5, page 87: 4) sub \$t2, \$t0, \$t1

§2.6, page 89: Both. AND with a mask pattern of 1s will leave 0s everywhere but the desired field. Shifting left by the correct amount removes the bits from the left of the field. Shifting right by the appropriate amount puts the field into the rightmost bits of the word, with 0s in the rest of the word. Note that AND leaves the field where it was originally, and the shift pair moves the field into the rightmost part of the word.

§2.7, page 96: I. All are true. II. 1).

§2.8, page 106: Both are true.

§2.9, page 111: I. 1) and 2) II. 3)

§2.10, page 120: I. 4) $+ -128K$. II. 6) a block of 256M. III. 4) $\$11$

§2.11, page 123: Both are true.

§2.12, page 132: 4) Machine independence.

This page intentionally left blank

3

*Numerical precision
is the very soul of
science.*

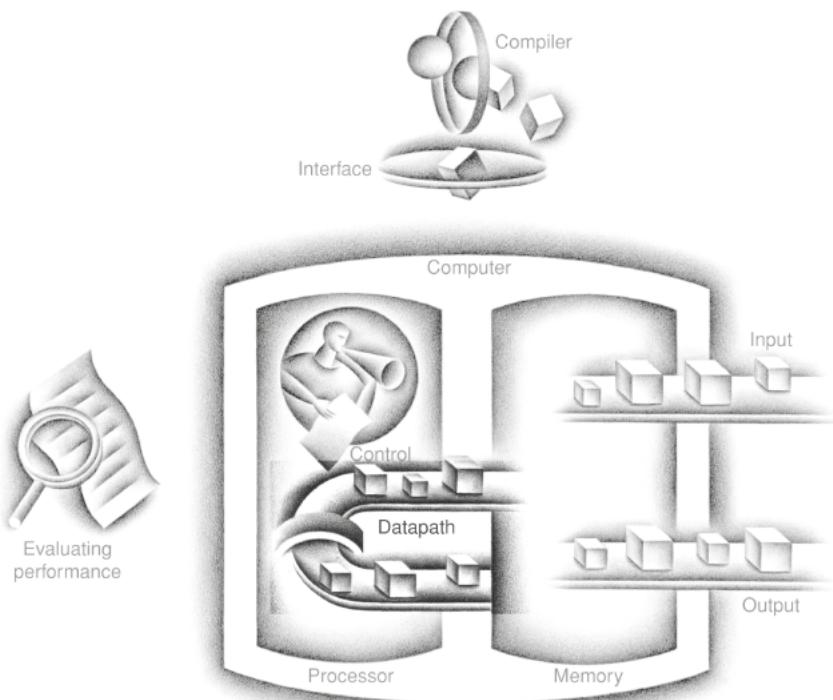
Sir D'Arcy Wentworth Thompson
On Growth and Form, 1917

Arithmetic for Computers

- 3.1 Introduction** 178
- 3.2 Addition and Subtraction** 178
- 3.3 Multiplication** 183
- 3.4 Division** 189
- 3.5 Floating Point** 196
- 3.6 Parallelism and Computer Arithmetic:
Subword Parallelism** 222
- 3.7 Real Stuff: Streaming SIMD Extensions and
Advanced Vector Extensions in x86** 224

- 3.8 Going Faster: Subword Parallelism and Matrix Multiply** 225
- 3.9 Fallacies and Pitfalls** 229
- 3.10 Concluding Remarks** 232
- 3.11 Historical Perspective and Further Reading** 236
- 3.12 Exercises** 237

The Five Classic Components of a Computer



3.1**Introduction**

Computer words are composed of bits; thus, words can be represented as binary numbers. Chapter 2 shows that integers can be represented either in decimal or binary form, but what about the other numbers that commonly occur? For example:

- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- And underlying these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel these mysteries including representation of real numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may explain quirks that you have already encountered with computers. Moreover, we show how to use this knowledge to make arithmetic-intensive programs go much faster.

3.2**Addition and Subtraction**

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

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.

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 +\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 =\quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}}
 \end{array}$$

The 4 bits to the right have all the action; Figure 3.1 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

ANSWER

Subtracting 6_{ten} from 7_{ten} can be done directly:

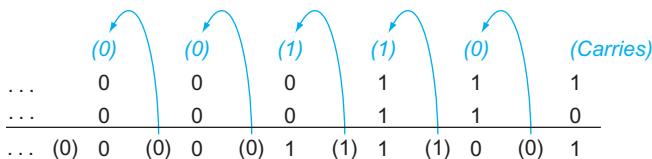


FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is $0 + 1 + 1$. This generates a 0 for this sum bit and a carry out of 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.

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 - 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

or via addition using the two's complement representation of -6 :

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 + 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{\text{two}} = -6_{\text{ten}} \\
 \hline
 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}
 \end{array}$$

Recall 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 subtraction, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $c - a = c + (-a)$ because we subtract by negating the second operand and then add. Therefore, 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.

Knowing when overflow cannot occur in addition and subtraction is all well and good, but how do we detect it when it *does* occur? 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 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. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum 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. Such a ridiculous result means a borrow occurred from the sign bit. Figure 3.2 shows the combination of operations, operands, and results that indicate an overflow.

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 3.2 Overflow conditions for addition and subtraction.

We have just seen how to detect overflow for two's complement numbers in a computer. What about overflow with 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 (`add i`), 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.

 **Appendix B** describes the hardware that performs addition and subtraction, which is called an **Arithmetic Logic Unit** or **ALU**.

Elaboration: A constant source of confusion for `addiu` is its name and what happens to its immediate field. The `u` stands for unsigned, which means addition cannot cause an overflow exception. However, the 16-bit immediate field is sign extended to 32 bits, just like `addi`, `slti`, and `sltiu`. Thus, the immediate field is signed, even if the operation is “unsigned.”

Hardware/ Software Interface

exception Also called **interrupt** on many computers. An unscheduled event that disrupts program execution; used to detect overflow.

The computer designer must decide how to handle arithmetic overflows. Although some languages like C and Java 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 4.9 covers exceptions in

more detail; Chapter 5 describes 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* (`mfcc0`) 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.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Summary

A major 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.

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas MIPS only has integer arithmetic operations on full words. As we recall from Chapter 2, MIPS does have data transfer operations for bytes and halfwords. What MIPS instructions should be generated for byte and halfword arithmetic operations?

Check Yourself

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. Load 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: One feature not generally found in general-purpose microprocessors is *saturating* operations. Saturation means that when a calculation overflows, the result is set to the largest positive number or most negative number, rather than a modulo calculation as in two's complement arithmetic. Saturation is likely what you want for media operations. For example, the volume knob on a radio set would be frustrating if, as you turned it, the volume would get continuously louder for a while and then immediately very soft. A knob with saturation would stop at the highest volume no matter how far you turned it. Multimedia extensions to standard instruction sets often offer saturating arithmetic.

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 *Elaboration* on page 89 in Chapter 2 for a description of the `xor` instruction):

```

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 3 signs !=; goto 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:  $2^{32} - \$t1 - 1$ )
slt $t3, $t3, $t2        #  $(2^{32} - \$t1 - 1) < \$t2$ 
                        #  $\Rightarrow 2^{32} - 1 < \$t1 + \$t2$ 
bne $t3,$zero,Overflow # if( $2^{32}-1 < \$t1+\$t2$ ) goto overflow

```

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 directive 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? Either you restore the old registers first, thereby destroying your return address from EPC, which 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 `$at` so that the assembler can use it as a temporary register (see *Hardware/Software Interface* in Section 2.10), compilers also abstain from using registers `$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.

Elaboration: The speed of addition is increased by determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the \log_2 of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry. The most popular is *carry lookahead*, which Section B.6 in [Appendix B](#) describes.

3.3

Multiplication

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiplication.

First, let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps of multiplication 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} :

$$\begin{array}{r}
 \text{Multiplicand} & 1000_{\text{ten}} \\
 \text{Multiplier} & \times \quad 1001_{\text{ten}} \\
 & \hline
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
 \text{Product} & \hline
 & 1001000_{\text{ten}}
 \end{array}$$

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 ($1 \times$ multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 ($0 \times$ multiplicand) in the proper place if the digit is 0.

Although the decimal example above happens 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.

*Multiplication is
vexation, Division is
as bad; The rule of
three doth puzzle me,
And practice drives me
mad.*

Anonymous,
Elizabethan manuscript,
1570

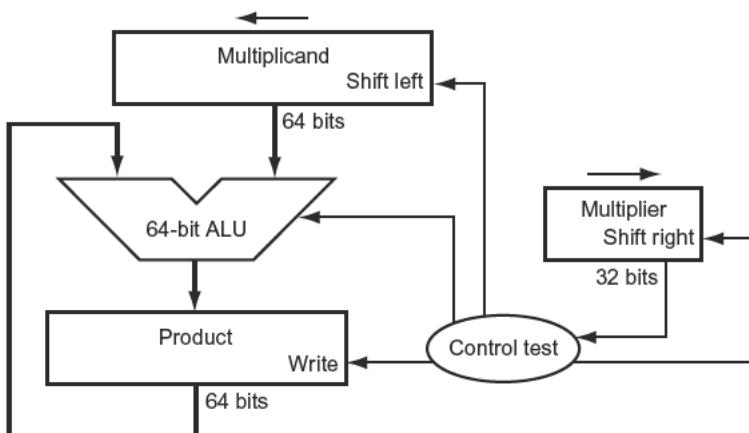


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted 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.

Sequential Version of the Multiplication Algorithm and Hardware

This design mimics the algorithm we learned in grammar school; Figure 3.3 shows the hardware. We have drawn the hardware so that data flows from top to bottom to resemble more closely 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 zero 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 3.4 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 with paper and pencil. 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 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 Section 1.10) reminds us that even a moderate frequency for a slow operation can limit performance.

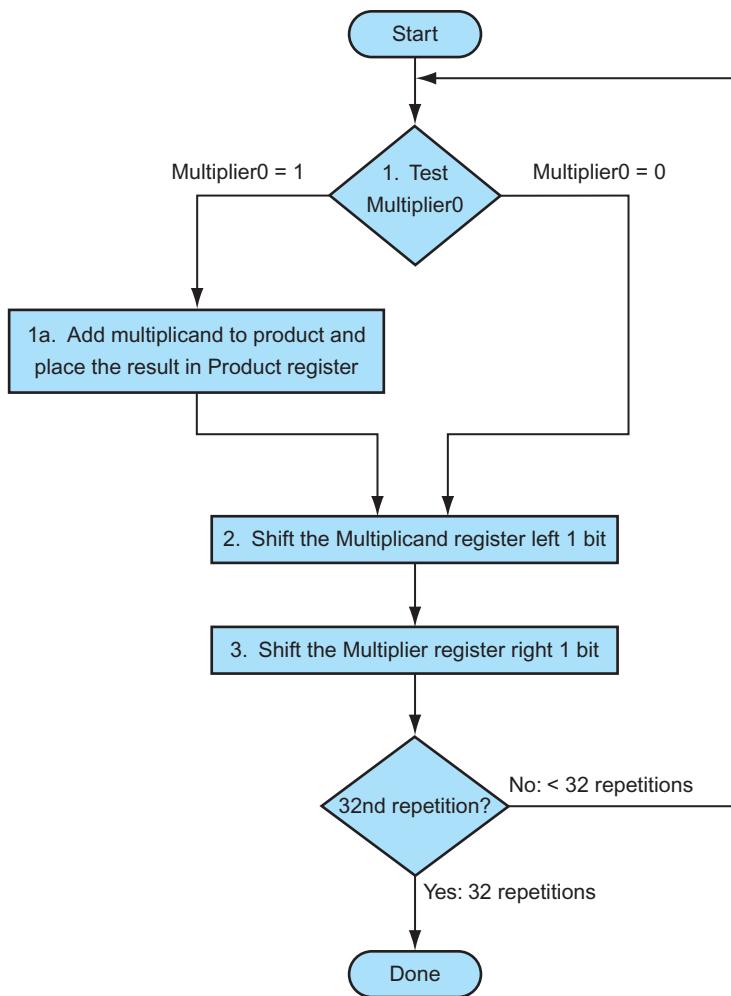


FIGURE 3.4 The first multiplication algorithm, using the hardware shown in Figure 3.3. If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.

This algorithm and hardware are easily refined to take 1 clock cycle per step. The speed-up 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 1. 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 3.5](#) shows the revised hardware.

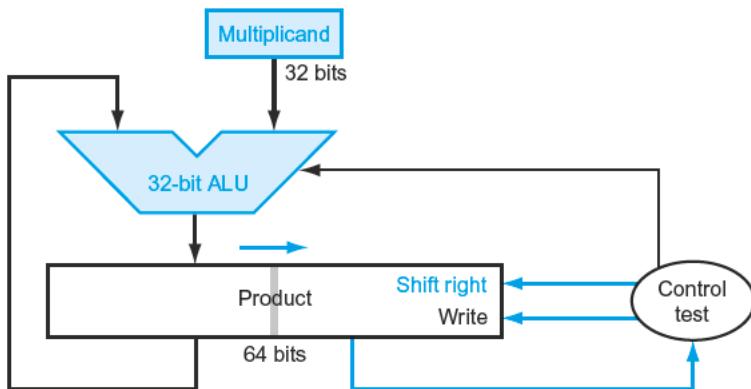


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

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.

EXAMPLE

A Multiply Algorithm

Using 4-bit numbers to save space, multiply $2_{\text{ten}} \times 3_{\text{ten}}$, or $0010_{\text{two}} \times 0011_{\text{two}}$.

ANSWER

Figure 3.6 shows the value of each register for each of the steps labeled according to Figure 3.4, with the final value of $0000\ 0110_{\text{two}}$ 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.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001 $\textcircled{1}$	0000 0010	0000 0000
1	1a: $1 \Rightarrow$ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000 $\textcircled{1}$	0000 0100	0000 0010
2	1a: $1 \Rightarrow$ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000 $\textcircled{0}$	0000 1000	0000 0110
3	1: 0 \Rightarrow No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000 $\textcircled{0}$	0001 0000	0000 0110
4	1: 0 \Rightarrow 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 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Signed Multiplication

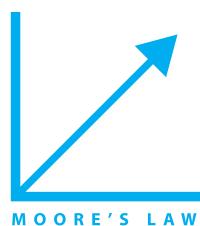
So far, we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. 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 we are dealing with numbers that have infinite digits, and 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.

Faster Multiplication

Moore's Law has provided so much more in resources that hardware designers can now build 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.

A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high. An alternative way to organize these 32 additions is in a parallel tree, as Figure 3.7 shows. Instead of waiting for 32 add times, we wait just the $\log_2(32)$ or five 32-bit add times.



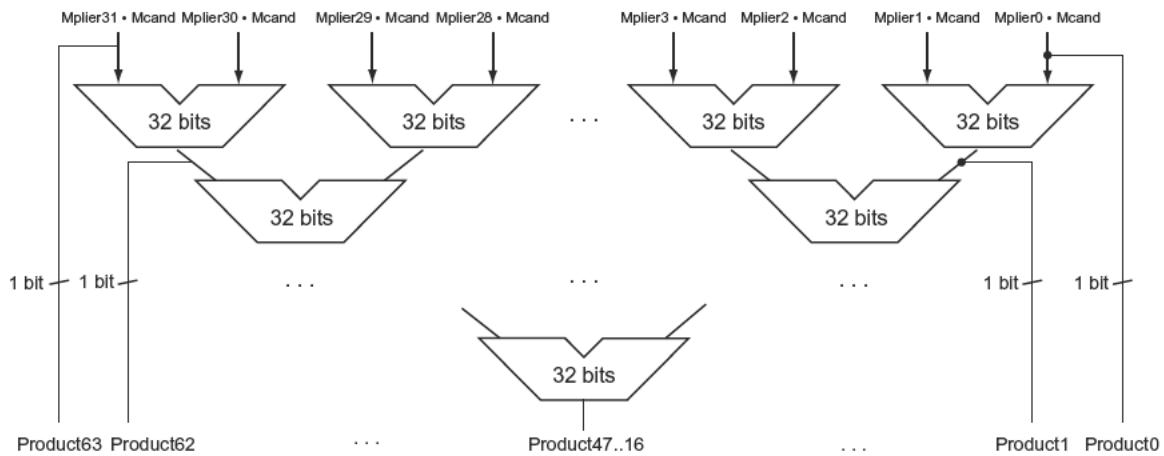


FIGURE 3.7 Fast multiplication hardware. Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31 adders and then organizes them to minimize delay.



In fact, multiply can go even faster than five add times because of the use of *carry save adders* (see Section B.6 in [Appendix B](#)) and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see Chapter 4).

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 hardware simply shifts and adds, as derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of 2. With much more hardware we can do the adds in **parallel**, and do them much faster.

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.

3.4

Division

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 $1,001,010_{\text{ten}}$ by 1000_{ten} :

$$\begin{array}{r}
 & 1001_{\text{ten}} & \text{Quotient} \\
 \text{Divisor } 1000_{\text{ten}} & \overline{)1001010_{\text{ten}}} & \text{Dividend} \\
 -1000 & \hline & \\
 & 10 & \\
 & 101 & \\
 & 1010 & \\
 -1000 & \hline & \\
 & 10_{\text{ten}} & \text{Remainder}
 \end{array}$$

Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, 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 the 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.

A Division Algorithm and Hardware

Figure 3.8 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.

Divide et impera.

Latin for “Divide and rule,” ancient political maxim cited by Machiavelli, 1532

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.

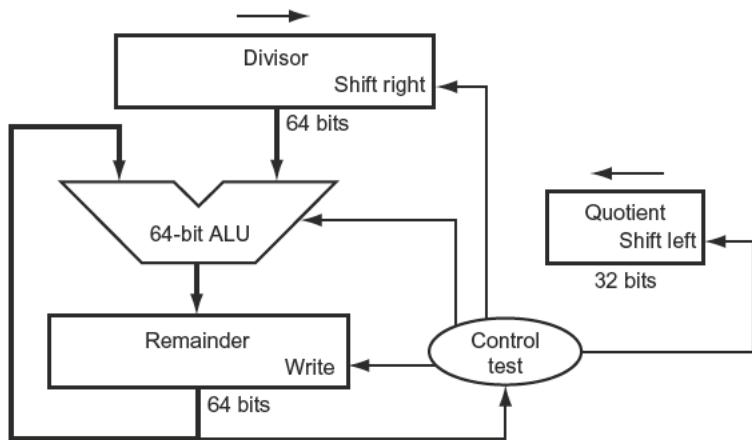


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized 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 3.9 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.

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 $0000\ 0111_{\text{two}}$ by 0010_{two} .

ANSWER

Figure 3.10 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.

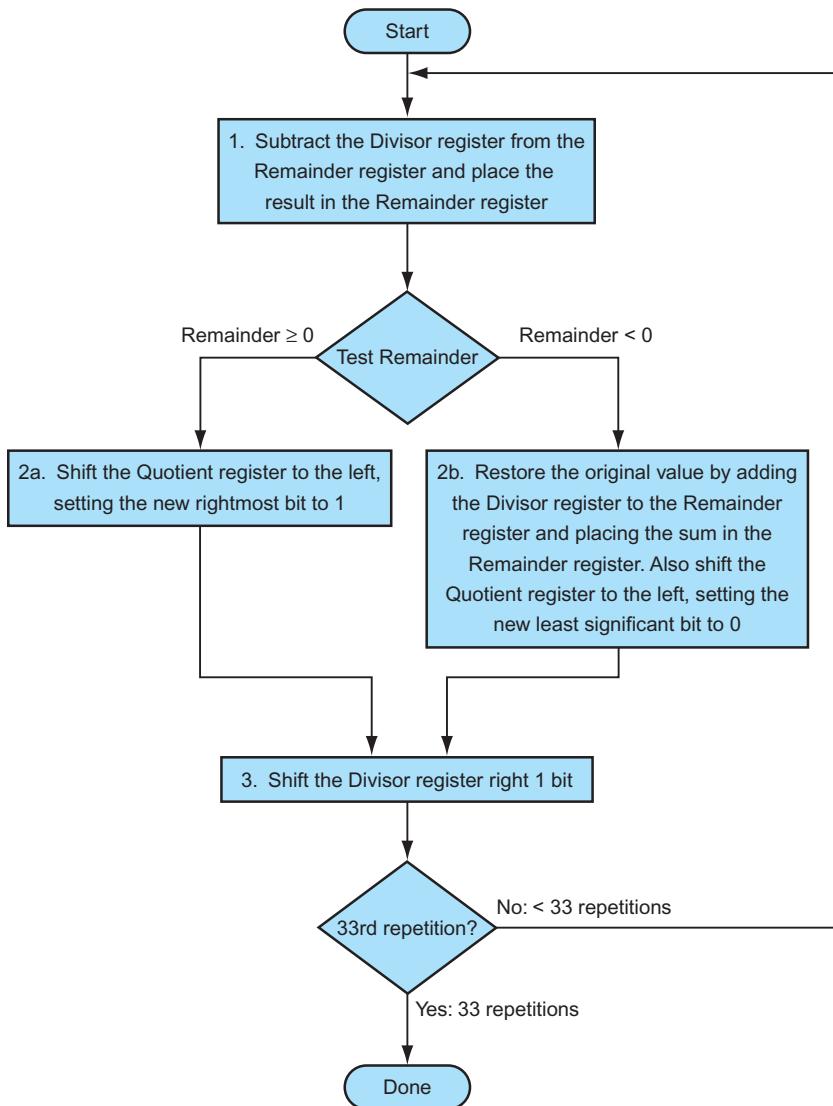


FIGURE 3.9 A division algorithm, using the hardware in Figure 3.8. If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient 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.

This algorithm and hardware can be refined to be faster and cheaper. The speed-up comes from shifting the operands and the quotient simultaneously with the subtraction. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. [Figure 3.11](#) shows the revised hardware.

Iteration	Step	Quotient	Divisor	Reminder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	0110 0111
	2b: Rem < 0 \Rightarrow +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	0111 0111
	2b: Rem < 0 \Rightarrow +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	0111 1111
	2b: Rem < 0 \Rightarrow +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 \geq 0 \Rightarrow 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 \geq 0 \Rightarrow sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

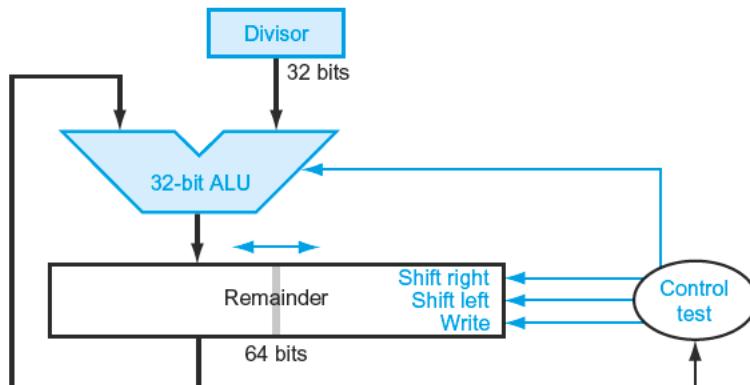


FIGURE 3.11 An Improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.5, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

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 $\pm 7_{\text{ten}}$ by $\pm 2_{\text{ten}}$. The first case is easy:

$$+7 \div +2: \text{Quotient} = +3, \text{ Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2: \text{Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3 \times 2) \\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2: \text{Quotient} = -3, \text{ Remainder} = -1$$

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:

$$+7 \div -2: \text{Quotient} = -3, \text{ Remainder} = +1$$

$$-7 \div -2: \text{Quotient} = +3, \text{ Remainder} = -1$$

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

Moore's Law applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many 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 step. The *SRT division* technique tries to **predict** 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 predictions. 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 the 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 231 in Section 3.9 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 3.5 and 3.11](#). 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. We accelerate division by predicting multiple quotient bits and then correcting mispredictions later. [Figure 3.12](#) summarizes the enhancements to the MIPS architecture for the last two sections.

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 \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	l1 \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store conditional word	sc \$s1,20(\$s2)	Memory[\$s2+20] = \$s1; \$s1 = 0 or 1	Store word as 2nd half atomic swap
Logical	load upper immediate	lui \$s1,100	\$s1 = 100 * 2 ¹⁶	Loads constant in upper 16 bits
	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 3.12 MIPS core architecture. 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. MIPS machine language is listed in the MIPS Reference Data Card at the front of this book.

Hardware/ Software Interface

MIPS divide instructions ignore overflow, so software must determine whether 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 cycle per step, is explored further in the exercises; the algorithm above is called *restoring* division. A third algorithm that doesn't save the result of the subtract if it's negative is called a *nonperforming* division algorithm. It averages one-third fewer arithmetic operations.

3.5

Floating Point

Speed gets you nowhere if you're headed the wrong way.

American proverb

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.

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\dots_{\text{ten}}$ (*pi*)

$2.71828\dots_{\text{ten}}$ (*e*)

0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in a nanosecond)

$3,155,760,000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^9$ (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, $1.0_{\text{ten}} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{\text{ten}} \times 10^{-8}$ and $10.0_{\text{ten}} \times 10^{-10}$ are not.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

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

$$1.\text{xxxxxxxx}_{\text{two}} \times 2^{\text{yyyy}}$$

(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 tradeoff 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.

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. As we recall from Chapter 2, this representation is *sign and magnitude*, since the sign is a separate bit from the rest of the 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
<i>s</i>	exponent																									fraction					
1 bit	8 bits																									23 bits					

In general, floating-point numbers are 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.)

floating point

Computer arithmetic that represents numbers in which the binary point is not fixed.

fraction The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the *mantissa*.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

overflow (floating-point) A situation in which a positive exponent becomes 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.

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.

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{\text{ten}} \times 10^{-38}$ and numbers almost as large as $2.0_{\text{ten}} \times 10^{38}$ 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.

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, we call this event **underflow**. This situation occurs when the negative exponent is 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.

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 field.

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																																
<i>s</i>	exponent										fraction																																																				
1 bit	11 bits											20 bits																																																			
fraction (continued)																																																															
32 bits																																																															

MIPS double precision allows numbers almost as small as $2.0_{\text{ten}} \times 10^{-308}$ and almost as large as $2.0_{\text{ten}} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

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.

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 222. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Thus $00 \dots 00_{\text{two}}$ 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

$$(-1)^s \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \dots) \times 2^E$$

Figure 3.13 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 they are 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.)

Placing the exponent before the significand also simplifies the 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, $1.0_{\text{two}} \times 2^{-1}$ 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	0	0	.	.	.

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{\text{two}} \times 2^{+1}$ 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	0	0	.	.	.

The desirable notation must therefore represent the most negative exponent as $00 \dots 00_{\text{two}}$ and the most positive as $11 \dots 11_{\text{two}}$. 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 an exponent of -1 is represented by the bit pattern of the value $-1 + 127_{\text{ten}}$, or $126_{\text{ten}} = 0111\ 1110_{\text{two}}$, and $+1$ is represented by $1 + 127$, or $128_{\text{ten}} = 1000\ 0000_{\text{two}}$. The exponent bias for double precision is 1023. 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 range of single precision numbers is then from as small as

$$\pm 1.0000000000000000000000000000000_{\text{two}} \times 2^{-126}$$

to as large as

$$\pm 1.1111111111111111111111111111111_{\text{two}} \times 2^{+127}.$$

Let's demonstrate.

Floating-Point Representation

EXAMPLE

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

The number -0.75_{ten} is also

$$-3/4_{\text{ten}} \text{ or } -3/2^2_{\text{ten}}$$

It is also represented by the binary fraction

$$-11_{\text{two}} / 2^2_{\text{ten}} \text{ or } -0.11_{\text{two}}$$

In scientific notation, the value is

$$-0.11_{\text{two}} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{\text{two}} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^s \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{\text{two}} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 000_{\text{two}}) \times 2^{(126 - 127)}$$

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	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit

8 bits

23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{two}}) \times 2^{(1022 - 1023)}$$

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	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

1 bit

11 bits

20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

32 bits

Now let's try going the other direction.

EXAMPLE

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

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	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

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 few subsections, 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: Following IEEE guidelines, the IEEE 754 committee was reformed 20 years after the standard to see what changes, if any, should be made. The revised standard IEEE 754-2008 includes nearly all the IEEE 754-1985 and adds a 16-bit format (“half precision”) and a 128-bit format (“quadruple precision”). No hardware has yet been built that supports quadruple precision, but it will surely come. The revised standard also add decimal floating point arithmetic, which IBM mainframes have implemented.

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. IBM mainframes now support IEEE 754 as well as the hex format.

Floating-Point Addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{\text{ten}} \times 10^1 + 1.610_{\text{ten}} \times 10^{-1}$. 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, $1.610_{\text{ten}} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$1.610_{\text{ten}} \times 10^{-1} = 0.1610_{\text{ten}} \times 10^0 = 0.01610_{\text{ten}} \times 10^1$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{\text{ten}} \times 10^1$. 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

$$0.016 \times 10^1$$

- Step 2. Next comes the addition of the significands:

$$\begin{array}{r} 9.999_{\text{ten}} \\ + 0.016_{\text{ten}} \\ \hline 10.015_{\text{ten}} \end{array}$$

The sum is $10.015_{\text{ten}} \times 10^1$.

- Step 3. This sum is not in normalized scientific notation, so we need to adjust it:

$$10.015_{\text{ten}} \times 10^1 = 1.0015_{\text{ten}} \times 10^2$$

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

$$1.0015_{\text{ten}} \times 10^2$$

is rounded to four digits in the significand to

$$1.002_{\text{ten}} \times 10^2$$

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 3.14](#) 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 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers (see the *Elaboration* on page 222). For the example below, remember that for single precision, the maximum exponent is 127, and the minimum exponent is -126.

EXAMPLE

Binary Floating-Point Addition

Try adding the numbers 0.5_{ten} and -0.4375_{ten} in binary using the algorithm in [Figure 3.14](#).

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:

$$\begin{aligned} 0.5_{\text{ten}} &= 1/2_{\text{ten}} &= 1/2^1_{\text{ten}} \\ &= 0.1_{\text{two}} &= 0.1_{\text{two}} \times 2^0 &= 1.000_{\text{two}} \times 2^{-1} \\ -0.4375_{\text{ten}} &= -7/16_{\text{ten}} &= -7/2^4_{\text{ten}} \\ &= -0.0111_{\text{two}} &= -0.0111_{\text{two}} \times 2^0 &= -1.110_{\text{two}} \times 2^{-2} \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{\text{two}} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{\text{two}} \times 2^{-2} = -0.111_{\text{two}} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{\text{two}} \times 2^{-1} + (-0.111_{\text{two}} \times 2^{-1}) = 0.001_{\text{two}} \times 2^{-1}$$

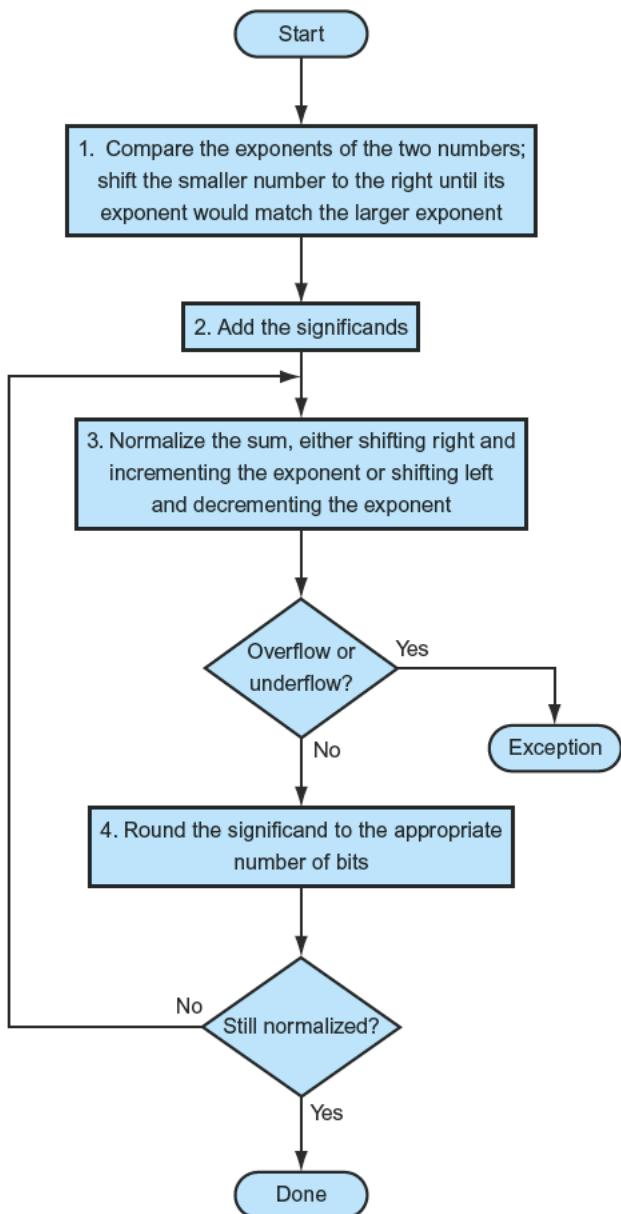


FIGURE 3.14 Floating-point addition. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{\text{two}} \times 2^{-1} &= 0.010_{\text{two}} \times 2^{-2} = 0.100_{\text{two}} \times 2^{-3} \\ &= 1.000_{\text{two}} \times 2^{-4} \end{aligned}$$

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:

$$1.000_{\text{two}} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{\text{two}} \times 2^{-4} &= 0.0001000_{\text{two}} = 0.0001_{\text{two}} \\ &= 1/2^4_{\text{ten}} = 1/16_{\text{ten}} = 0.0625_{\text{ten}} \end{aligned}$$

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 3.15](#) sketches the basic organization of hardware for floating-point addition.

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: $1.110_{\text{ten}} \times 10^{10} \times 9.200_{\text{ten}} \times 10^{-5}$. 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:

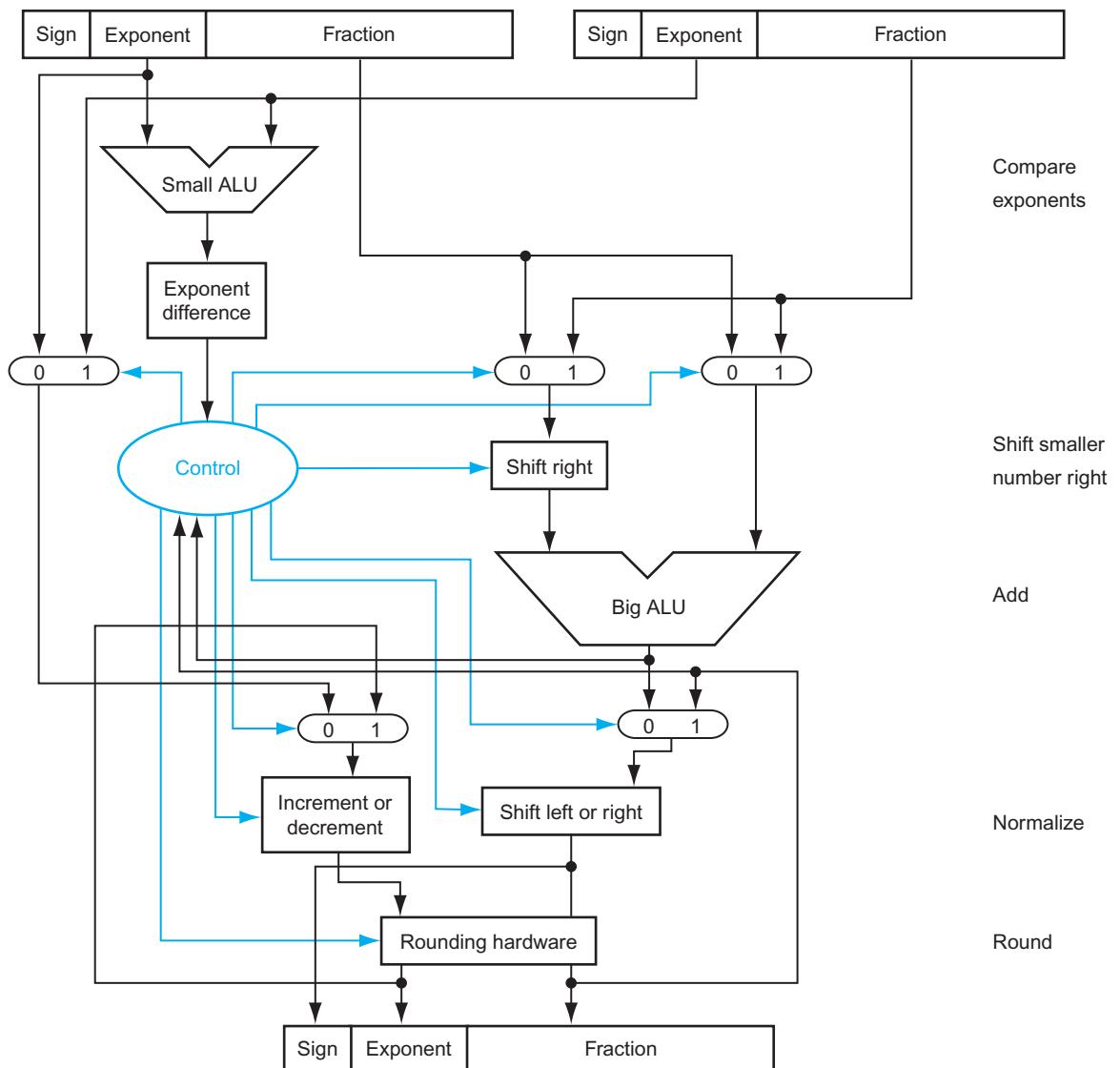


FIGURE 3.15 Block diagram of an arithmetic unit dedicated to floating-point addition. The steps of Figure 3.14 correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference 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 actual final result.

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:

$$\begin{array}{r} 1.110_{\text{ten}} \\ \times 9.200_{\text{ten}} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{\text{ten}} \end{array}$$

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{\text{ten}}$$

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:

$$10.212_{\text{ten}} \times 10^5 = 1.0212_{\text{ten}} \times 10^6$$

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

$$1.0212_{\text{ten}} \times 10^6$$

is rounded to four digits in the significand to

$$1.021_{\text{ten}} \times 10^6$$

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

$$+1.021_{\text{ten}} \times 10^6$$

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.

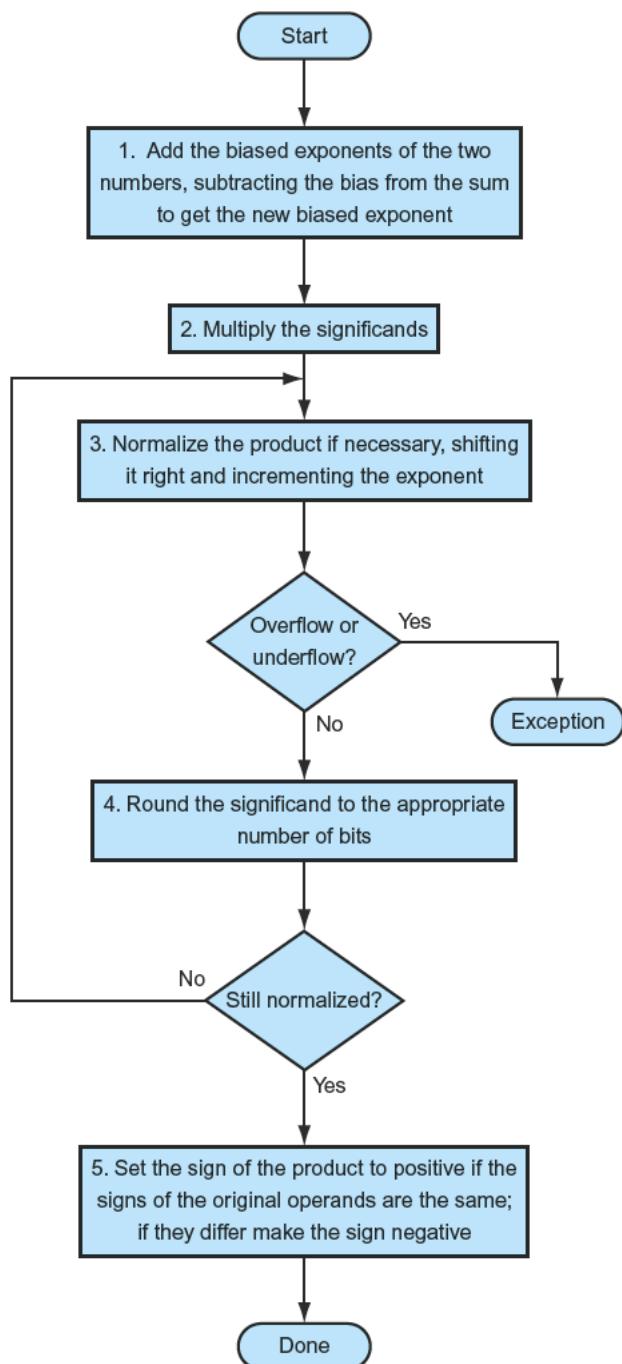


FIGURE 3.16 Floating-point multiplication. The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.

Once again, as [Figure 3.16](#) 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).

EXAMPLE

ANSWER

Binary Floating-Point Multiplication

Let's try multiplying the numbers 0.5_{ten} and -0.4375_{ten} , using the steps in [Figure 3.16](#).

In binary, the task is multiplying $1.000_{\text{two}} \times 2^{-1}$ by $-1.110_{\text{two}} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$\begin{aligned} (-1 + 127) + (-2 + 127) - 127 &= (-1 - 2) + (127 + 127 - 127) \\ &= -3 + 127 = 124 \end{aligned}$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{\text{two}} \\ \times 1.110_{\text{two}} \\ \hline 0000 \\ 1000 \\ 1000 \\ \hline 1110000_{\text{two}} \end{array}$$

The product is $1.110000_{\text{two}} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{\text{two}} \times 2^{-3}$.

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:

$$1.110_{\text{two}} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{\text{two}} \times 2^{-3} &= -0.001110_{\text{two}} = -0.00111_{\text{two}} \\ &= -7/2^5_{\text{ten}} = -7/32_{\text{ten}} = -0.21875_{\text{ten}} \end{aligned}$$

The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten} .

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* (bclt) 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: lwcl and swcl. The base registers for floating-point data transfers which are used for addresses 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:

```
lwcl      $f4,c($sp)  # Load 32-bit F.P. number into F4
lwcl      $f6,a($sp)  # Load 32-bit F.P. number into F6
add.s    $f2,$f4,$f6  # F2 = F4 + F6 single precision
swcl      $f2,b($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. Thus, the pair of single precision registers \$f2 and \$f3 also form the double precision register named \$f2.

[Figure 3.17](#) 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 2.19 in Chapter 2, [Figure 3.18](#) shows the encoding of these instructions.

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	$\$f2 = \$f4 \times \$f6$	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	$\$f2 = \$f4 \times \$f6$	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	lwcl \$f1,100(\$s2)	$\$f1 = \text{Memory}[\$s2 + 100]$	32-bit data to FP register
	store word copr. 1	swcl \$f1,100(\$s2)	$\text{Memory}[\$s2 + 100] = \$f1$	32-bit data to memory
Conditional branch	branch on FP true	bc1t 25	if (cond == 1) go to PC + 4	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	
lwcl	I	49	20	2	100				lwcl \$f2,100(\$s4)
swcl	I	57	20	2	100				swcl \$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 3.17 MIPS floating-point architecture revealed thus far. See Appendix A, Section A.10, for more detail. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

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	FIPt						
3(011)								
4(100)	lb	lh	lw	lw	lbu	lhu	lw	
5(101)	sb	sh	sw	sw			sw	
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 3.18 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, `FIPt` 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 Chapter 2 or this chapter, with Appendix A covering all instructions. This information is also found in column 2 of the MIPS Reference Data Card at the front of this book.

Hardware/ Software Interface

One issue that architects 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):

```
lwcl $f18, const32($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.

Compiling Floating-Point C Procedure with Two-Dimensional Matrices into MIPS

EXAMPLE

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of $C = C + A * B$. It is commonly called DGEMM, for Double precision, General Matrix Multiply. We'll see versions of DGEMM again in Section 3.8 and subsequently in Chapters 4, 5, and 6. Let's assume C , A , and B are all square matrices with 32 elements in each dimension.

```
void mm (double c[][], double a[][], double b[][])
{
    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)
                c[i][j] = c[i][j] + a[i][k] * b[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?

Note that `c[i][j]` is used in the innermost loop above. Since the loop index is `k`, the index does not affect `c[i][j]`, so we can avoid loading and storing `c[i][j]` each iteration. Instead, the compiler loads `c[i][j]` into a register outside the loop, accumulates the sum of the products of `a[i][k]` and

ANSWER

$b[k][j]$ in that same register, and then stores the sum into $c[i][j]$ upon termination of the innermost loop.

We keep the code simpler by using the assembly language pseudoinstructions `l i` (which loads a constant into a register), and `l.d` and `s.d` (which the assembler turns into a pair of data transfer instructions, `lwcl` or `swcl`, 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 $c[i][j]$, 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 * 25 (size of row of c)
```

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 c , giving the address of $c[i][j]$, and then load the double precision number $c[i][j]$ into $\$f4$:

```
addu $t2, $a0, $t2  # $t2 = byte address of c[i][j]
l.d  $f4, 0($t2)    # $f4 = 8 bytes of c[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number $b[k][j]$.

```
L3: sll $t0, $s2, 5      # $t0 = k * 25 (size of row of b)
    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 b[k][j]
    l.d  $f16, 0($t0)   # $f16 = 8 bytes of b[k][j]
```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number $a[i][k]$.

```

sll      $t0, $s0, 5      # $t0 = i * 25 (size of row of a)
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 a[i][k]
l.d     $f18, 0($t0)    # $f18 = 8 bytes of a[i][k]

```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of a and b located in registers $\$f18$ and $\$f16$, and then accumulate the sum in $\$f4$.

```

mul.d  $f16, $f18, $f16  # $f16 = a[i][k] * b[k][j]
add.d  $f4, $f4, $f16    # f4 = c[i][j] + a[i][k] * b[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 $c[i][j]$.

```

addiu  $s2, $s2, 1      # $k = k + 1
bne    $s2, $t1, L3      # if (k != 32) go to L3
s.d    $f4, 0($t2)      # c[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
...

```

[Figure 3.22](#) below shows the x86 assembly language code for a slightly different version of DGEMM in [Figure 3.21](#).

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.

Elaboration: Only 16 of the 32 MIPS floating-point registers could originally be used for double precision operations: $\$f0, \$f2, \$f4, \dots, \$f30$. Double precision is computed using pairs of these single precision 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 (see Section 3.6). For example, `add.ps $f0, $f2, $f4` is equivalent to `add.s $f0, $f2, $f4` followed by `add.s $f1, $f3, $f5`.

Elaboration: 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, was 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 5.) 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: As mentioned in Section 3.4, accelerating division is more challenging than multiplication. In addition to SRT, another 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 $1/c$, which is then multiplied by the other operand. Iteration techniques cannot be rounded properly without calculating many extra bits. A TI chip solved 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 above 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 access. It would also need to check that the object reference is not null.

Accurate Arithmetic

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.

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 getting 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 two extra bits on the right during intermediate additions, called **guard** and **round**, respectively. Let's do a decimal example to illustrate their value.

Rounding with Guard Digits

Add $2.56_{\text{ten}} \times 10^0$ to $2.34_{\text{ten}} \times 10^2$, 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.

EXAMPLE

First we must shift the smaller number to the right to align the exponents, so $2.56_{\text{ten}} \times 10^0$ becomes $0.0256_{\text{ten}} \times 10^2$. 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

$$\begin{array}{r} 2.3400_{\text{ten}} \\ + 0.0256_{\text{ten}} \\ \hline 2.3656_{\text{ten}} \end{array}$$

Thus the sum is $2.3656_{\text{ten}} \times 10^2$. 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 tiebreaker. Rounding the sum up with three significant digits yields $2.37_{\text{ten}} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} 2.34_{\text{ten}} \\ + 0.02_{\text{ten}} \\ \hline 2.36_{\text{ten}} \end{array}$$

The answer is $2.36_{\text{ten}} \times 10^2$, 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 were 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 represented.

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 one 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.

IEEE 754 has 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 U.S. Internal Revenue Service (IRS) 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 round 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 $0.50 \dots 00_{\text{ten}}$ and $0.50 \dots 01_{\text{ten}}$ when rounding.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{\text{ten}} \times 10^{-1}$ to $2.34_{\text{ten}} \times 10^2$ 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.

Elaboration: PowerPC, SPARC64, AMD SSE5, and Intel AVX architectures provide a single instruction that does a multiply and add on three registers: $a = a + (b \times c)$. Obviously, this instruction allows potentially higher floating-point performance for this common operation. Equally important is that instead of performing two roundings—after the multiply and then after the add—which would happen with separate instructions, the multiply add instruction can perform a single rounding after the add. A single rounding step increases the precision of multiply add. Such operations with a single rounding are called **fused multiply add**. It was added to the IEEE 754-2008 standard (see  [Section 3.11](#)).

fused multiply add

A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

Summary

The *Big Picture* that follows 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

$$(-1)^{\text{S}} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

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.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size and 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.

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	add, addi, sub, mult, div AND, ANDi, OR, ORi, NOR,slt, slti
-	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwcl, swcl	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

In the last chapter, we presented the storage classes of the programming language C (see the *Hardware/Software Interface* section in Section 2.7). The table above shows some of the C and Java data types, the MIPS data transfer instructions, and instructions that operate on those types that appear in Chapter 2 and this chapter. Note that Java omits unsigned integers.

Hardware/ Software Interface

The revised IEEE 754-2008 standard added a 16-bit floating-point format with five exponent bits. What do you think is the likely range of numbers it could represent?

Check Yourself

1. $1.0000\ 00 \times 2^0$ to $1.1111\ 1111\ 11 \times 2^{31}, 0$
2. $\pm 1.0000\ 0000\ 0 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 1 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
3. $\pm 1.0000\ 0000\ 00 \times 2^{-14}$ to $\pm 1.1111\ 1111\ 11 \times 2^{15}, \pm 0, \pm \infty, \text{NaN}$
4. $\pm 1.0000\ 0000\ 00 \times 2^{-15}$ to $\pm 1.1111\ 1111\ 11 \times 2^{14}, \pm 0, \pm \infty, \text{NaN}$

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 fraction. 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

$$1.0000\ 0000\ 0000\ 0000\ 000_{\text{two}} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 001_{\text{two}} \times 2^{-126}, \text{ or } 1.0_{\text{two}} \times 2^{-149}$$

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. Moreover, if programmers do not expect denorms, their programs may surprise them.

3.6

Parallelism and Computer Arithmetic: Subword Parallelism

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.

Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see Section 2.9), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data. By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands. The cost of such partitioned adders was small.

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They have been also called vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia



PARALLELISM

applications led to arithmetic instructions that support narrower operations that can easily operate in parallel.

For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support subword parallelism, which can be used either with ARMv7 or ARMv8. It added 256 bytes of new registers for NEON that can be viewed as 32 registers 8 bytes wide or 16 registers 16 bytes wide. NEON supports all the subword data types you can imagine *except* 64-bit floating point numbers:

- 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers
- 32-bit floating point numbers

Figure 3.19 gives a summary of the basic NEON instructions.

Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3,4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3,4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

FIGURE 3.19 Summary of ARM NEON Instructions for subword parallelism. We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

Elaboration: In addition to signed and unsigned integers, ARM includes “fixed-point” format of four sizes called I8, I16, I32, and I64, of which 16, 8, 4, and 2 fit in a 128-bit register, respectively. A portion of the fixed point is for the fraction (to the right of the binary point) and the rest of the data is the integer portion (to the left of the binary point). The location of the binary point is up to the software. Many ARM processors do not have floating point hardware and thus floating point operations must be performed by library routines. Fixed point arithmetic can be significantly faster than software floating point routines, but more work for the programmer.

3.7

Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

The original MMX (*MultiMedia eXtension*) and SSE (*Streaming SIMD Extension*) instructions for the x86 included similar operations to those found in ARM NEON. Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture as part of SSE2, including double precision floating-point registers and operations. It includes eight 64-bit registers that can be used for floating-point operands. AMD expanded the number to 16 registers, called XMM, as part of AMD64, which Intel relabeled EM64T for its use. Figure 3.20 summarizes the SSE and SSE2 instructions.

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. Thus, the 16 floating-point registers for SSE2 are actually 128 bits wide. 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 (PS) or two doubles (PD).

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV{H/L}{PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX{SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

FIGURE 3.20 The SSE/SSE2 floating-point Instructions of the x86. xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for *Scalar Single* precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for *Packed Single* precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for *Scalar Double* precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for *Packed Double* precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

In 2011 Intel doubled the width of the registers again, now called YMM, with *Advanced Vector Extensions* (AVX). Thus, a single operation can now specify eight 32-bit floating-point operations or four 64-bit floating-point operations. The legacy SSE and SSE2 instructions now operate on the lower 128 bits of the YMM registers. Thus, to go from 128-bit and 256-bit operations, you prepend the letter “v” (for vector) in front of the SSE2 assembly language operations and then use the YMM register names instead of the XMM register name. For example, the SSE2 instruction to perform two 64-bit floating-point multiplies

```
addpd %xmm0, %xmm4
```

It becomes

```
vaddpd %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies.

Elaboration: AVX also added three address instructions to x86. For example, vaddpd can now specify

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

instead of the standard two address version

```
addpd %xmm0, %xmm4 # %xmm4 = %xmm4 + %xmm0
```

(Unlike MIPS, the destination is on the right in x86.) Three addresses can reduce the number of registers and instructions needed for a computation.

3.8

Going Faster: Subword Parallelism and Matrix Multiply

To demonstrate the performance impact of subword parallelism, we'll run the same code on the Intel Core i7 first without AVX and then with it. [Figure 3.21](#) shows an unoptimized version of a matrix-matrix multiply written in C. As we saw in Section 3.5, this program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Starting with this edition, we have added a new section entitled “Going Faster” to demonstrate the performance benefit of adapting software to the underlying hardware, in this case the Sandy Bridge version of the Intel Core i7 microprocessor. This new section in Chapters 3, 4, 5, and 6 will incrementally improve DGEMM performance using the ideas that each chapter introduces.

[Figure 3.22](#) shows the x86 assembly language output for the inner loop of [Figure 3.21](#). The five floating point-instructions start with a v like the AVX instructions, but note that they use the XMM registers instead of YMM, and they include sd in the name, which stands for scalar double precision. We'll define the subword parallel instructions shortly.

```

1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.     {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for( int k = 0; k < n; k++ )
8.             cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.    }
11. }
```

FIGURE 3.21 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision General Matrix Multiply (GEMM). Because we are passing the matrix dimension as the parameter n , this version of DGEMM uses single dimensional versions of matrices C , A , and B and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Section 3.5. The comments remind us of this more intuitive notation.

```

1. vmovsd (%r10),%xmm0          # Load 1 element of C into %xmm0
2. mov    %rsi,%rcx              # register %rcx = %rsi
3. xor    %eax,%eax              # register %eax = 0
4. vmovsd (%rcx),%xmm1          # Load 1 element of B into %xmm1
5. add    %r9,%rcx              # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
7. add    $0x1,%rax              # register %rax = %rax + 1
8. cmp    %eax,%edi              # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0      # Add %xmm1, %xmm0
10. jg     30 <dgemm+0x30>        # jump if %eax > %edi
11. add    $0x1,%r11d              # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)          # Store %xmm0 into C element
```

FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21. Although it is dealing with just 64-bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the Elaboration in Section 3.7).

```

1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                               _mm256_broadcast_sd(B+k+j*n)));
11.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }

```

FIGURE 3.23 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86. Figure 3.24 shows the assembly language produced by the compiler for the inner loop.

While compiler writers may eventually be able to routinely produce high-quality code that uses the AVX instructions of the x86, for now we must “cheat” by using C intrinsics that more or less tell the compiler exactly how to produce good code. Figure 3.23 shows the enhanced version of Figure 3.21 for which the Gnu C compiler produces AVX code. Figure 3.24 shows annotated x86 code that is the output of compiling using gcc with the -O3 level of optimization.

The declaration on line 6 of Figure 3.23 uses the `__m256d` data type, which tells the compiler the variable will hold 4 double-precision floating-point values. The intrinsic `_mm256_load_pd()` also on line 6 uses AVX instructions to load 4 double-precision floating-point numbers in parallel (`_pd`) from the matrix `C` into `c0`. The address calculation `C+i+j*n` on line 6 represents element `C[i+j*n]`. Symmetrically, the final step on line 11 uses the intrinsic `_mm256_store_pd()` to store 4 double-precision floating-point numbers from `c0` into the matrix `C`. As we’re going through 4 elements each iteration, the outer `for` loop on line 4 increments `i` by 4 instead of by 1 as on line 3 of Figure 3.21.

Inside the loops, on line 9 we first load 4 elements of `A` again using `_mm256_load_pd()`. To multiply these elements by one element of `B`, on line 10 we first use the intrinsic `_mm256_broadcast_sd()`, which makes 4 identical copies of the scalar double precision number—in this case an element of `B`—in one of the YMM registers. We then use `_mm256_mul_pd()` on line 9 to multiply the four double-precision results in parallel. Finally, `_mm256_add_pd()` on line 8 adds the 4 products to the 4 sums in `c0`.

Figure 3.24 shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the five AVX instructions—they all start with `v` and

```

1. vmovapd (%r11),%ymm0          # Load 4 elements of C into %ymm0
2. mov    %rbx,%rcx             # register %rcx = %rbx
3. xor    %eax,%eax            # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymml # Make 4 copies of B element
5. add    $0x8,%rax            # register %rax = %rax + 8
6. vmulpd (%rcx),%ymml,%ymml   # Parallel mul %ymml,4 A elements
7. add    %r9,%rcx             # register %rcx = %rcx + %r9
8. cmp    %r10,%rax            # compare %r10 to %rax
9. vaddpd %ymml,%ymm0,%ymm0    # Parallel add %ymml, %ymm0
10. jne   50 <dgemm+0x50>      # jump if not %r10 != %rax
11. add    $0x1,%esi            # register %esi = %esi + 1
12. vmovapd %ymm0,(%r11)        # Store %ymm0 into 4 C elements

```

FIGURE 3.24 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.23. Note the similarities to Figure 3.22, with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision.

four of the five use pd for parallel double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in Figure 3.22 above: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from *scalar double* (sd) using XMM registers to *parallel double* (pd) with YMM registers. The one exception is line 4 of Figure 3.24. Every element of A must be multiplied by one element of B. One solution is to place four identical copies of the 64-bit B element side-by-side into the 256-bit YMM register, which is just what the instruction *vbroadcastsd* does.

For matrices of dimensions of 32 by 32, the unoptimized DGEMM in Figure 3.21 runs at 1.7 GigaFLOPS (Floating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge). The optimized code in Figure 3.23 performs at 6.4 GigaFLOPS. The AVX version is 3.85 times as fast, which is very close to the factor of 4.0 increase that you might hope for from performing 4 times as many operations at a time by using **subword parallelism**.



Elaboration: As mentioned in the Elaboration in Section 1.6, Intel offers Turbo mode that temporarily runs at a higher clock rate until the chip gets too hot. This Intel Core i7 (Sandy Bridge) can increase from 2.6 GHz to 3.3 GHz in Turbo mode. The results above are with Turbo mode turned off. If we turn it on, we improve all the results by the increase in the clock rate of $3.3/2.6 = 1.27$ to 2.1 GFLOPS for unoptimized DGEMM and 8.1 GFLOPS with AVX. Turbo mode works particularly well when using only a single core of an eight-core chip, as in this case, as it lets that single core use much more than its fair share of power since the other cores are idle.

3.9

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: 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 c , where x_i means the i th bit, represents the number

$$\dots + (x^3 \times 2^3) + (x^2 \times 2^2) 1 (x_1 \times 2^1) + (x_0 \times 2^0)$$

Shifting the bits of c 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

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{\text{two}}$$

According to this fallacy, shifting right by two should divide by $4_{\text{ten}} (2^2)$:

$$0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $1,073,741,822_{\text{ten}}$ instead of -1_{ten} .

A solution would be to have an arithmetic right shift that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}}$$

The result is -2_{ten} instead of -1_{ten} ; close, but no cigar.

Pitfall: Floating-point addition is not associative.

Associativity holds for a sequence of two's complement integer additions, even if the computation overflows. Alas, because floating-point numbers are approximations of real numbers and because computer arithmetic has limited precision, it does not hold for floating-point numbers. 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, let's see if $c + (a + b) = (c + a) + b$. Assume $c = -1.5_{\text{ten}} \times 10^{38}$, $a = 1.5_{\text{ten}} \times 10^{38}$, and $b = 1.0$, and that these are all single precision numbers.

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

$$\begin{aligned}
 c + (a + b) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\
 &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\
 &= 0.0 \\
 c + (a + b) &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\
 &= (0.0_{\text{ten}}) + 1.0 \\
 &= 1.0
 \end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{\text{ten}} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{\text{ten}} \times 10^{38} + 1.0$ is still $1.5_{\text{ten}} \times 10^{38}$. That is why the sum of c , a , and b is 0.0 or 1.0, depending on the order of the floating-point additions, so $c + (a + b) \neq (c + a) + b$. Therefore, floating-point addition is *not* associative.

Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.

Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, “Do the two versions get the same answer?” If the answer is no, you presume there is a bug in the parallel version that you need to track down.

This approach assumes that computer arithmetic does not affect the results when going from sequential to parallel. That is, if you were to add a million numbers together, you would get the same results whether you used 1 processor or 1000 processors. This assumption holds for two’s complement integers, since integer addition is associative. Alas, since floating-point addition is not associative, the assumption does not hold.

A more vexing version of this fallacy occurs on a parallel computer where the operating system scheduler may use a different number of processors depending on what other programs are running on a parallel computer. As the varying number of processors from each run would cause the floating-point sums to be calculated in different orders, getting slightly different answers each time despite running identical code with identical input may flummox unaware parallel programmers.

Given this quandary, programmers who write parallel code with floating-point numbers need to verify whether the results are credible even if they don’t give the same exact answer as the sequential code. The field that deals with such issues is called numerical analysis, which is the subject of textbooks in its own right. Such concerns are one reason for the popularity of numerical libraries such as LAPACK and SCALAPAK, which have been validated in both their sequential and parallel forms.

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 3.25). The following is the inside story behind the headlines.

The Pentium used 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 subtracted from the remainder to generate a new remainder. Like nonrestoring division, 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 engineers thought could never be accessed, and they optimized the logic to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11



FIGURE 3.25 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.

bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

A math professor at Lynchburg College in Virginia, Thomas Nicely, discovered the bug in September 1994. After calling Intel technical support and getting no official reaction, he posted his discovery on the Internet. This post led to a story in a trade magazine, which in turn caused Intel to issue a press release. It called the bug a glitch that would affect only theoretical mathematicians, with the average spreadsheet user seeing an error every 27,000 years. IBM Research soon counterclaimed that the average spreadsheet user would see an error every 24 days. Intel soon threw in the towel by making the following announcement on December 21:

"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 engineers 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?

3.10

Concluding Remarks

Over the decades, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic is found in every computer sold today, and if it includes floating point support, it offers the IEEE 754 binary floating-point arithmetic.

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. Chapters 4 and 5 discuss 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 of floating point are part of the inspiration for the field of numerical analysis. The recent switch to **parallelism** shines the searchlight on numerical analysis again, as solutions that were long considered safe on sequential computers must be reconsidered when trying to find the fastest algorithm for parallel computers that still achieves a correct result.

Data-level parallelism, specifically subword parallelism, offers a simple path to higher performance for programs that are intensive in arithmetic operations for either integer or floating-point data. We showed that we could speed up matrix multiply nearly fourfold by using instructions that could execute four floating-point operations at a time.

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. Two figures try to make this clear.

[Figure 3.26](#) 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 3.27](#) are the instructions the MIPS processor executes that are not found in [Figure 3.26](#). We call the full set of hardware instructions *MIPS-32*. On the right of [Figure 3.27](#) are the instructions accepted by the assembler that are not part of MIPS-32. We call this set of instructions *Pseudo MIPS*.

[Figure 3.28](#) gives the popularity of the MIPS instructions for SPEC CPU2006 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 0.2% of the instructions executed.

Note that although programmers and compiler writers may use MIPS-32 to have a richer menu of options, MIPS core instructions dominate integer SPEC CPU2006 execution, and the integer core plus arithmetic core dominate SPEC CPU2006 floating point, as the table below shows.

Instruction subset	Integer	Fl. pt.
MIPS core	98%	31%
MIPS arithmetic core	2%	66%
Remaining MIPS-32	0%	3%

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 you can see, the MIPS core includes the most popular MIPS instructions; be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious computers. No matter what the instruction set or its size—MIPS, ARM, x86—never forget that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, string, instruction, and so on. In stored program computers, it is the operation on the bit pattern that determines its meaning.



PARALLELISM

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
load linked (atomic update)	ll	I	branch on floating-point false	bc1f	I
store cond. (atomic update)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
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 3.26 The MIPS Instruction set. This book concentrates on the instructions in the left column. This information is also found in columns 1 and 2 of the MIPS Reference Data Card at the front of this book.

Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or ($rs \oplus rt$)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate (signed or unsigned)	neg <u>s</u>	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw (signed or uns.)	mul <u>s</u>	rd,rs,rt
shift right arithmetic variable	srav	R	multiply and check oflw (signed or uns.)	mul <u>os</u>	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder (signed or unsigned)	rem <u>s</u>	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left (unaligned)	lw1	I	load address	la	rd,addr
load word right (unaligned)	lwr	I	load double	ld	rd,addr
store word left (unaligned)	sw1	I	store double	sd	rd,addr
store word right (unaligned)	swr	I	unaligned load word	ulw	rd,addr
load linked (atomic update)	ll	I	unaligned store word	usw	rd,addr
store cond. (atomic update)	sc	I	unaligned load halfword (signed or uns.)	ulhs	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (S or uns.)	madd <u>s</u>	R	branch on equal zero	beqz	rs,L
multiply and subtract (S or uns.)	msub <u>s</u>	I	branch on compare (signed or unsigned)	bxs	rs,rt,L
branch on \geq zero and link	bgezal	I	($x = lt, le, gt, ge$)		
branch on $<$ zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare (signed or unsigned)	sxs	rd,rs,rt
branch compare to zero likely	bxzl	I	($x = lt, le, gt, ge$)		
($x = lt, le, gt, ge$)			load to floating point (s or d)	l.f	rd,addr
branch compare reg likely	bxl	I	store from floating point (s or d)	s.f	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
($x = eq, neq, lt, le, gt, ge$)					
return from exception	rfe	R			
system call	syscall	I			
break (cause exception)	break	I			
move from FP to integer	mfcl	R			
move to FP from integer	mtcl	R			
FP move (s or d)	mov. <u>f</u>	R			
FP move if zero (s or d)	movz. <u>f</u>	R			
FP move if not zero (s or d)	movn. <u>f</u>	R			
FP square root (s or d)	sqrt. <u>f</u>	R			
FP absolute value (s or d)	abs. <u>f</u>	R			
FP negate (s or d)	neg. <u>f</u>	R			
FP convert (w, s, or d)	cvt. <u>ff</u>	R			
FP compare un (s or d)	c.xn. <u>f</u>	R			

FIGURE 3.27 Remaining MIPS-32 and Pseudo MIPS instruction sets. f means single (s) or double (d) precision 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). The underscore represents the letter to include to represent that datatype.

Core MIPS	Name	Integer	Floating point	Arithmetic core + MIPS-32	Name	Integer	Floating point
add	add	0.0%	0.0%	FP add double	add.d	0.0%	10.6%
add immediate	addi	0.0%	0.0%	FP subtract double	sub.d	0.0%	4.9%
add unsigned	addu	5.2%	3.5%	FP multiply double	mul.d	0.0%	15.0%
add immediate unsigned	addiu	9.0%	7.2%	FP divide double	div.d	0.0%	0.2%
subtract unsigned	subu	2.2%	0.6%	FP add single	add.s	0.0%	1.5%
AND	AND	0.2%	0.1%	FP subtract single	sub.s	0.0%	1.8%
AND immediate	ANDi	0.7%	0.2%	FP multiply single	mul.s	0.0%	2.4%
OR	OR	4.0%	1.2%	FP divide single	div.s	0.0%	0.2%
OR immediate	ORi	1.0%	0.2%	load word to FP double	l.d	0.0%	17.5%
NOR	NOR	0.4%	0.2%	store word to FP double	s.d	0.0%	4.9%
shift left logical	sll	4.4%	1.9%	load word to FP single	l.s	0.0%	4.2%
shift right logical	srl	1.1%	0.5%	store word to FP single	s.s	0.0%	1.1%
load upper immediate	lui	3.3%	0.5%	branch on floating-point true	bc1t	0.0%	0.2%
load word	lw	18.6%	5.8%	branch on floating-point false	bc1f	0.0%	0.2%
store word	sw	7.6%	2.0%	floating-point compare double	c.x.d	0.0%	0.6%
load byte	lbu	3.7%	0.1%	multiply	mul	0.0%	0.2%
store byte	sb	0.6%	0.0%	shift right arithmetic	sra	0.5%	0.3%
branch on equal (zero)	beq	8.6%	2.2%	load half	lhu	1.3%	0.0%
branch on not equal (zero)	bne	8.4%	1.4%	store half	sh	0.1%	0.0%
jump and link	jal	0.7%	0.2%				
jump register	jr	1.1%	0.2%				
set less than	slt	9.9%	2.3%				
set less than immediate	slti	3.1%	0.3%				
set less than unsigned	sltu	3.4%	0.8%				
set less than imm. uns.	sltiu	1.1%	0.1%				

FIGURE 3.28 The frequency of the MIPS Instructions for SPEC CPU2006 Integer and floating point. All instructions that accounted for at least 0.2% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here.

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



Historical Perspective and Further Reading

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 x86. See the rest of [Section 3.11](#) online.

3.12 Exercises

3.1 [5] <§3.2> What is $5ED4 - 07A4$ when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

3.2 [5] <§3.2> What is $5ED4 - 07A4$ when these values represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

3.3 [10] <§3.2> Convert $5ED4$ into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

3.4 [5] <§3.2> What is $4365 - 3412$ when these values represent unsigned 12-bit octal numbers? The result should be written in octal. Show your work.

3.5 [5] <§3.2> What is $4365 - 3412$ when these values represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

3.6 [5] <§3.2> Assume 185 and 122 are unsigned 8-bit decimal integers. Calculate $185 - 122$. Is there overflow, underflow, or neither?

3.7 [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate $185 + 122$. Is there overflow, underflow, or neither?

3.8 [5] <§3.2> Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate $185 - 122$. Is there overflow, underflow, or neither?

3.9 [10] <§3.2> Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate $151 + 214$ using saturating arithmetic. The result should be written in decimal. Show your work.

3.10 [10] <§3.2> Assume 151 and 214 are signed 8-bit decimal integers stored in two's complement format. Calculate $151 - 214$ using saturating arithmetic. The result should be written in decimal. Show your work.

3.11 [10] <§3.2> Assume 151 and 214 are unsigned 8-bit integers. Calculate $151 + 214$ using saturating arithmetic. The result should be written in decimal. Show your work.

3.12 [20] <§3.3> Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.

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

3.13 [20] <§3.3> Using a table similar to that shown in [Figure 3.6](#), calculate the product of the hexadecimal unsigned 8-bit integers 62 and 12 using the hardware described in [Figure 3.5](#). You should show the contents of each register on each step.

3.14 [10] <§3.3> Calculate the time necessary to perform a multiply using the approach given in [Figures 3.3 and 3.4](#) if an integer is 8 bits wide and each step of the operation takes 4 time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

3.15 [10] <§3.3> Calculate the time necessary to perform a multiply using the approach described in the text (31 adders stacked vertically) if an integer is 8 bits wide and an adder takes 4 time units.

3.16 [20] <§3.3> Calculate the time necessary to perform a multiply using the approach given in [Figure 3.7](#) if an integer is 8 bits wide and an adder takes 4 time units.

3.17 [20] <§3.3> As discussed in the text, one possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9×6 , for example, can be written $(2 \times 2 \times 2 + 1) \times 6$, we can calculate 9×6 by shifting 6 to the left 3 times and then adding 6 to that result. Show the best way to calculate $0 \times 33 \times 0 \times 55$ using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

3.18 [20] <§3.4> Using a table similar to that shown in [Figure 3.10](#), calculate 74 divided by 21 using the hardware described in [Figure 3.8](#). You should show the contents of each register on each step. Assume both inputs are unsigned 6-bit integers.

3.19 [30] <§3.4> Using a table similar to that shown in [Figure 3.10](#), calculate 74 divided by 21 using the hardware described in [Figure 3.11](#). You should show the contents of each register on each step. Assume A and B are unsigned 6-bit integers. This algorithm requires a slightly different approach than that shown in [Figure 3.9](#). You will want to think hard about this, do an experiment or two, or else go to the web to figure out how to make this work correctly. (Hint: one possible solution involves using the fact that [Figure 3.11](#) implies the remainder register can be shifted either direction.)

3.20 [5] <§3.5> What decimal number does the bit pattern $0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0$ represent if it is a two's complement integer? An unsigned integer?

3.21 [10] <§3.5> If the bit pattern $0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0$ is placed into the Instruction Register, what MIPS instruction will be executed?

3.22 [10] <§3.5> What decimal number does the bit pattern $0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0 \times 0$ represent if it is a floating point number? Use the IEEE 754 standard.

3.23 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

3.24 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

3.25 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming it was stored using the single precision IBM format (base 16, instead of base 2, with 7 bits of exponent).

3.26 [20] <§3.5> Write down the binary bit pattern to represent -1.5625×10^{-1} assuming a format similar to that employed by the DEC PDP-8 (the leftmost 12 bits are the exponent stored as a two's complement number, and the rightmost 24 bits are the fraction stored as a two's complement number). No hidden 1 is used. Comment on how the range and accuracy of this 36-bit pattern compares to the single and double precision IEEE 754 standards.

3.27 [20] <§3.5> IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent -1.5625×10^{-1} assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

3.28 [20] <§3.5> The Hewlett-Packard 2114, 2115, and 2116 used a format with the leftmost 16 bits being the fraction stored in two's complement format, followed by another 16-bit field which had the leftmost 8 bits as an extension of the fraction (making the fraction 24 bits long), and the rightmost 8 bits representing the exponent. However, in an interesting twist, the exponent was stored in sign-magnitude format with the sign bit on the far right! Write down the bit pattern to represent -1.5625×10^{-1} assuming this format. No hidden 1 is used. Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.

3.29 [20] <§3.5> Calculate the sum of 2.6125×10^1 and $4.150390625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.

3.30 [30] <§3.5> Calculate the product of -8.0546875×10^0 and $-1.79931640625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision format described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps; however, as is done in the example in the text, you can do the multiplication in human-readable format instead of using the techniques described in Exercises 3.12 through 3.14. Indicate if there is overflow or underflow. Write your answer in both the 16-bit floating point format described in Exercise 3.27 and also as a decimal number. How accurate is your result? How does it compare to the number you get if you do the multiplication on a calculator?

3.31 [30] <§3.5> Calculate by hand 8.625×10^1 divided by -4.875×10^0 . Show all the steps necessary to achieve your answer. Assume there is a guard, a round bit, and a sticky bit, and use them if necessary. Write the final answer in both the 16-bit floating point format described in Exercise 3.27 and in decimal and compare the decimal result to that which you get if you use a calculator.

3.32 [20] <§3.9> Calculate $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.33 [20] <§3.9> Calculate $3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.34 [10] <§3.9> Based on your answers to 3.32 and 3.33, does $(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}) + 1.771 \times 10^3 = 3.984375 \times 10^{-1} + (3.4375 \times 10^{-1} + 1.771 \times 10^3)$?

3.35 [30] <§3.9> Calculate $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^2$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.36 [30] <§3.9> Calculate $3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^2)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.37 [10] <§3.9> Based on your answers to 3.35 and 3.36, does $(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}) \times 1.05625 \times 10^2 = 3.41796875 \times 10^{-3} \times (6.34765625 \times 10^{-3} \times 1.05625 \times 10^2)$?

3.38 [30] <§3.9> Calculate $1.666015625 \times 10^0 \times (1.9760 \times 10^4 + -1.9744 \times 10^4)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.39 [30] <§3.9> Calculate $(1.666015625 \times 10^0 \times 1.9760 \times 10^4) + (1.666015625 \times 10^0 \times -1.9744 \times 10^4)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

3.40 [10] <§3.9> Based on your answers to 3.38 and 3.39, does $(1.666015625 \times 10^0 \times 1.9760 \times 10^4) + (1.666015625 \times 10^0 \times -1.9744 \times 10^4) = 1.666015625 \times 10^0 \times (1.9760 \times 10^4 + -1.9744 \times 10^4)$?

3.41 [10] <§3.5> Using the IEEE 754 floating point format, write down the bit pattern that would represent $-1/4$. Can you represent $-1/4$ exactly?

3.42 [10] <§3.5> What do you get if you add $-1/4$ to itself 4 times? What is $-1/4 \times 4$? Are they the same? What should they be?

3.43 [10] <§3.5> Write down the bit pattern in the fraction of value $1/3$ assuming a floating point format that uses binary numbers in the fraction. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.44 [10] <§3.5> Write down the bit pattern in the fraction assuming a floating point format that uses Binary Coded Decimal (base 10) numbers in the fraction instead of base 2. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.45 [10] <§3.5> Write down the bit pattern assuming that we are using base 15 numbers in the fraction instead of base 2. (Base 16 numbers use the symbols 0–9 and A–F. Base 15 numbers would use 0–9 and A–E.) Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

3.46 [20] <§3.5> Write down the bit pattern assuming that we are using base 30 numbers in the fraction instead of base 2. (Base 16 numbers use the symbols 0–9 and A–F. Base 30 numbers would use 0–9 and A–T.) Assume there are 20 bits, and you do not need to normalize. Is this representation exact?

3.47 [45] <§§3.6, 3.7> The following C code implements a four-tap FIR filter on input array sig_in. Assume that all arrays are 16-bit fixed-point values.

```
for (i = 3;i<128;i++)
sig_out[i]=sig_in[i-3]*f[0]+sig_in[i-2]*f[1]
+sig_in[i-1]*f[2]+sig_in[i]*f[3];
```

Assume you are to write an optimized implementation this code in assembly language on a processor that has SIMD instructions and 128-bit registers. Without knowing the details of the instruction set, briefly describe how you would implement this code, maximizing the use of sub-word operations and minimizing the amount of data that is transferred between registers and memory. State all your assumptions about the instructions you use.

§3.2, page 182: 2.
§3.5, page 221: 3.

**Answers to
Check Yourself**

4

In a major matter, no details are small.

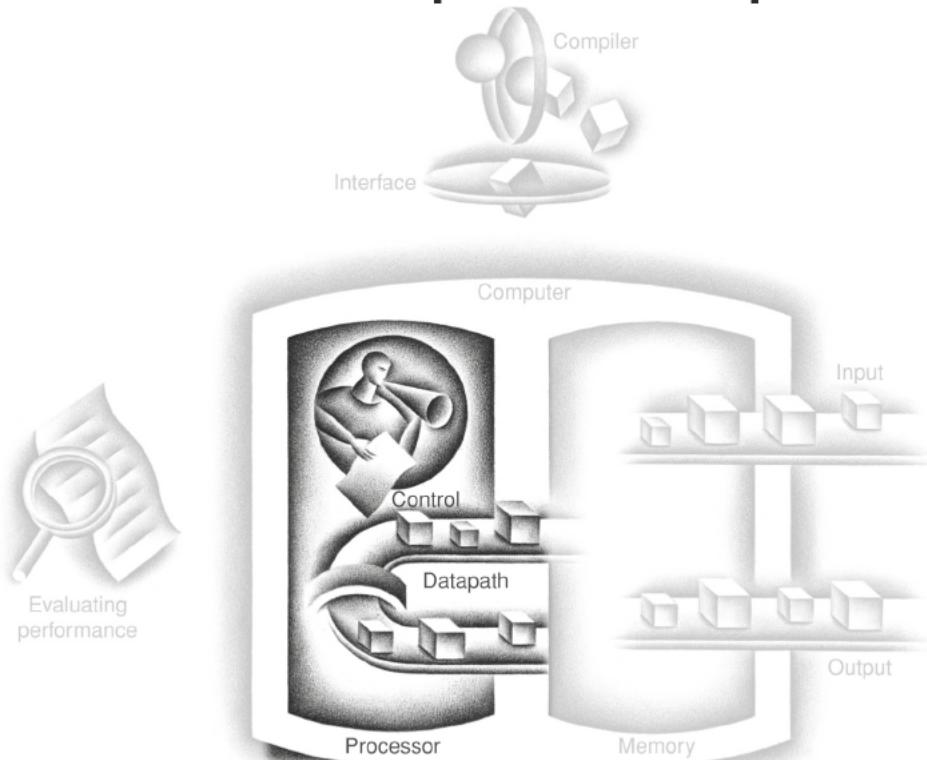
French Proverb

The Processor

- 4.1 Introduction** 244
- 4.2 Logic Design Conventions** 248
- 4.3 Building a Datapath** 251
- 4.4 A Simple Implementation Scheme** 259
- 4.5 An Overview of Pipelining** 272
- 4.6 Pipelined Datapath and Control** 286
- 4.7 Data Hazards: Forwarding versus Stalling** 303
- 4.8 Control Hazards** 316
- 4.9 Exceptions** 325
- 4.10 Parallelism via Instructions** 332

- 4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines** 344
- 4.12 Going Faster: Instruction-Level Parallelism and Matrix Multiply** 351
- 4.13 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations** 354
- 4.14 Fallacies and Pitfalls** 355
- 4.15 Concluding Remarks** 356
- 4.16 Historical Perspective and Further Reading** 357
- 4.17 Exercises** 357

Five Classic Components of a Computer



4.1

Introduction

Chapter 1 explains that the performance of a computer is determined by three key factors: instruction count, clock cycle time, and *clock cycles per instruction* (CPI). Chapter 2 explains that the compiler and the instruction set architecture determine the instruction count required for a given program. However, the implementation of the processor determines both the clock cycle time and the number of clock cycles per instruction. In this chapter, we construct the datapath and control unit for two different implementations of the MIPS instruction set.



PIPELINING

This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview in this section. It is followed by a section that builds up a datapath and constructs a simple version of a processor sufficient to implement an instruction set like MIPS. The bulk of the chapter covers a more realistic **pipelined** MIPS implementation, followed by a section that develops the concepts necessary to implement more complex instruction sets, like the x86.

For the reader interested in understanding the high-level interpretation of instructions and its impact on program performance, this initial section and Section 4.5 present the basic concepts of pipelining. Recent trends are covered in Section 4.10, and Section 4.11 describes the recent Intel Core i7 and ARM Cortex-A8 architectures. Section 4.12 shows how to use instruction-level parallelism to more than double the performance of the matrix multiply from Section 3.8. These sections provide enough background to understand the pipeline concepts at a high level.

For the reader interested in understanding the processor and its performance in more depth, Sections 4.3, 4.4, and 4.6 will be useful. Those interested in learning how to build a processor should also cover 4.2, 4.7, 4.8, and 4.9. For readers with an interest in modern hardware design, [Section 4.13](#) describes how hardware design languages and CAD tools are used to implement hardware, and then how to use a hardware design language to describe a pipelined implementation. It also gives several more illustrations of how pipelining hardware executes.

A Basic MIPS Implementation

We will be examining an implementation that includes a subset of the core MIPS instruction set:

- The memory-reference instructions *load word* (*lw*) and *store word* (*sw*)
- The arithmetic-logical instructions add, sub, AND, OR, and slt
- The instructions *branch equal* (*beq*) and *jump* (*j*), which we add last

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions.

However, it illustrates the key principles used in creating a datapath and designing the control. The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the computer. Many of the key design principles introduced in Chapter 1 can be illustrated by looking at the implementation, such as *Simplicity favors regularity*. In addition, most concepts used to implement the MIPS subset in this chapter are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors.

An Overview of the Implementation

In Chapter 2, we looked at the core MIPS instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction. The simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.

Figure 4.1 shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

First, in several places, [Figure 4.1](#) shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction. In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. Appendix B describes the multiplexor, which selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

The second omission in [Figure 4.1](#) is that several of the units must be controlled depending on the type of instruction. For example, the data memory must read

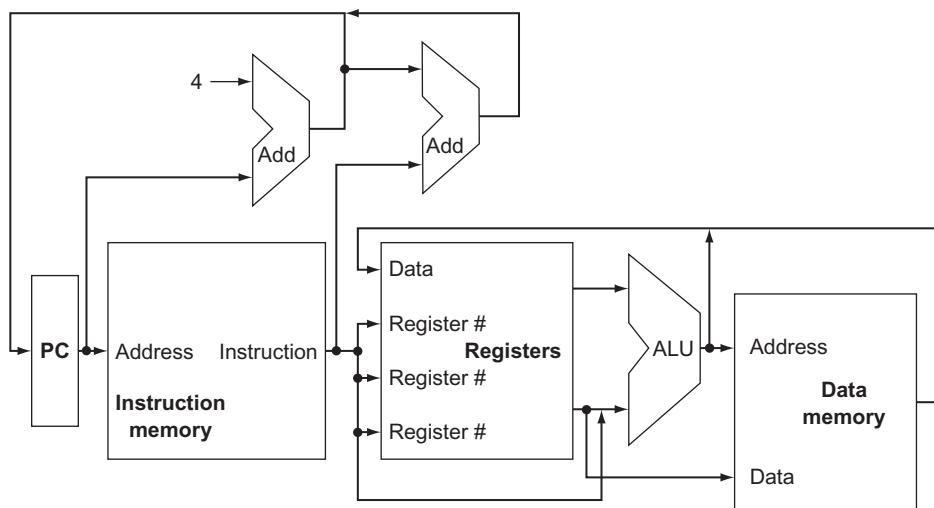


FIGURE 4.1 An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

on a load and written on a store. The register file must be written only on a load or an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations. (Appendix B describes the detailed design of the ALU.) Like the multiplexors, control lines that are set on the basis of various fields in the instruction direct these operations.

Figure 4.2 shows the datapath of Figure 4.1 with the three required multiplexors added, as well as control lines for the major functional units. A *control unit*, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors. The third multiplexor,

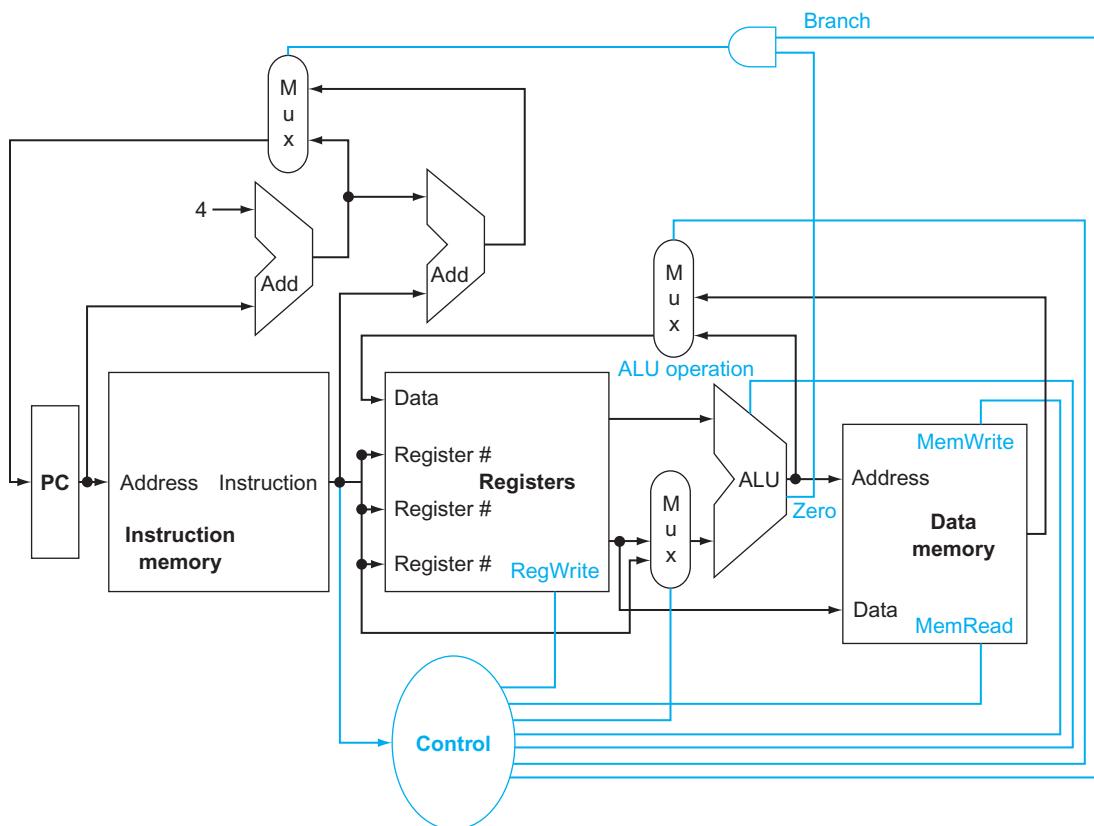


FIGURE 4.2 The basic implementation of the MIPS subset, including the necessary multiplexors and control lines. The top multiplexor (“Mux”) controls what value replaces the PC ($PC + 4$ or the branch destination address); the multiplexor is controlled by the gate that “ANDs” together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

which determines whether $\text{PC} + 4$ or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a beq instruction. The regularity and simplicity of the MIPS instruction set means that a simple decoding process can be used to determine how to set the control lines.

In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, enhance a control unit to control what actions are taken for different instruction classes. Sections 4.3 and 4.4 describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of [Figures 4.1 and 4.2](#). In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

While easier to understand, this approach is not practical, since the clock cycle must be severely stretched to accommodate the longest instruction. After designing the control for this simple computer, we will look at pipelined implementation with all its complexities, including exceptions.

Check Yourself How many of the five classic components of a computer—shown on page 243—do [Figures 4.1 and 4.2](#) include?

4.2

Logic Design Conventions

To discuss the design of a computer, we must decide how the hardware logic implementing the computer will operate and how the computer is clocked. This section reviews a few key ideas in digital logic that we will use extensively in this chapter. If you have little or no background in digital logic, you will find it helpful to read [Appendix B](#) before continuing.

The datapath elements in the MIPS implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state. The elements that operate on data values are all **combinational**, which means that their outputs depend only on the current inputs. Given the same input, a combinational element always produces the same output. The ALU shown in [Figure 4.1](#) and discussed in [Appendix B](#) is an example of a combinational element. Given a set of inputs, it always produces the same output because it has no internal storage.

Other elements in the design are not combinational, but instead contain *state*. An element contains state if it has some internal storage. We call these elements **state elements** because, if we pulled the power plug on the computer, we could restart it accurately by loading the state elements with the values they contained before we pulled the plug. Furthermore, if we saved and restored the state elements, it would be as if the computer had never lost power. Thus, these state elements completely characterize the computer. In [Figure 4.1](#), the instruction and data memories, as well as the registers, are all examples of state elements.

combinational element

An operational element, such as an AND gate or an ALU.

state element

A memory element, such as a register or a memory.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. For example, one of the logically simplest state elements is a D-type flip-flop (see [Appendix B](#)), which has exactly these two inputs (a value and a clock) and one output. In addition to flip-flops, our MIPS implementation uses two other types of state elements: memories and registers, both of which appear in [Figure 4.1](#). The clock is used to determine when the state element should be written; a state element can be read at any time.

Logic components that contain state are also called *sequential*, because their outputs depend on both their inputs and the contents of the internal state. For example, the output from the functional unit representing the registers depends both on the register numbers supplied and on what was written into the registers previously. The operation of both the combinational and sequential elements and their construction are discussed in more detail in [Appendix B](#).

Clocking Methodology

A **clocking methodology** defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes, because if a signal is written at the same time it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two! Computer designs cannot tolerate such unpredictability. A clocking methodology is designed to make hardware predictable.

For simplicity, we will assume an **edge-triggered clocking** methodology. An edge-triggered clocking methodology means that any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or *vice versa* (see [Figure 4.3](#)). Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements. The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

clocking methodology The approach used to determine when data is valid and stable relative to the clock.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

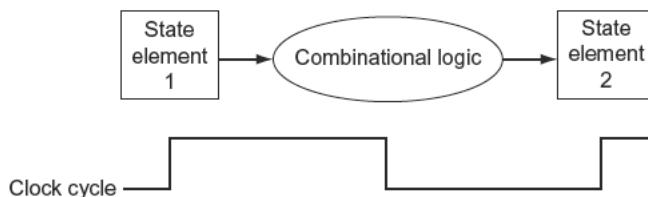


FIGURE 4.3 Combinational logic, state elements, and the clock are closely related.

In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed to be positive edge-triggered; that is, they change on the rising clock edge.

Figure 4.3 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: all signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

For simplicity, we do not show a write **control signal** when a state element is written on every active clock edge. In contrast, if a state element is not updated on every clock, then an explicit write control signal is required. Both the clock signal and the write control signal are inputs, and the state element is changed only when the write control signal is asserted and a clock edge occurs.

We will use the word **asserted** to indicate a signal that is logically high and *assert* to specify that a signal should be driven logically high, and **deasserted** or **deasserted** to represent logically low. We use the terms assert and deassert because when we implement hardware, at times 1 represents logically high and at times it can represent logically low.

An edge-triggered methodology allows us to read the contents of a register, send the value through some combinational logic, and write that register in the same clock cycle. **Figure 4.4** gives a generic example. It doesn't matter whether we assume that all writes take place on the rising clock edge (from low to high) or on the falling clock edge (from high to low), since the inputs to the combinational logic block cannot change except on the chosen clock edge. In this book we use the rising clock edge. With an edge-triggered timing methodology, there is *no* feedback within a single clock cycle, and the logic in **Figure 4.4** works correctly. In **Appendix B**, we briefly discuss additional timing constraints (such as setup and hold times) as well as other timing methodologies.

For the 32-bit MIPS architecture, nearly all of these state and logic elements will have inputs and outputs that are 32 bits wide, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 32 bits in width. The figures will indicate *buses*, which are signals wider than 1 bit, with thicker lines. At times, we will want to combine several buses to form a wider bus; for example, we may want to obtain a 32-bit bus by combining two 16-bit buses. In such cases, labels on the bus lines will make it

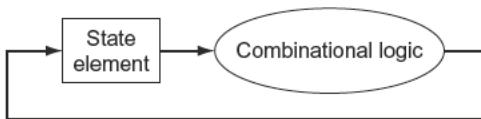


FIGURE 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

clear that we are concatenating buses to form a wider bus. Arrows are also added to help clarify the direction of the flow of data between elements. Finally, **color** indicates a control signal as opposed to a signal that carries data; this distinction will become clearer as we proceed through this chapter.

True or false: Because the register file is both read and written on the same clock cycle, any MIPS datapath using edge-triggered writes must have more than one copy of the register file.

Check Yourself

Elaboration: There is also a 64-bit version of the MIPS architecture, and, naturally enough, most paths in its implementation would be 64 bits wide.

4.3

Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions. Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.

Figure 4.5a shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 4.5b also shows the **program counter (PC)**, which as we saw in Chapter 2 is a register that holds the address of the current instruction. Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU described in detail in  Appendix B simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label *Add*, as in Figure 4.5, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure 4.6 shows how to combine the three elements from Figure 4.5 to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

Now let's consider the R-format instructions (see Figure 2.20 on page 120). They all read two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes add, sub, AND, OR, and slt,



datapath element

A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

program counter (PC)

The register containing the address of the instruction in the program being executed.

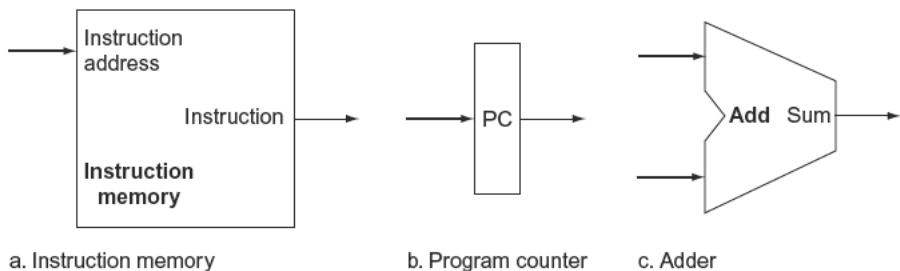


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

which were introduced in Chapter 2. Recall that a typical instance of such an instruction is `add $t1,$t2,$t3`, which reads `$t2` and `$t3` and writes `$t1`.

register file A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

The processor's 32 general-purpose registers are stored in a structure called a **register file**. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.

R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the *register number* to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the *data* to be written into the register. The register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge. Figure 4.7a shows the result; we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

Figure 4.7b shows the ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The 4-bit control signal of the ALU is described in detail in [Appendix B](#); we will review the ALU control shortly when we need to know how to set it.

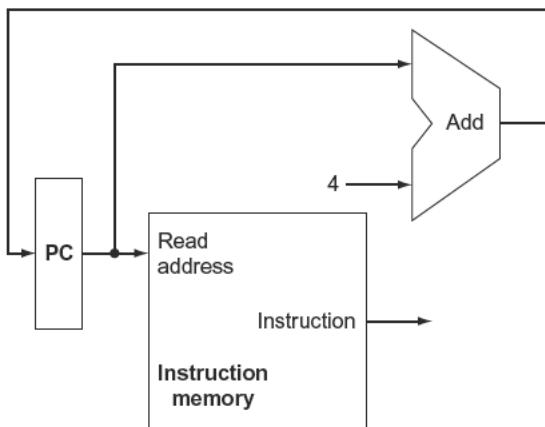


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

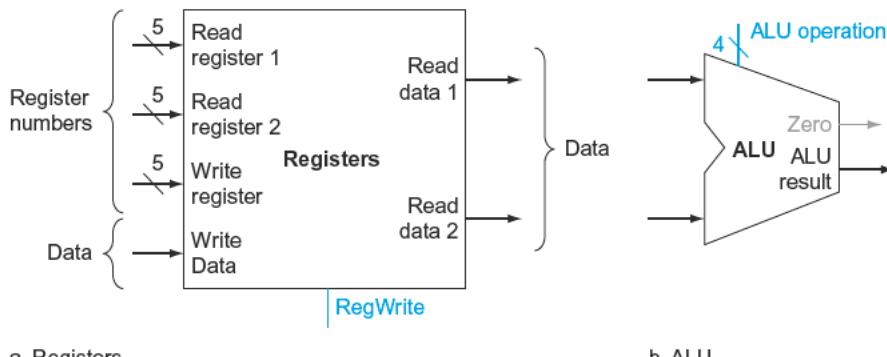


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of [Appendix B](#). The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in [Appendix B](#). We will use the Zero detection output of the ALU shortly to implement branches. The overflow output will not be needed until Section 4.9, when we discuss exceptions; we omit it until then.

Next, consider the MIPS load word and store word instructions, which have the general form `lw $t1, offset_value($t2)` or `sw $t1, offset_value($t2)`. These instructions compute a memory address by adding the base register, which is `$t2`, to the 16-bit signed offset field contained in the instruction. If the instruction is a store, the value to be stored must also be read from the register file where it resides in `$t1`. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is `$t1`. Thus, we will need both the register file and the ALU from [Figure 4.7](#).

In addition, we will need a unit to **sign-extend** the 16-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory. [Figure 4.8](#) shows these two elements.

The `beq` instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the **branch target address** relative to the branch instruction address. Its form is `beq $t1, $t2, offset`. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC. There are two details in the definition of branch instructions (see Chapter 2) to which we must pay attention:

- The instruction set architecture specifies that the base for the branch address calculation is the address of the instruction following the branch. Since we compute $\text{PC} + 4$ (the address of the next instruction) in the instruction fetch datapath, it is easy to use this value as the base for computing the branch target address.
- The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of 4.

To deal with the latter complication, we will need to shift the offset field by 2.

As well as computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address. When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch** is **taken**. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch** is **not taken**.

Thus, the branch datapath must do two operations: compute the branch target address and compare the register contents. (Branches also affect the instruction fetch portion of the datapath, as we will deal with shortly.) [Figure 4.9](#) shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes a sign extension unit, from [Figure 4.8](#) and an adder. To perform the compare, we need to use the register file shown in [Figure 4.7a](#) to supply the two register operands (although we will not need to write into the register file). In addition, the comparison can be done using the ALU we

sign-extend To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

branch target

address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

branch taken

A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.

branch not taken or (untaken branch)

A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

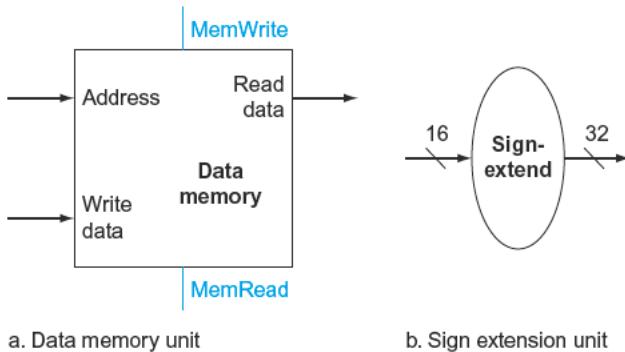


FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit.

The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section B.8 of [Appendix B](#) for further discussion of how real memory chips work.

designed in [Appendix B](#). Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits. Simply concatenating 00 to the jump offset accomplishes this shift, as described in Chapter 2.

Elaboration: In the MIPS instruction set, **branches are delayed**, meaning that the instruction immediately following the branch is always executed, *independent* of whether the branch condition is true or false. When the condition is false, the execution looks like a normal branch. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address. The motivation for delayed branches arises from how pipelining affects branches (see Section 4.8). For simplicity, we generally ignore delayed branches in this chapter and implement a nondelayed beq instruction.

branch A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

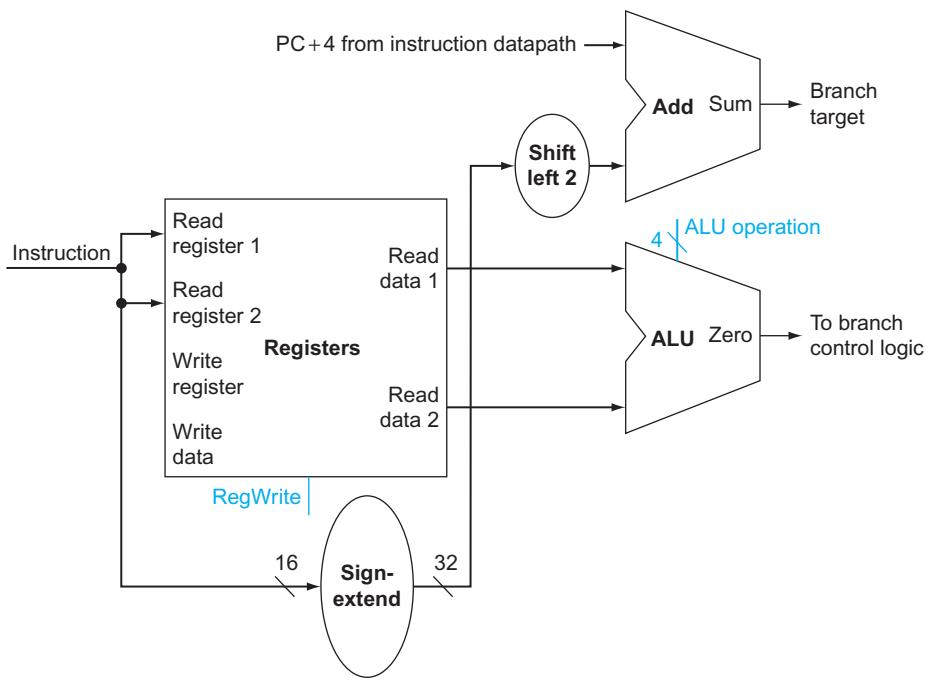


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled Shift left 2 is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the “shift” is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only “sign bits.” Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Creating a Single Datapath

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. This simplest datapath will attempt to execute all instructions in one clock cycle. This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

Building a Datapath

EXAMPLE

The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:

- The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the sign-extended 16-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

Show how to build a datapath for the operational portion of the memory-reference and arithmetic-logical instructions that uses a single register file and a single ALU to handle both types of instructions, adding any necessary multiplexors.

To create a datapath with only a single register file and a single ALU, we must support two different sources for the second ALU input, as well as two different sources for the data stored into the register file. Thus, one multiplexor is placed at the ALU input and another at the data input to the register file. [Figure 4.10](#) shows the operational portion of the combined datapath.

ANSWER

Now we can combine all the pieces to make a simple datapath for the core MIPS architecture by adding the datapath for instruction fetch ([Figure 4.6](#)), the datapath from R-type and memory instructions ([Figure 4.10](#)), and the datapath for branches ([Figure 4.9](#)). [Figure 4.11](#) shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU for comparison of the register operands, so we must keep the adder from [Figure 4.9](#) for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address ($PC + 4$) or the branch target address to be written into the PC.

Now that we have completed this simple datapath, we can add the control unit. The control unit must be able to take inputs and generate a write signal for each state element, the selector control for each multiplexor, and the ALU control. The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

- I. Which of the following is correct for a load instruction? Refer to [Figure 4.10](#).
 - a. MemtoReg should be set to cause the data from memory to be sent to the register file.

Check Yourself

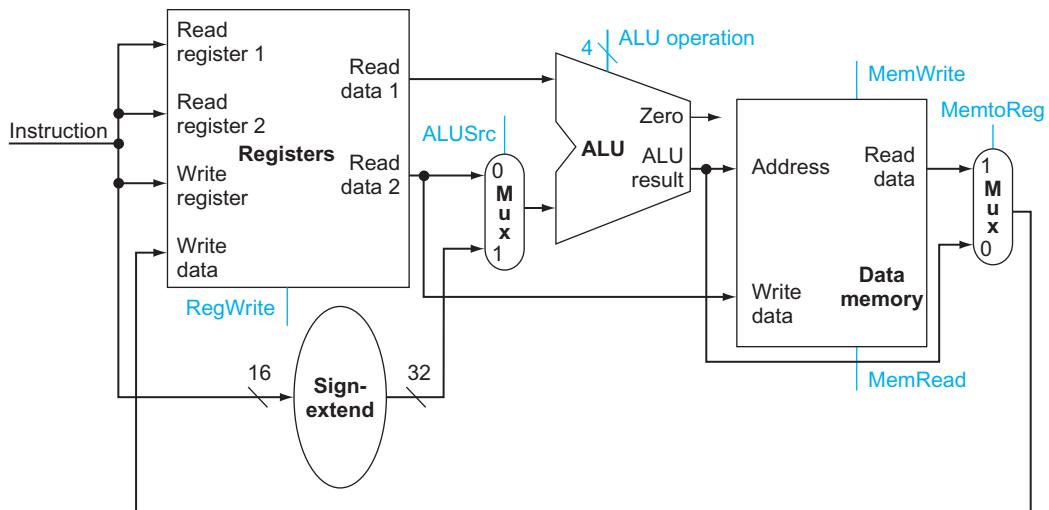


FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

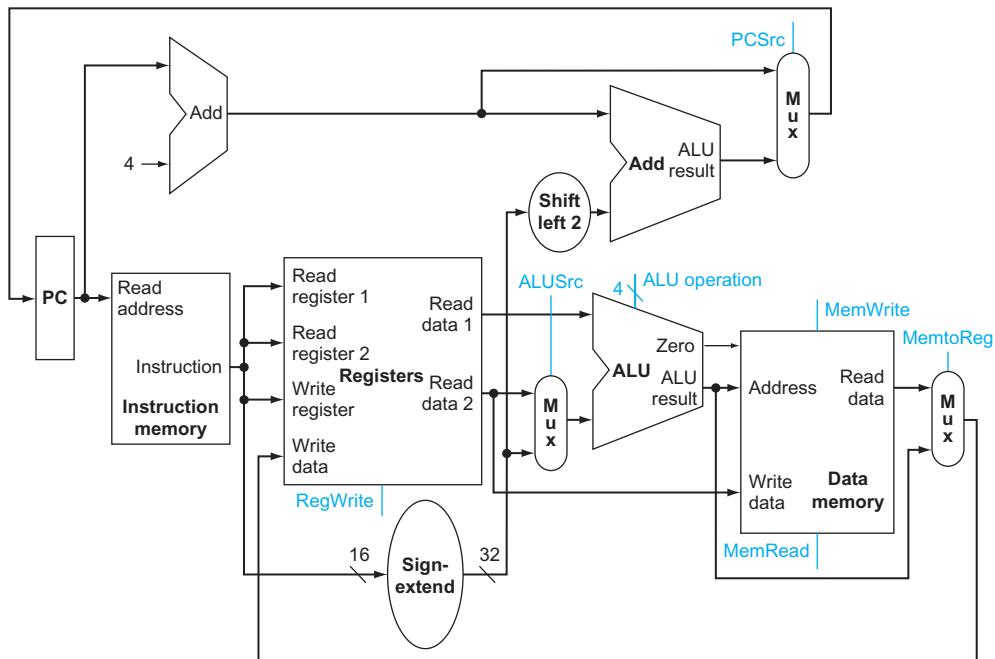


FIGURE 4.11 The simple datapath for the core MIPS architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store word, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches. The support for jumps will be added later.

- b. MemtoReg should be set to cause the correct register destination to be sent to the register file.
 - c. We do not care about the setting of MemtoReg for loads.
- II. The single-cycle datapath conceptually described in this section *must* have separate instruction and data memories, because
- a. the formats of data and instructions are different in MIPS, and hence different memories are needed.
 - b. having separate memories is less expensive.
 - c. the processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle

4.4

A Simple Implementation Scheme

In this section, we look at what might be thought of as the simplest possible implementation of our MIPS subset. We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers *load word* (lw), *store word* (sw), *branch equal* (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than. We will later enhance the design to include a jump instruction (j).

The ALU Control

The MIPS ALU in [Appendix B](#) defines the 6 following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR is needed for other parts of the MIPS instruction set not found in the subset we are implementing.) For load word and store word instructions, we use the ALU to compute the memory address by addition. For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field

in the low-order bits of the instruction (see Chapter 2). For branch equal, the ALU must perform a subtraction.

We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations shown previously.

In [Figure 4.12](#), we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

This style of using multiple levels of decoding—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique. Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. Such optimizations are important, since the speed of the control unit is often critical to clock cycle time.

There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the four ALU operation control bits. Because only a small number of the 64 possible values of the function field are of interest and the function field is used only when the ALUOp bits equal 10, we can use a small piece of logic that recognizes the subset of possible values and causes the correct setting of the ALU control bits.

As a step in designing this logic, it is useful to create a truth table for the interesting combinations of the function code field and the ALUOp bits, as we've

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type Instruction. The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input. See  [Appendix B](#).

done in Figure 4.13; this **truth table** shows how the 4-bit ALU control is set depending on these two input fields. Since the full truth table is very large ($2^8 = 256$ entries) and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value. Throughout this chapter, we will use this practice of showing only the truth table entries for outputs that must be asserted and not showing those that are all deasserted or don't care. (This practice has a disadvantage, which we discuss in Section D.2 of [Appendix D](#).)

Because in many instances we do not care about the values of some of the **inputs**, and because we wish to keep the tables compact, we also include **don't-care terms**. A don't-care term in this truth table (represented by an X in an input column) indicates that the output does not depend on the value of the input corresponding to that column. For example, when the ALUOp bits are 00, as in the first row of Figure 4.13, we always set the ALU control to 0010, independent of the function code. In this case, then, the function code inputs will be don't cares in this line of the truth table. Later, we will see examples of another type of don't-care term. If you are unfamiliar with the concept of don't-care terms, see [Appendix B](#) for more information.

Once the truth table has been constructed, it can be optimized and then turned into gates. This process is completely mechanical. Thus, rather than show the final steps here, we describe the process and the result in Section D.2 of [Appendix D](#).

Designing the Main Control Unit

Now that we have described how to design an ALU that uses the function code and a 2-bit signal as its control inputs, we can return to looking at the rest of the control. To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in Figure 4.11. To understand how to connect the fields of an instruction to the datapath, it is useful to review

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

truth table From logic, a representation of a logical operation by listing all the values of the inputs and then in each case showing what the resulting outputs should be.

don't-care term An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0
a. R-type instruction						
Field	35 or 43	rs	rt		address	
Bit positions	31:26	25:21	20:16		15:0	
b. Load or store instruction						
Field	4	rs	rt		address	
Bit positions	31:26	25:21	20:16		15:0	
c. Branch instruction						

FIGURE 4.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats. The jump instructions use another format, which we will discuss shortly.

(a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, AND, OR, and s_{lt}. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35_{ten}) and store (opcode = 43_{ten}) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address.

the formats of the three instruction classes: the R-type, branch, and load-store instructions. Figure 4.14 shows these formats.

There are several major observations about this instruction format that we will rely on:

- The op field, which as we saw in Chapter 2 is called the **opcode**, is always contained in bits 31:26. We will refer to this field as Op[5:0].
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).
- The 16-bit offset for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

The first design principle from Chapter 2—*simplicity favors regularity*—pays off here in specifying control.

opcode The field that denotes the operation and format of an instruction.

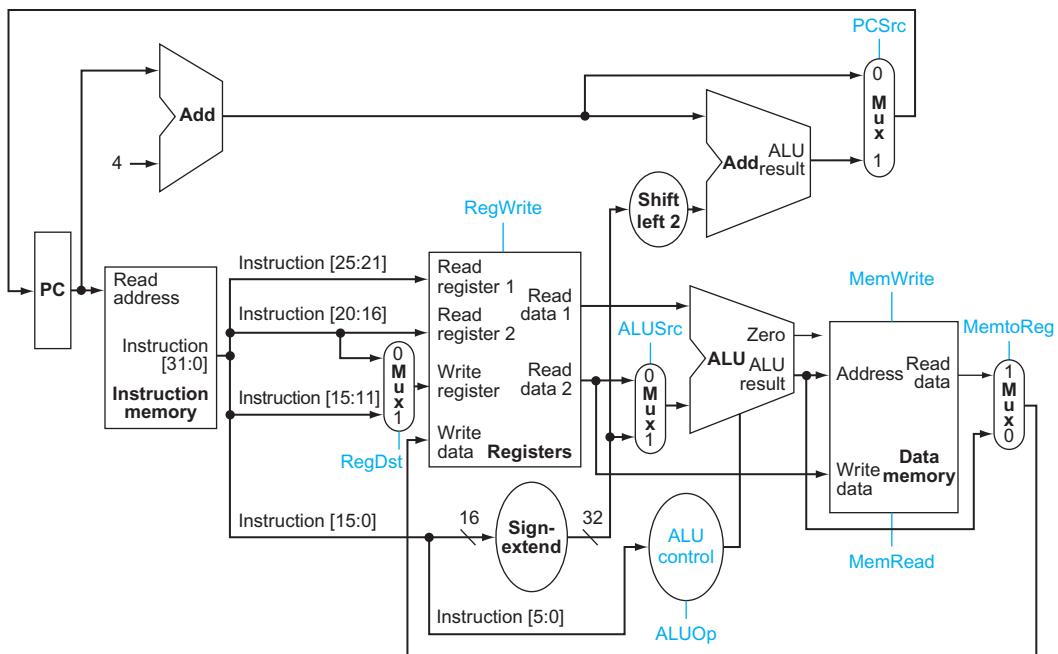


FIGURE 4.15 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Using this information, we can add the instruction labels and extra multiplexor (for the Write register number input of the register file) to the simple datapath. Figure 4.15 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a single control line.

Figure 4.15 shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution. Figure 4.16 describes the function of these seven control lines.

Now that we have looked at the function of each of the control signals, we can look at how to set them. The control unit can set all but one of the control signals based solely on the opcode field of the instruction. The PCSrc control line is the exception. That control line should be asserted if the instruction is branch on equal (a decision that the control unit can make) *and* the Zero output of the ALU, which is used for equality comparison, is asserted. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call *Branch*, with the Zero signal out of the ALU.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See [Appendix B](#) for further discussion of this problem.)

These nine control signals (seven from [Figure 4.16](#) and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26. [Figure 4.17](#) shows the datapath with the control unit and the control signals.

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to define the control function informally. Because the setting of the control lines depends only on the opcode, we define whether each control signal should be 0, 1, or don't care (X) for each of the opcode values. [Figure 4.18](#) defines how the control signals should be set for each opcode; this information follows directly from [Figures 4.12, 4.16, and 4.17](#).

Operation of the Datapath

With the information contained in [Figures 4.16 and 4.18](#), we can design the control unit logic, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath. The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

[Figure 4.19](#) shows the operation of the datapath for an R-type instruction, such as add \$t1,\$t2,\$t3. Although everything occurs in one clock cycle, we can

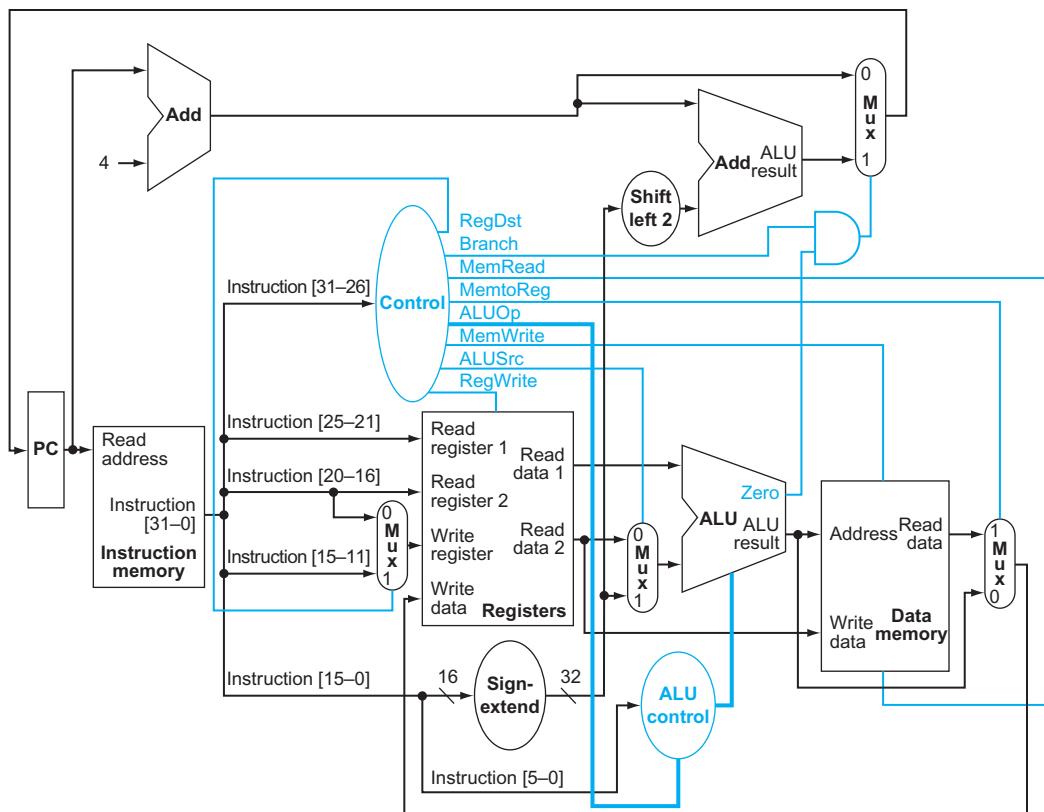


FIGURE 4.17 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

think of four steps to execute the instruction; these steps are ordered by the flow of information:

1. The instruction is fetched, and the PC is incremented.
2. Two registers, \$t2 and \$t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

FIGURE 4.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (add, sub, AND, OR, and s_tt). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0; since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

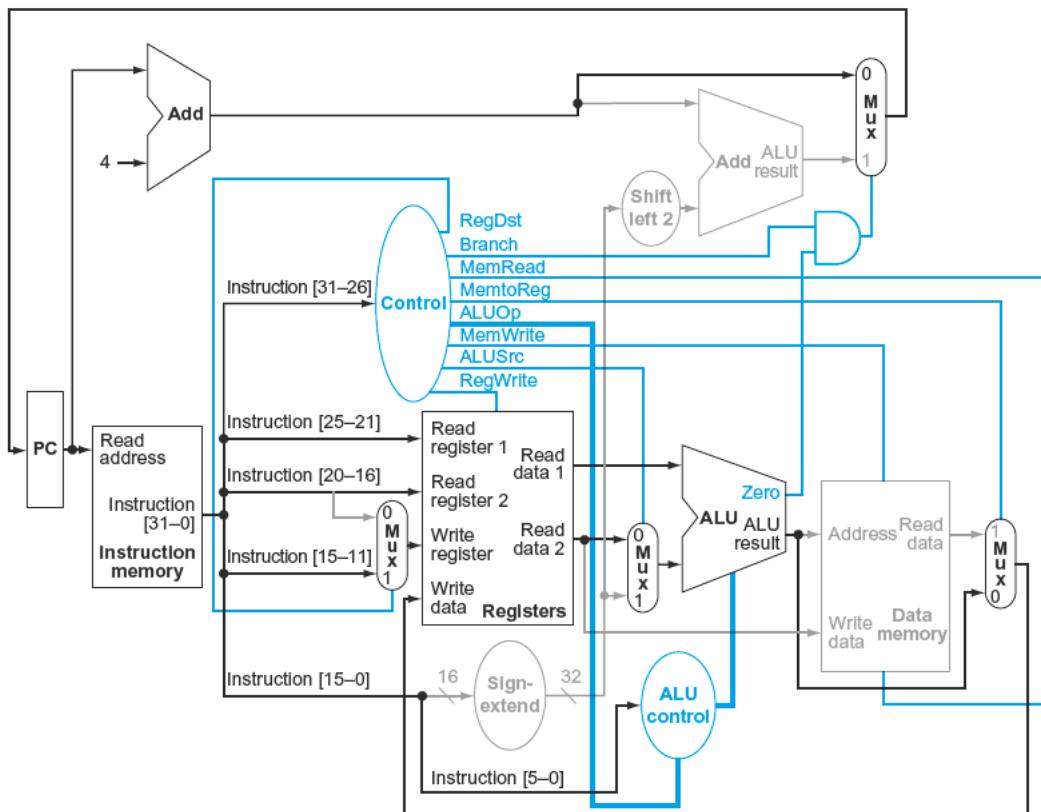


FIGURE 4.19 The datapath In operation for an R-type Instruction, such as add \$t1,\$t2,\$t3. The control lines, datapath units, and connections that are active are highlighted.

- The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register (\$t1).

Similarly, we can illustrate the execution of a load word, such as

```
lw $t1, offset($t2)
```

in a style similar to Figure 4.19. Figure 4.20 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

- An instruction is fetched from the instruction memory, and the PC is incremented.
- A register (\$t2) value is read from the register file.

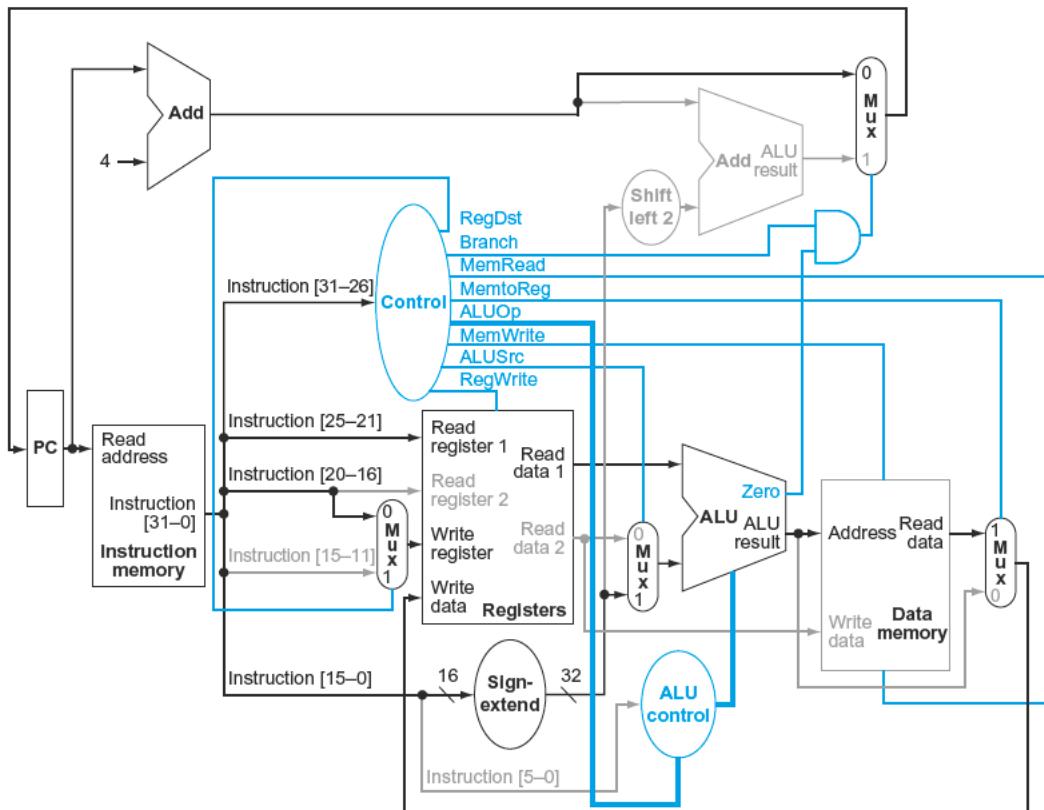


FIGURE 4.20 The datapath In operation for a load Instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction (\$t1).

Finally, we can show the operation of the branch-on-equal instruction, such as `beq $t1, $t2, offset`, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with $\text{PC} + 4$ or the branch target address. [Figure 4.21](#) shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.

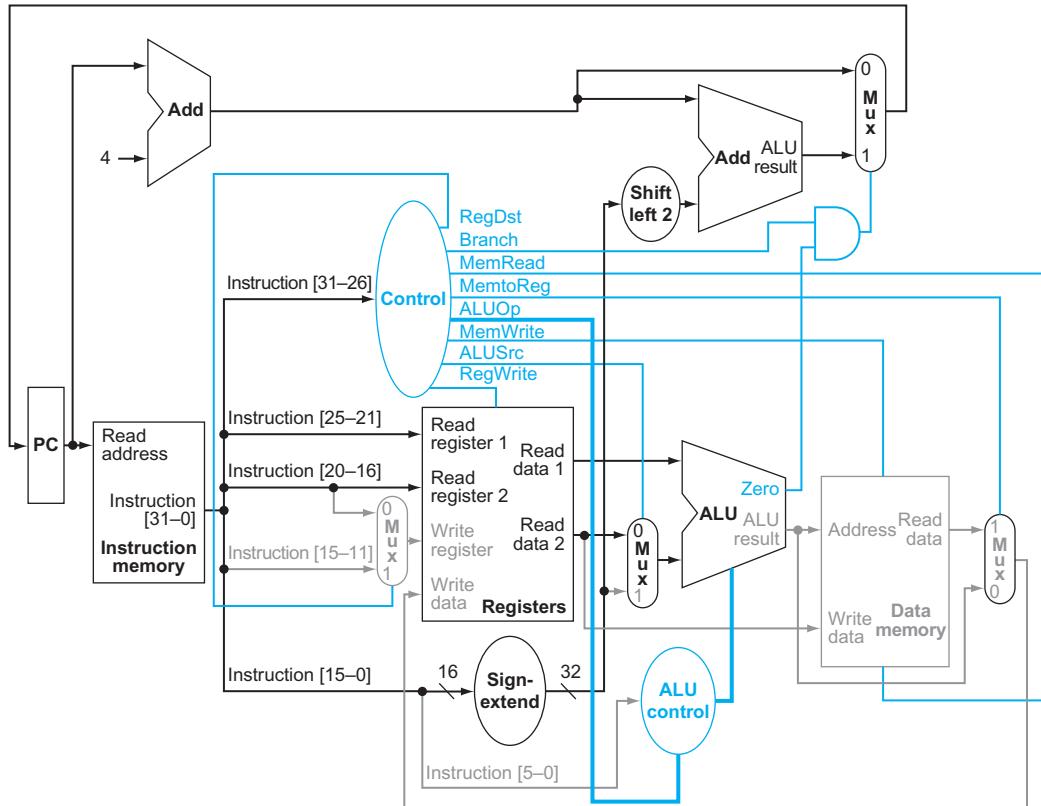


FIGURE 4.21 The datapath in operation for a branch-on-equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

2. Two registers, \$t1 and \$t2, are read from the register file.
3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.
4. The Zero result from the ALU is used to decide which adder result to store into the PC.

Finalizing Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of [Figure 4.18](#). The outputs are the control lines, and the input is the 6-bit opcode field, Op [5:0]. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.

[Figure 4.22](#) shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion. We show this final step in Section D.2 in [Appendix D](#).

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 4.22 The control function for the simple single-cycle Implementation Is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. (Remember that Op [5:0] corresponds to bits 31:26 of the instruction, which is the op field.) The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{Op5} \cdot \overline{Op2}$, since this is sufficient to distinguish the R-format instructions from lw, sw, and beq. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

single-cycle implementation Also called **single clock cycle implementation**. An implementation in which an instruction is executed in one clock cycle. While easy to understand, it is too slow to be practical.

EXAMPLE

ANSWER

Now that we have a **single-cycle implementation** of most of the MIPS core instruction set, let's add the jump instruction to show how the basic datapath and control can be extended to handle other instructions in the instruction set.

Implementing Jumps

Figure 4.17 shows the implementation of many of the instructions we looked at in Chapter 2. One class of instructions missing is that of the jump instruction. Extend the datapath and control of Figure 4.17 to include the jump instruction. Describe how to set any new control lines.

The jump instruction, shown in Figure 4.23, looks somewhat like a branch instruction but computes the target PC differently and is not conditional. Like a branch, the low-order 2 bits of a jump address are always 00_{two} . The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the instruction. The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4. Thus, we can implement a jump by storing into the PC the concatenation of

- the upper 4 bits of the current PC + 4 (these are bits 31:28 of the sequentially following instruction address)
- the 26-bit immediate field of the jump instruction
- the bits 00_{two}

Figure 4.24 shows the addition of the control for jump added to Figure 4.17. An additional multiplexor is used to select the source for the new PC value, which is either the incremented PC ($\text{PC} + 4$), the branch target PC, or the jump target PC. One additional control signal is needed for the additional multiplexor. This control signal, called *Jump*, is asserted only when the instruction is a jump—that is, when the opcode is 2.

Field	000010	address
Bit positions	31:26	25:0

FIGURE 4.23 Instruction format for the jump instruction (opcode = 2). The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

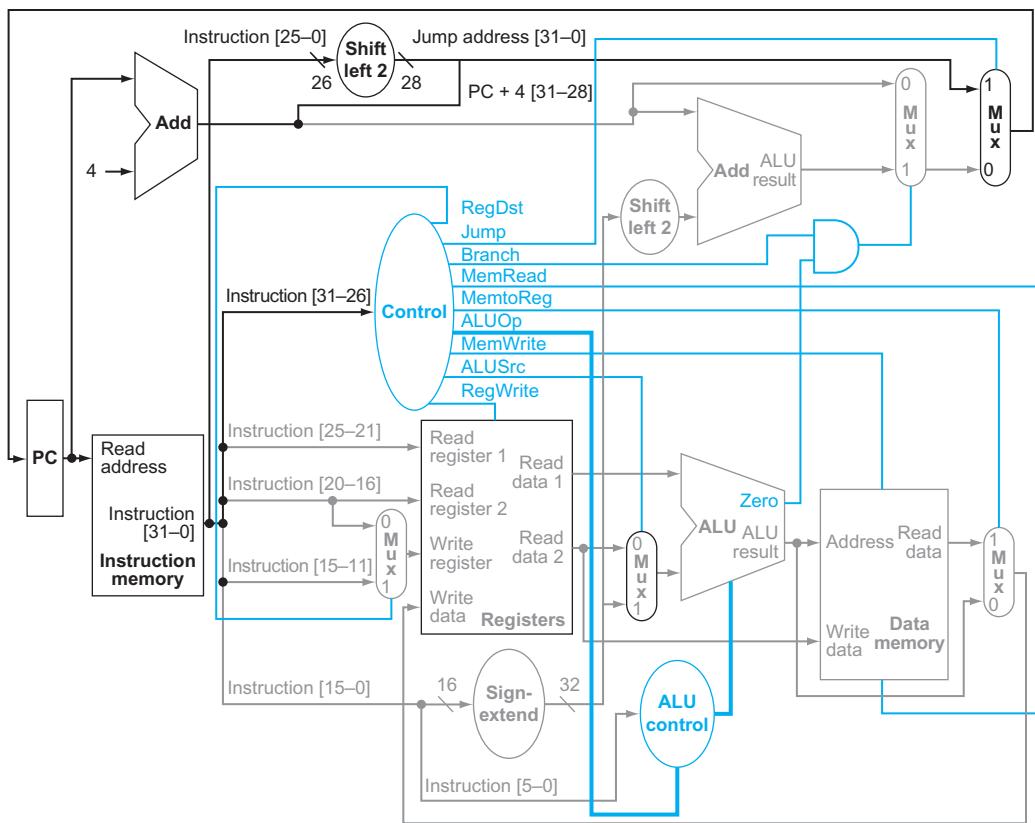


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

Why a Single-Cycle Implementation Is Not Used Today

Although the single-cycle design will work correctly, it would not be used in modern designs because it is inefficient. To see why this is so, notice that the clock cycle must have the same length for every instruction in this single-cycle design. Of course, the longest possible path in the processor determines the clock cycle. This path is almost certainly a load instruction, which uses five functional units in series: the instruction memory, the register file, the ALU, the data memory, and the register file. Although the CPI is 1 (see Chapter 1), the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early



COMMON CASE FAST

computers with very simple instruction sets did use this implementation technique. However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design wouldn't work well at all.

Because we must assume that the clock cycle is equal to the worst-case delay for all instructions, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time. A single-cycle implementation thus violates the great idea from Chapter 1 of making the **common case fast**.

In next section, we'll look at another implementation technique, called pipelining, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput. Pipelining improves efficiency by executing multiple instructions simultaneously.

Check Yourself

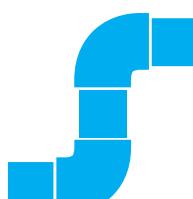
Look at the control signals in [Figure 4.22](#). Can you combine any together? Can any control signal output in the figure be replaced by the inverse of another? (Hint: take into account the don't cares.) If so, can you use one signal for the other without adding an inverter?

4.5

An Overview of Pipelining

Never waste time.
American proverb

pipelining An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.



PIPELINING

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, **pipelining** is nearly universal.

This section relies heavily on one analogy to give an overview of the pipelining terms and issues. If you are interested in just the big picture, you should concentrate on this section and then skip to Sections 4.10 and 4.11 to see an introduction to the advanced pipelining techniques used in recent processors such as the Intel Core i7 and ARM Cortex-A8. If you are interested in exploring the anatomy of a pipelined computer, this section is a good introduction to Sections 4.6 through 4.9.

Anyone who has done a lot of laundry has intuitively used pipelining. The *non-pipelined* approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, start over with the next dirty load.

The *pipelined* approach takes much less time, as [Figure 4.25](#) shows. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load. When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load

into the washer. Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer. At this point all steps—called *stages* in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.

The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour. Pipelining improves throughput of our laundry system. Hence, pipelining would not decrease the time to complete one load of laundry, but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.

If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the

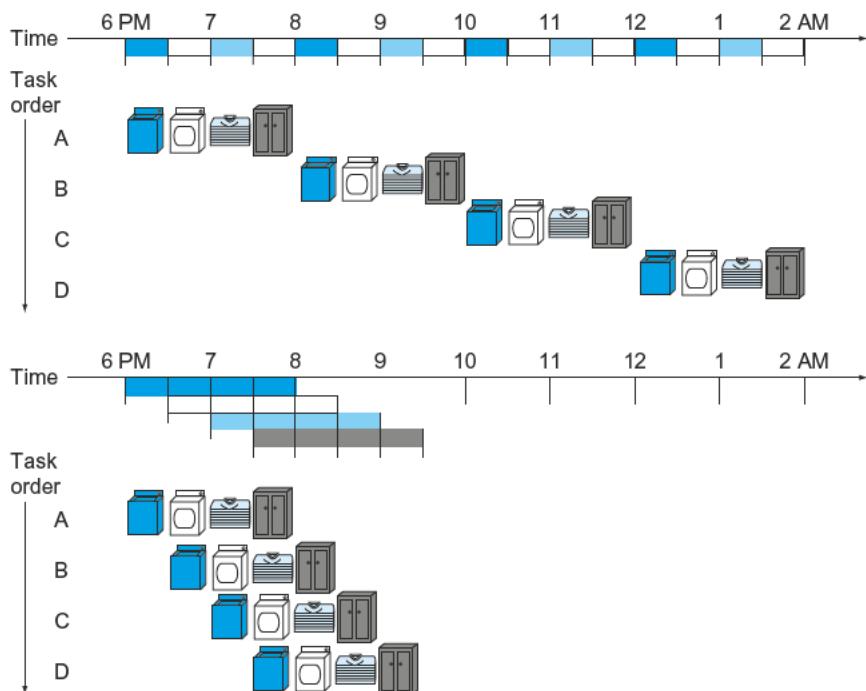


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, “folder,” and “storer” each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

pipeline, in this case four: washing, drying, folding, and putting away. Therefore, pipelined laundry is potentially four times faster than nonpipelined: 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It's only 2.3 times faster in [Figure 4.25](#), because we only show 4 loads. Notice that at the beginning and end of the workload in the pipelined version in [Figure 4.25](#), the pipeline is not completely full; this start-up and wind-down affects performance when the number of tasks is not large compared to the number of stages in the pipeline. If the number of loads is much larger than 4, then the stages will be full most of the time and the increase in throughput will be very close to 4.

The same principles apply to processors where we pipeline instruction-execution. MIPS instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register.

Hence, the MIPS pipeline we explore in this chapter has five stages. The following example shows that pipelining speeds up instruction execution just as it speeds up the laundry.

EXAMPLE

Single-Cycle versus Pipelined Performance

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to eight instructions: load word (`lw`), store word (`sw`), add (`add`), subtract (`sub`), AND (`and`), OR (`or`), set less than (`slt`), and branch on equal (`beq`).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write. In the single-cycle model, every instruction takes exactly one clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

ANSWER

[Figure 4.26](#) shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction—in [Figure 4.26](#) it is `lw`—so the time required for every instruction is 800 ps. Similarly

to Figure 4.25, Figure 4.27 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3×800 ns or 2400 ps.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps, even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×200 ps or 600 ps.

We can turn the pipelining speed-up discussion above into a formula. If the stages are perfectly balanced, then the time between instructions on the pipelined processor—assuming ideal conditions—is equal to

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle. The example shows, however, that the stages may be imperfectly balanced. Moreover, pipelining involves some overhead, the source of which will be clearer shortly. Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speed-up will be less than the number of pipeline stages.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.26 Total time for each instruction calculated from the time for each component.

This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

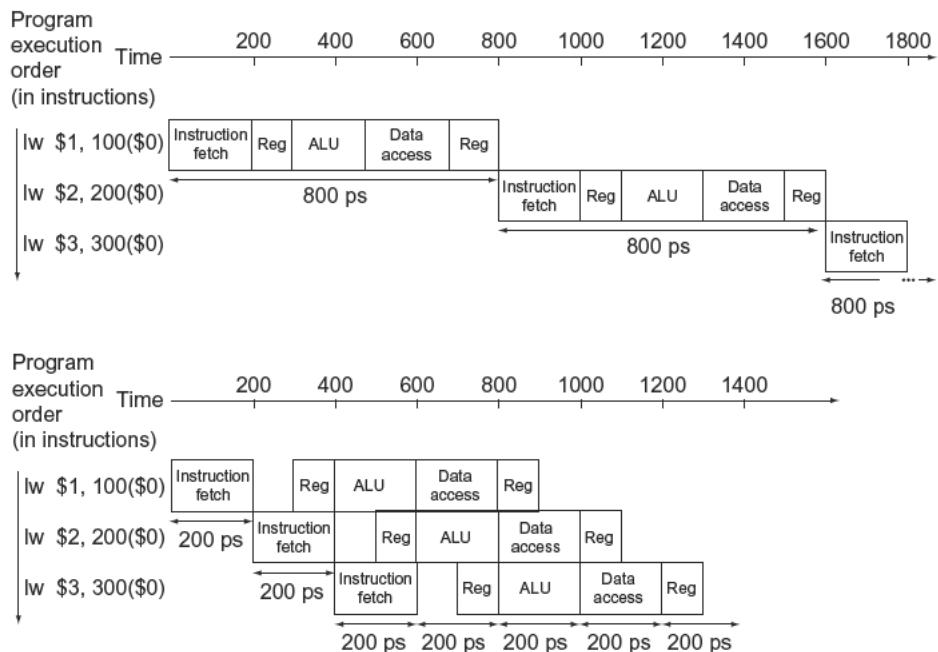


FIGURE 4.27 Single-cycle, nonpipelined execution In top versus pipelined execution In bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Moreover, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 1400 ps versus 2400 ps. Of course, this is because the number of instructions is not large. What would happen if we increased the number of instructions? We could extend the previous figures to 1,000,003 instructions. We would add 1,000,000 instructions in the pipelined example; each instruction adds 200 ps to the total execution time. The total execution time would be $1,000,000 \times 200 \text{ ps} + 1400 \text{ ps}$, or 200,001,400 ps. In the nonpipelined example, we would add 1,000,000 instructions, each taking 800 ps, so total execution time would be $1,000,000 \times 800 \text{ ps} + 2400 \text{ ps}$, or 800,002,400 ps. Under these conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

$$\frac{800,002,400 \text{ ps}}{200,001,400 \text{ ps}} \approx \frac{800 \text{ ps}}{200 \text{ ps}} \approx 4.00$$

Pipelining improves performance by *increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction*, but instruction throughput is the important metric because real programs execute billions of instructions.

Designing Instruction Sets for Pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the MIPS instruction set, which was designed for pipelined execution.

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. In an instruction set like the x86, where instructions vary from 1 byte to 15 bytes, pipelining is considerably more challenging. Recent implementations of the x86 architecture actually translate x86 instructions into simple operations that look like MIPS instructions and then pipeline the simple operations rather than the native x86 instructions! (See Section 4.10.)

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages. We will shortly see the downside of longer pipelines.

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage. If we could operate on the operands in memory, as in the x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, as discussed in Chapter 2, operands must be aligned in memory. Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

Hazards

The first hazard is called a **structural hazard**. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

structural hazard When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in [Figure 4.27](#) had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

Data Hazards

data hazard Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads must wait that have completed drying and are ready to fold as well as those that have finished washing and are ready to dry.

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (\$s0):

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

EXAMPLE

Forwarding with Two Instructions

For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in [Figure 4.28](#) to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in [Figure 4.25](#).

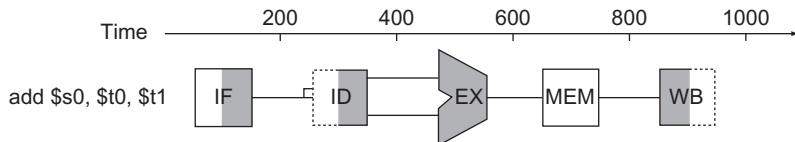


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, *MEM* has a white background because *add* does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of *ID* is shaded in the second stage because the register file is read, and the left half of *WB* is shaded in the fifth stage because the register file is written.

Figure 4.29 shows the connection to forward the value in $\$s0$ after the execution stage of the add instruction as input to the execution stage of the sub instruction.

ANSWER

In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.

Forwarding works very well and is described in detail in Section 4.7. It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of $\$s0$ instead of an add. As we can imagine from looking at Figure 4.29, the

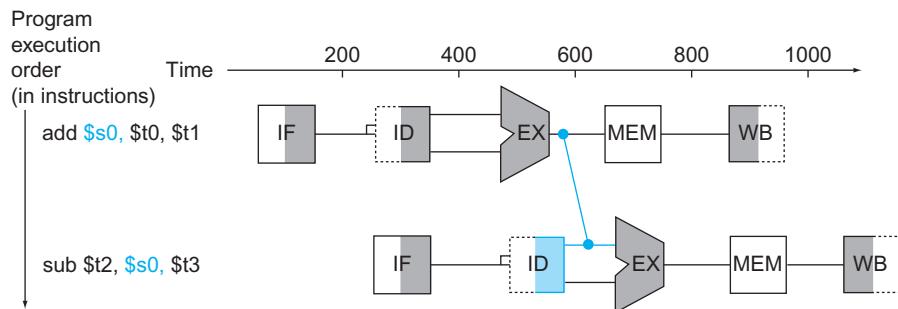


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of *add* to the input of the EX stage for *sub*, replacing the value from register $\$s0$ read in the second stage of *sub*.

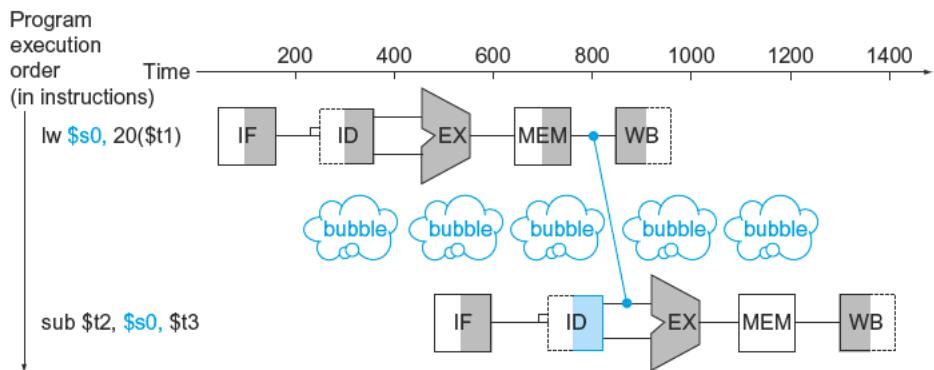


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard.

load-use data hazard

A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

pipeline stall Also called **bubble**. A stall initiated in order to resolve a hazard.

EXAMPLE

Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from `$t0`:

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add $t3, $t1,$t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add $t5, $t1,$t4
sw    $t5, 16($t0)
```

desired data would be available only *after* the fourth stage of the first instruction in the dependence, which is too late for the *input* of the third stage of `sub`. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**, as Figure 4.30 shows. This figure shows an important pipeline concept, officially called a **pipeline stall**, but often given the nickname **bubble**. We shall see stalls elsewhere in the pipeline. Section 4.7 shows how we can handle hard cases like these, using either hardware detection and stalls or software that reorders code to try to avoid load-use pipeline stalls, as this example illustrates.

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

Both add instructions have a hazard because of their respective dependence on the immediately preceding `lw` instruction. Notice that bypassing eliminates several other potential hazards, including the dependence of the first `add` on the first `lw` and any hazards for store instructions. Moving up the third `lw` instruction to become the third instruction eliminates both hazards:

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
lw    $t4, 8($t0)
add   $t3, $t1,$t2
sw    $t3, 12($t0)
add   $t5, $t1,$t4
sw    $t5, 16($t0)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

Forwarding yields another insight into the MIPS architecture, in addition to the four mentioned on page 277. Each MIPS instruction writes at most one result and does this in the last stage of the pipeline. Forwarding is harder if there are multiple results to forward per instruction or if there is a need to write a result early on in instruction execution.

Elaboration: The name “forwarding” comes from the idea that the result is passed forward from an earlier instruction to a later instruction. “Bypassing” comes from passing the result around the register file to the desired unit.

Control Hazards

The third type of hazard is called a **control hazard**, arising from the need to make a decision based on the results of one instruction while others are executing.

Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until after the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here is the first of two solutions to control hazards in the laundry room and its computer equivalent.

Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula.

This conservative option certainly works, but it is slow.

ANSWER

control hazard Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

The equivalent decision task in a computer is the branch instruction. Notice that we must begin fetching the instruction following the branch on the very next clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory! Just as with laundry, one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline (see Section 4.8 for details). Even with this extra hardware, the pipeline involving conditional branches would look like Figure 4.31. The `lw` instruction, executed if the branch fails, is stalled one extra 200 ps clock cycle before starting.

EXAMPLE

Performance of “Stall on Branch”

Estimate the impact on the *clock cycles per instruction* (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.

ANSWER

Figure 3.27 in Chapter 3 shows that branches are 17% of the instructions executed in SPECint2006. Since the other instructions run have a CPI of 1, and branches took one extra clock cycle for the stall, then we would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case.

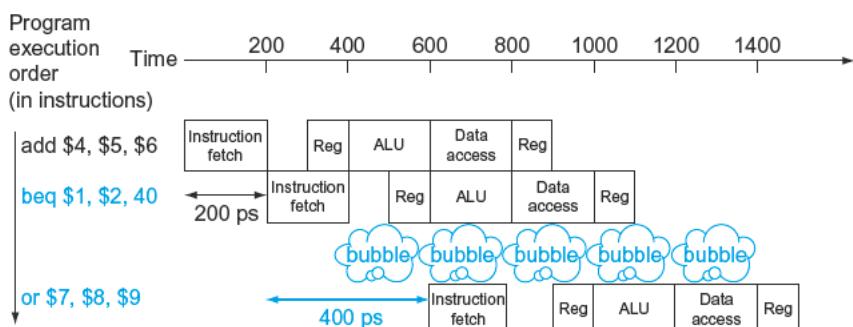


FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.8. The effect on performance, however, is the same as would occur if a bubble were inserted.

If we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we'd see an even larger slowdown if we stall on branches. The cost of this option is too high for most computers to use and motivates a second solution to the control hazard using one of our great ideas from Chapter 1:

Predict: If you're pretty sure you have the right formula to wash uniforms, then just *predict* that it will work and wash the second load while waiting for the first load to dry.

This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Computers do indeed use **prediction** to handle branches. One simple approach is to predict always that branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall. Figure 4.32 shows such an example.



PREDICTION

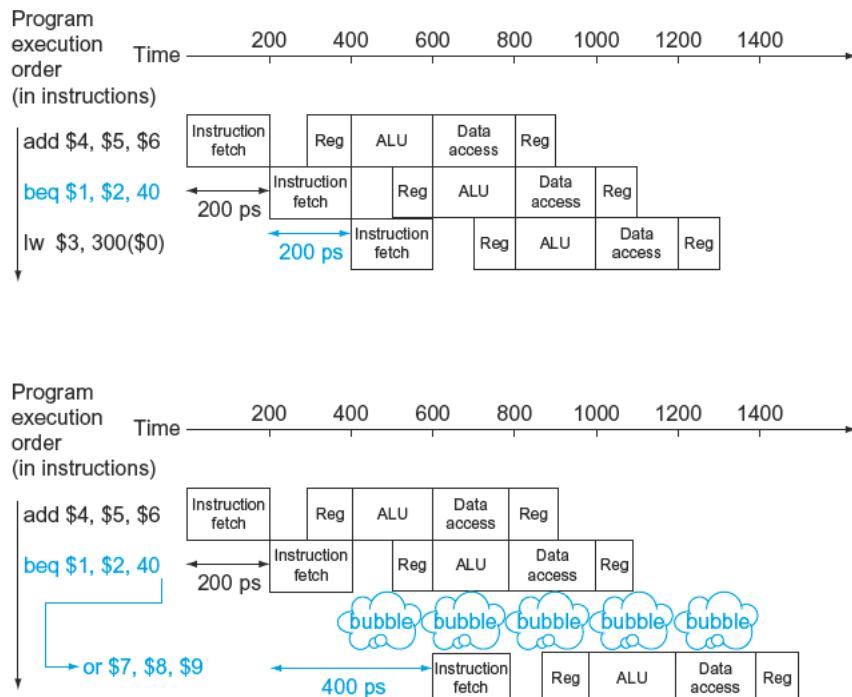


FIGURE 4.32 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.8 will reveal the details.

branch prediction

A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.



PREDICTION

A more sophisticated version of **branch prediction** would have some branches predicted as taken and some as untaken. In our analogy, the dark or home uniforms might take one formula while the light or road uniforms might take another. In the case of programming, at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for branches that jump to an earlier address.

Such rigid approaches to branch prediction rely on stereotypical behavior and don't account for the individuality of a specific branch instruction. *Dynamic* hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next **prediction** depending on the success of recent guesses.

One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future. As we will see later, the amount and type of history kept have become extensive, with the result being that dynamic branch predictors can correctly predict branches with more than 90% accuracy (see Section 4.8). When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address. In our laundry analogy, we must stop taking new loads so that we can restart the load that we incorrectly predicted.

As in the case of all other solutions to control hazards, longer pipelines exacerbate the problem, in this case by raising the cost of misprediction. Solutions to control hazards are described in more detail in Section 4.8.

Elaboration: There is a third approach to the control hazard, called *delayed decision*. In our analogy, whenever you are going to make such a decision about laundry, just place a load of nonfootball clothes in the washer while waiting for football uniforms to dry. As long as you have enough dirty clothes that are not affected by the test, this solution works fine.

Called the *delayed branch* in computers, and mentioned above, this is the solution actually used by the MIPS architecture. The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer. MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that follows this safe instruction. In our example, the add instruction before the branch in Figure 4.31 does not affect the branch and can be moved after the branch to fully hide the branch delay. Since delayed branches are useful when the branches are short, no processor uses a delayed branch of more than one cycle. For longer branch delays, hardware-based branch prediction is usually used.

Pipeline Overview Summary

Pipelining is a technique that exploits **parallelism** among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike programming a multiprocessor, it is fundamentally invisible to the programmer.

In the next few sections of this chapter, we cover the concept of pipelining using the MIPS instruction subset from the single-cycle implementation in Section 4.4 and show a simplified version of its pipeline. We then look at the problems that **pipelining** introduces and the performance attainable under typical situations.

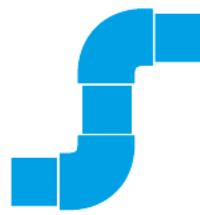
If you wish to focus more on the software and the performance implications of pipelining, you now have sufficient background to skip to Section 4.10. Section 4.10 introduces advanced pipelining concepts, such as superscalar and dynamic scheduling, and Section 4.11 examines the pipelines of recent microprocessors.

Alternatively, if you are interested in understanding how pipelining is implemented and the challenges of dealing with hazards, you can proceed to examine the design of a pipelined datapath and the basic control, explained in Section 4.6. You can then use this understanding to explore the implementation of forwarding and stalls in Section 4.7. You can then read Section 4.8 to learn more about solutions to branch hazards, and then see how exceptions are handled in Section 4.9.

For each code sequence below, state whether it must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.



PARALLELISM



PIPELINING

Check Yourself

Sequence 1	Sequence 2	Sequence 3
<pre>lw \$t0,0(\$t0) add \$t1,\$t0,\$t0</pre>	<pre>add \$t1,\$t0,\$t0 addi \$t2,\$t0,#5 addi \$t4,\$t1,#5</pre>	<pre>addi \$t1,\$t0,#1 addi \$t2,\$t0,#2 addi \$t3,\$t0,#2 addi \$t3,\$t0,#4 addi \$t5,\$t0,#5</pre>

Outside the memory system, the effective operation of the pipeline is usually the most important factor in determining the CPI of the processor and hence its performance. As we will see in Section 4.10, understanding the performance of a modern multiple-issue pipelined processor is complex and requires understanding more than just the issues that arise in a simple pipelined processor. Nonetheless, structural, data, and control hazards remain important in both simple pipelines and more sophisticated ones.

For modern pipelines, structural hazards usually revolve around the floating-point unit, which may not be fully pipelined, while control hazards are usually more of a problem in integer programs, which tend to have higher branch frequencies as well as less predictable branches. Data hazards can be performance bottlenecks

Understanding Program Performance



The BIG Picture

latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.



There is less in this than meets the eye.

Tallulah Bankhead, remark to Alexander Woollcott, 1922

in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower branch frequency and more regular memory access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular memory access, involving more use of pointers. As we will see in Section 4.10, there are more ambitious compiler and hardware techniques for reducing data dependences through scheduling.

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction, also called the **latency**. For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in Chapter 1, pipelining improves instruction *throughput* rather than individual instruction *execution time* or *latency*.

Instruction sets can either simplify or make life harder for pipeline designers, who must already cope with structural, control, and data hazards. Branch **prediction** and forwarding help make a computer fast while still getting the right answers.

4.6

Pipelined Datapath and Control

Figure 4.33 shows the single-cycle datapath from Section 4.4 with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

In Figure 4.33, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the

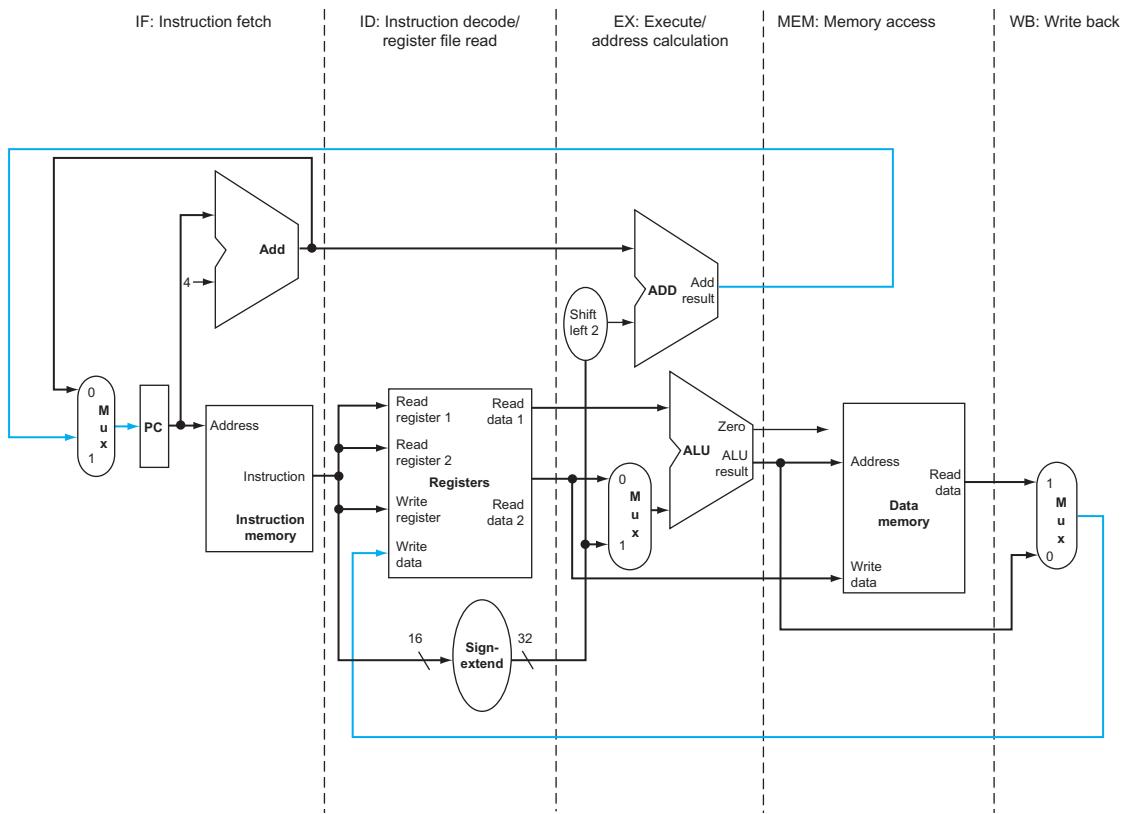


FIGURE 4.33 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

five stages as they complete execution. Returning to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backward.

There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline. Note that

the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a timeline to show their relationship. [Figure 4.34](#) shows the execution of the instructions in [Figure 4.27](#) by displaying their private datapaths on a common timeline. We use a stylized version of the datapath in [Figure 4.33](#) to show the relationships in [Figure 4.34](#).

[Figure 4.34](#) seems to suggest that three instructions need three datapaths. Instead, we add registers to hold data so that portions of a single datapath can be shared during instruction execution.

For example, as [Figure 4.34](#) shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages. To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages in [Figure 4.33](#). Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

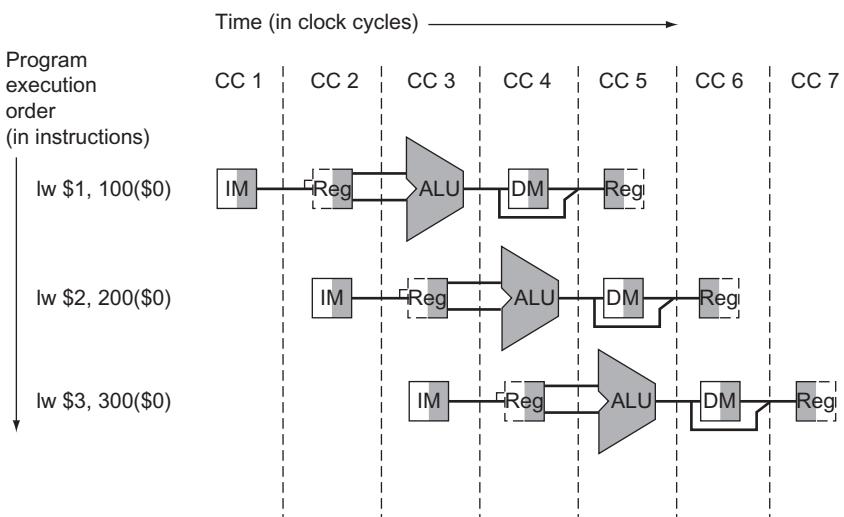


FIGURE 4.34 Instructions being executed using the single-cycle datapath in Figure 4.33, assuming pipelined execution. Similar to Figures 4.28 through 4.30, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.33. IM represents the instruction memory and the PC in the instruction fetch stage, Reg stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

[Figure 4.35](#) shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor—the register file, memory, or the PC—so a separate pipeline register is redundant to the state that is updated. For example, a load instruction will place its result in 1 of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.

Of course, every instruction updates the PC, whether by incrementing it or by setting it to a branch destination address. The PC can be thought of as a pipeline register: one that feeds the IF stage of the pipeline. Unlike the shaded pipeline registers in [Figure 4.35](#), however, the PC is part of the visible architectural state; its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded. In the laundry analogy, you could think of the PC as corresponding to the basket that holds the load of dirty clothes before the wash step.

To show how the pipelining works, throughout this chapter we show sequences of figures to demonstrate operation over time. These extra pages would seem to require much more time for you to understand. Fear not; the sequences take much

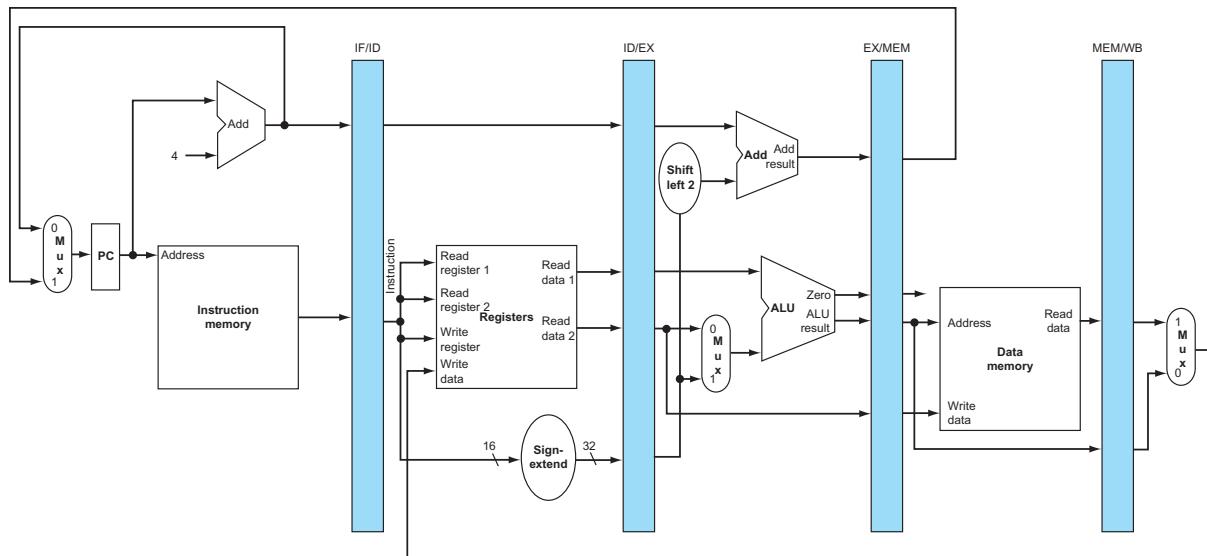


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled IF/ID because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

less time than it might appear, because you can compare them to see what changes occur in each clock cycle. Section 4.7 describes what happens when there are data hazards between pipelined instructions; ignore them for now.

Figures 4.36 through 4.38, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. We show a load first because it is active in all five stages. As in Figures 4.28 through 4.30, we highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*.

We show the instruction abbreviation $l\ w$ with the name of the pipe stage that is active in each figure. The five stages are the following:

1. *Instruction fetch*: The top portion of Figure 4.36 shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as `beq`. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.
2. *Instruction decode and register file read*: The bottom portion of Figure 4.36 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.
3. *Execute or address calculation*: Figure 4.37 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.
4. *Memory access*: The top portion of Figure 4.38 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.
5. *Write-back*: The bottom portion of Figure 4.38 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register. Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are the five pipe stages of the store instruction:

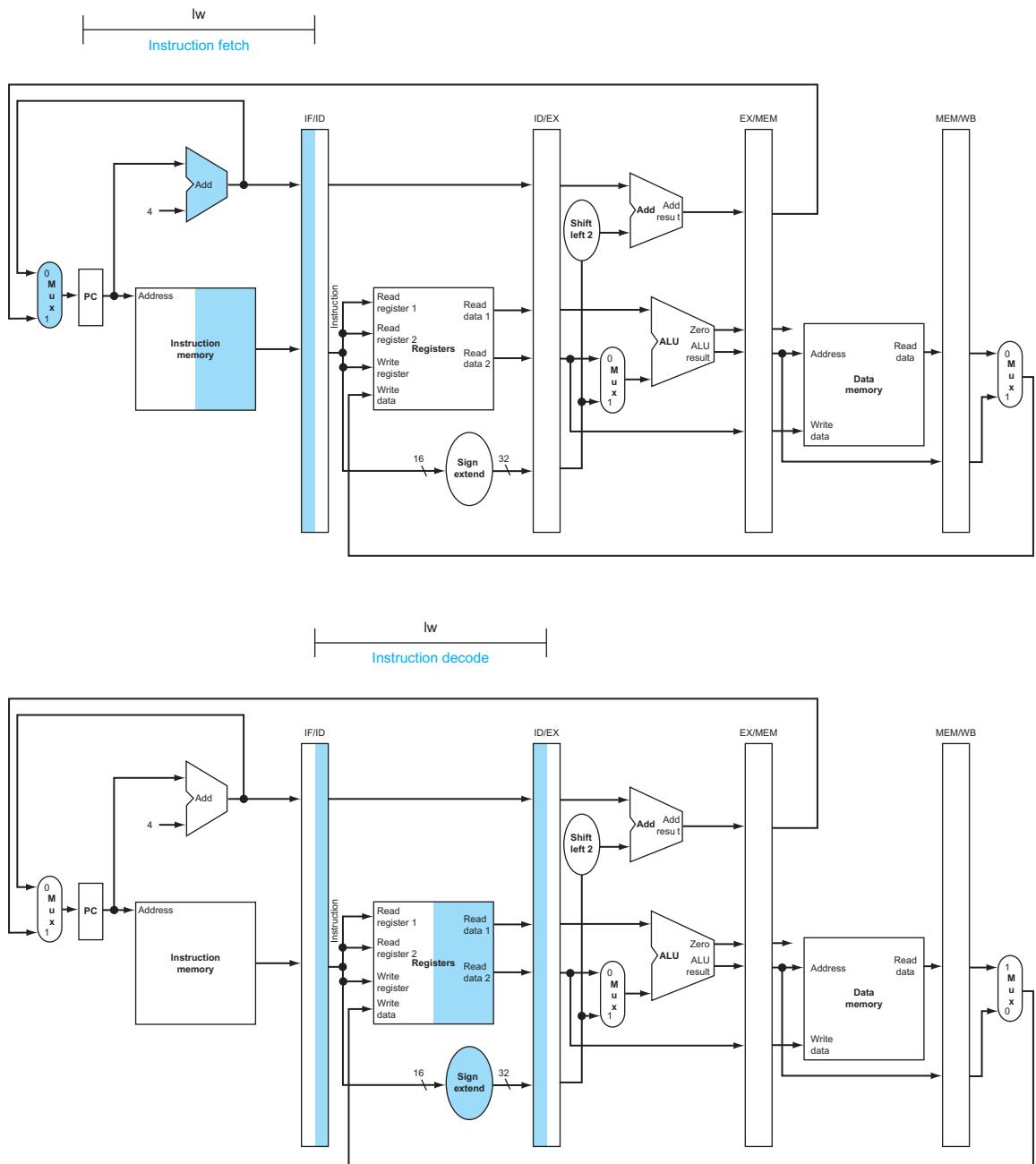


FIGURE 4.36 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.35 highlighted. The highlighting convention is the same as that used in Figure 4.28. As in Section 4.2, there is no confusion when reading and writing registers, because the contents change only on the clock edge. Although the load needs only the top register in stage 2, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register. We don't need all three operands, but it simplifies control to keep all three.

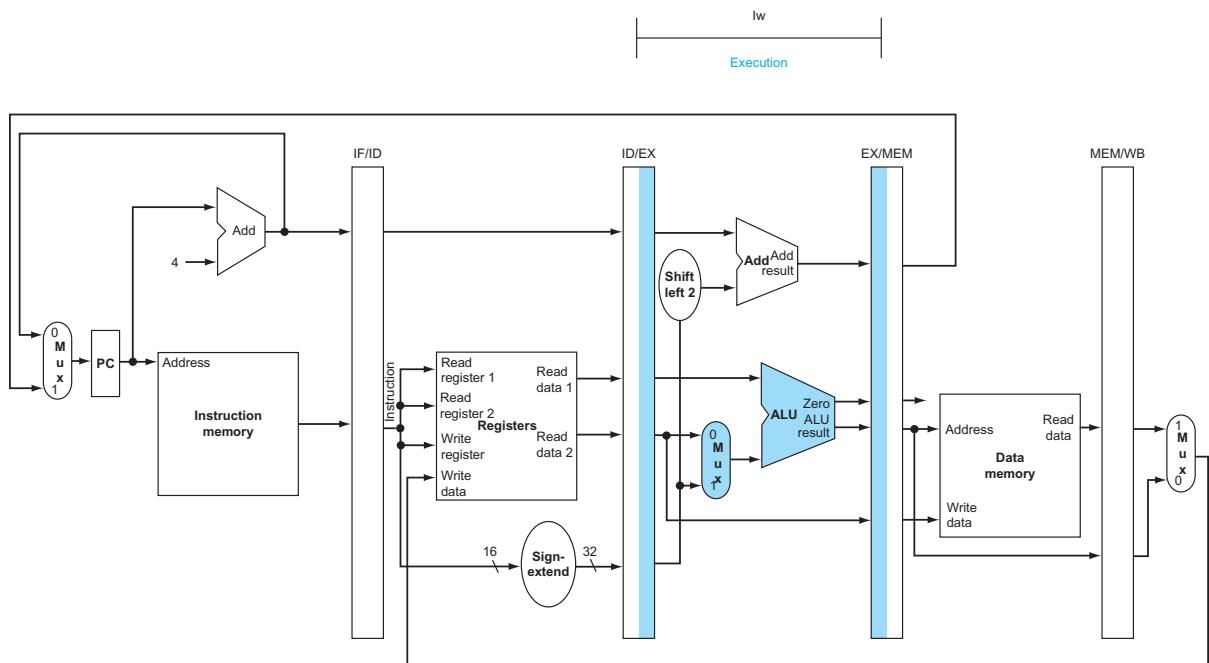


FIGURE 4.37 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.35 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

1. *Instruction fetch:* The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of Figure 4.36 works for store as well as load.
2. *Instruction decode and register file read:* The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 4.36 for load instructions also shows the operations of the second stage for stores. These first two stages are executed by all instructions, since it is too early to know the type of the instruction.
3. *Execute and address calculation:* Figure 4.39 shows the third step; the effective address is placed in the EX/MEM pipeline register.
4. *Memory access:* The top portion of Figure 4.40 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

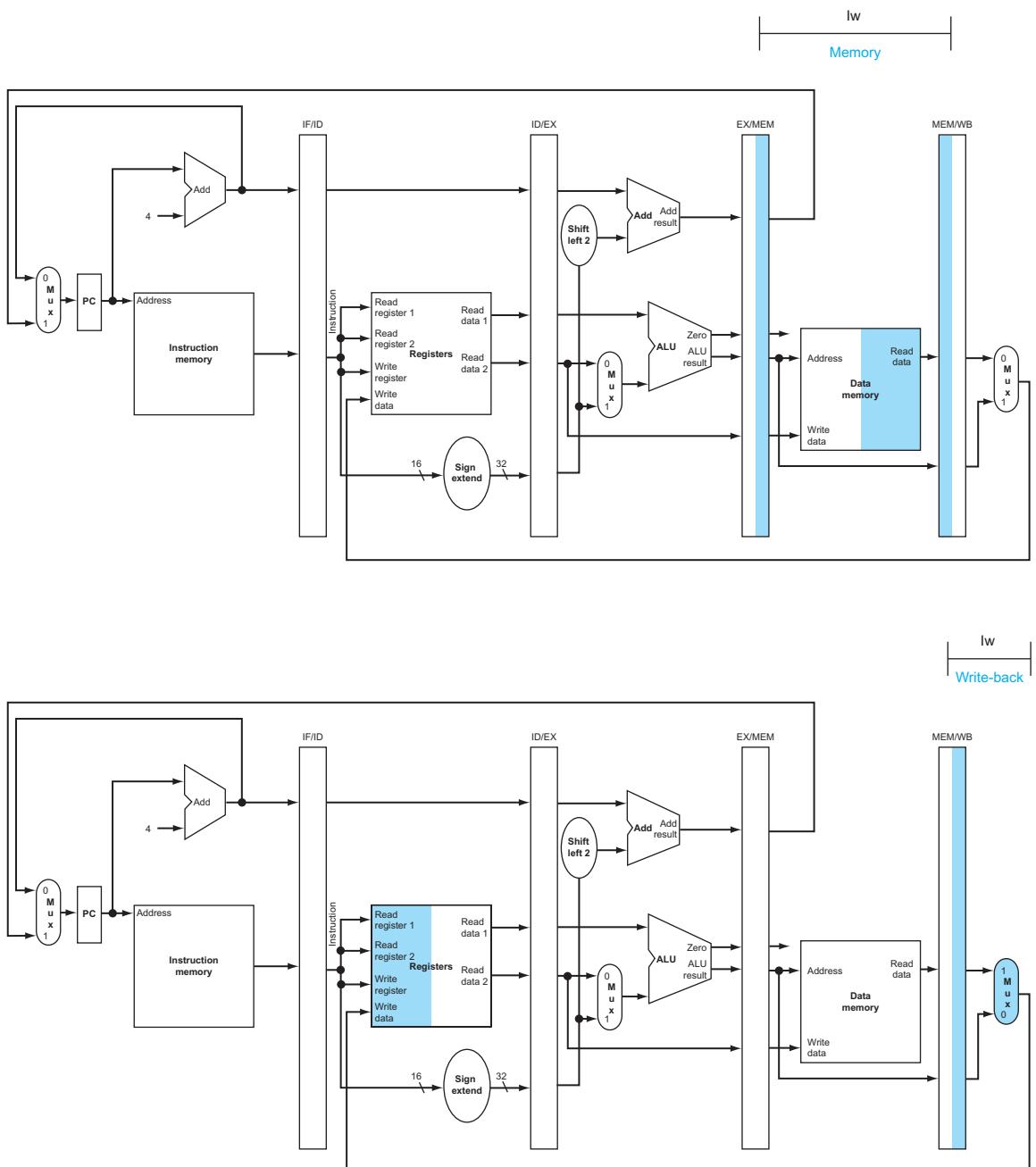


FIGURE 4.38 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.35 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register. Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath. Note: there is a bug in this design that is repaired in Figure 4.41.

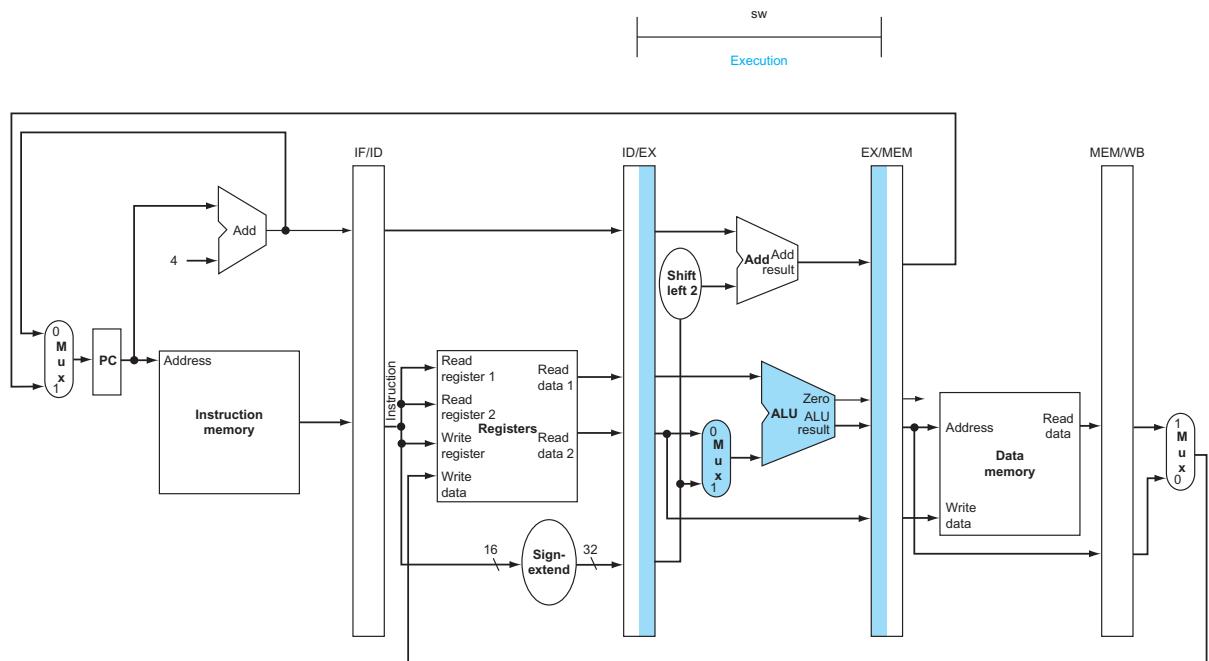


FIGURE 4.39 EX: The third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 4.37, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

5. *Write-back:* The bottom portion of Figure 4.40 shows the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

The store instruction again illustrates that to pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise, the information is lost when the next instruction enters that pipeline stage. For the store instruction we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.

Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a *single* pipeline stage. Otherwise, we would have a *structural hazard* (see page 277). Hence these components, and their control, can be associated with a single pipeline stage.

Now we can uncover a bug in the design of the load instruction. Did you see it? Which register is changed in the final stage of the load? More specifically, which

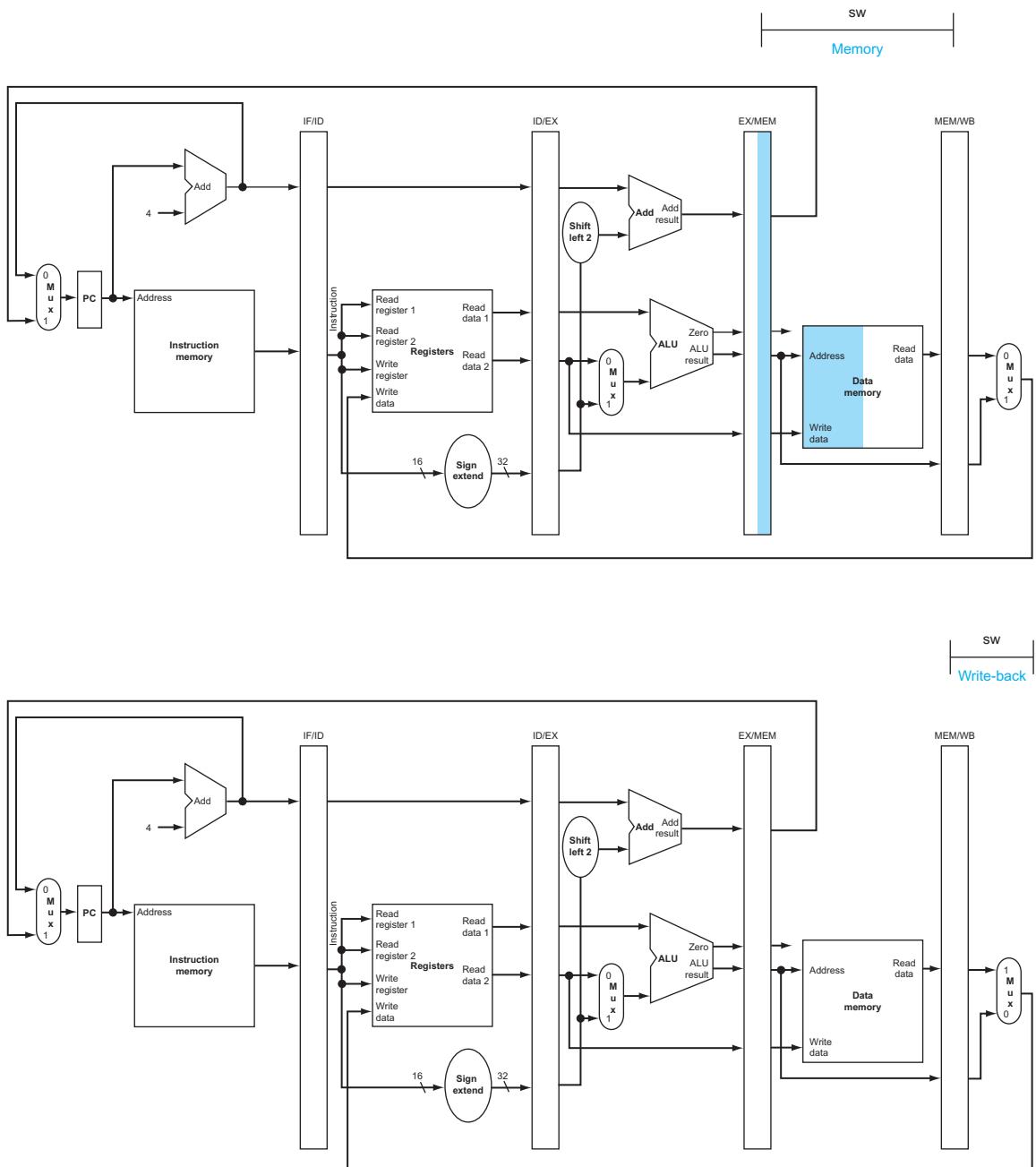


FIGURE 4.40 MEM and WB: The fourth and fifth pipe stages of a store instruction. In the fourth stage, the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably after the load instruction!

Hence, we need to preserve the destination register number in the load instruction. Just as store passed the register *contents* from the ID/EX to the EX/MEM pipeline registers for use in the MEM stage, load must pass the *register number* from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage. Another way to think about the passing of the register number is that to share the pipelined datapath, we need to preserve the instruction read during the IF stage, so each pipeline register contains a portion of the instruction needed for that stage and later stages.

[Figure 4.41](#) shows the correct version of the datapath, passing the write register number first to the ID/EX register, then to the EX/MEM register, and finally to the MEM/WB register. The register number is used during the WB stage to specify the register to be written. [Figure 4.42](#) is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load word instruction in [Figures 4.36 through 4.38](#). See Section 4.8 for an explanation of how to make the branch instruction work as expected.

Graphically Representing Pipelines

Pipelining can be difficult to understand, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are

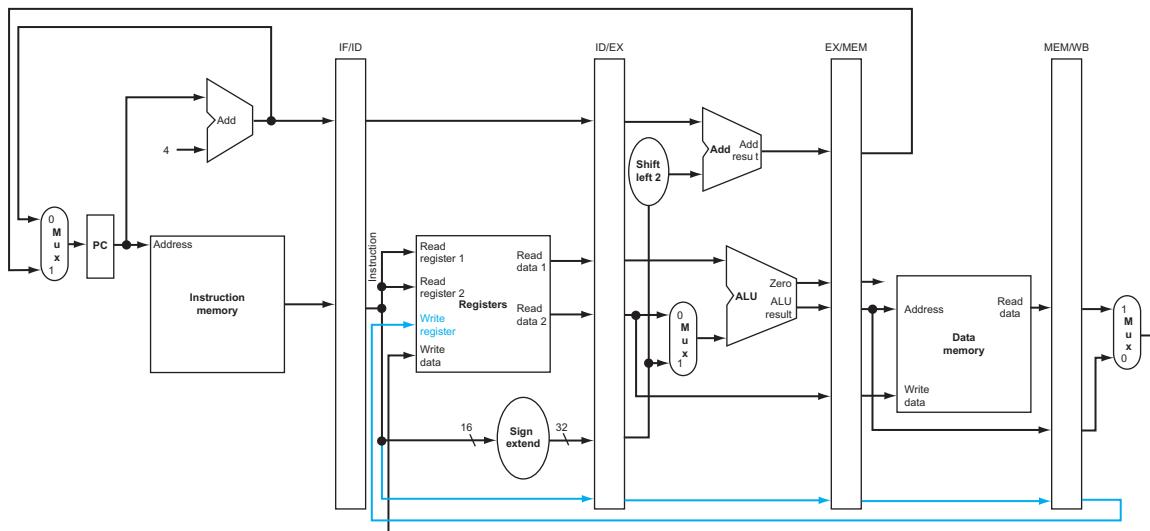


FIGURE 4.41 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.

two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as [Figure 4.34](#) on page 288, and *single-clock-cycle pipeline diagrams*, such as [Figures 4.36 through 4.40](#). The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
lw      $10, 20($1)
sub    $11, $2, $3
add    $12, $3, $4
lw      $13, 24($1)
add    $14, $5, $6
```

[Figure 4.43](#) shows the multiple-clock-cycle pipeline diagram for these instructions. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom of the page, similar to the laundry pipeline in [Figure 4.25](#). A representation of the pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles. These stylized datapaths represent the five stages of our pipeline graphically, but a rectangle naming each pipe stage works just as well. [Figure 4.44](#) shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that [Figure 4.43](#) shows the physical resources used at each stage, while [Figure 4.44](#) uses the name of each stage.

Single-clock-cycle pipeline diagrams show the state of the entire datapath during a single clock cycle, and usually all five instructions in the pipeline are identified by labels above their respective pipeline stages. We use this type of figure to show the details of what is happening within the pipeline during each clock cycle; typically,

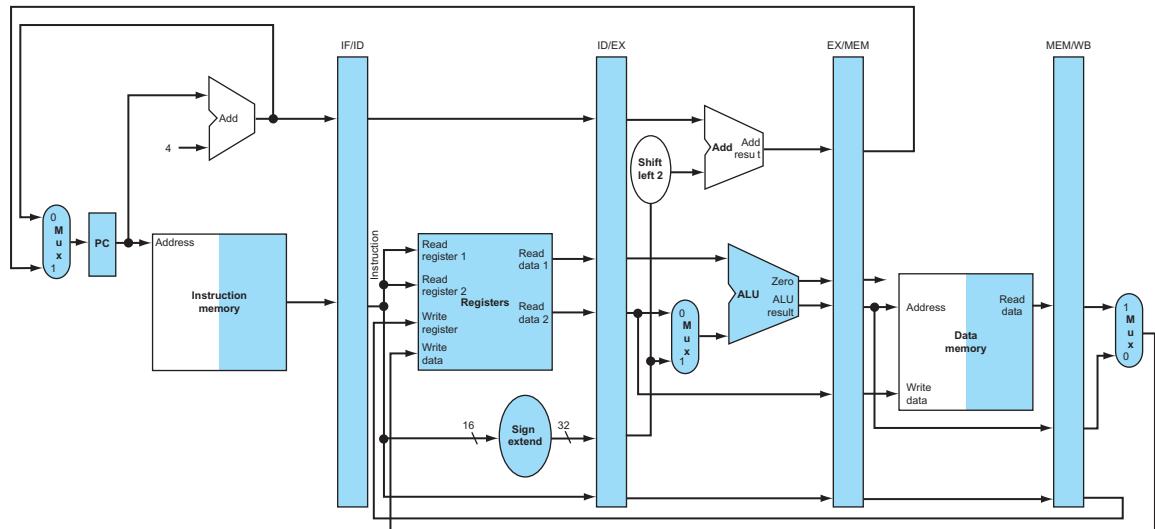


FIGURE 4.42 The portion of the datapath in [Figure 4.41](#) that is used in all five stages of a load instruction.

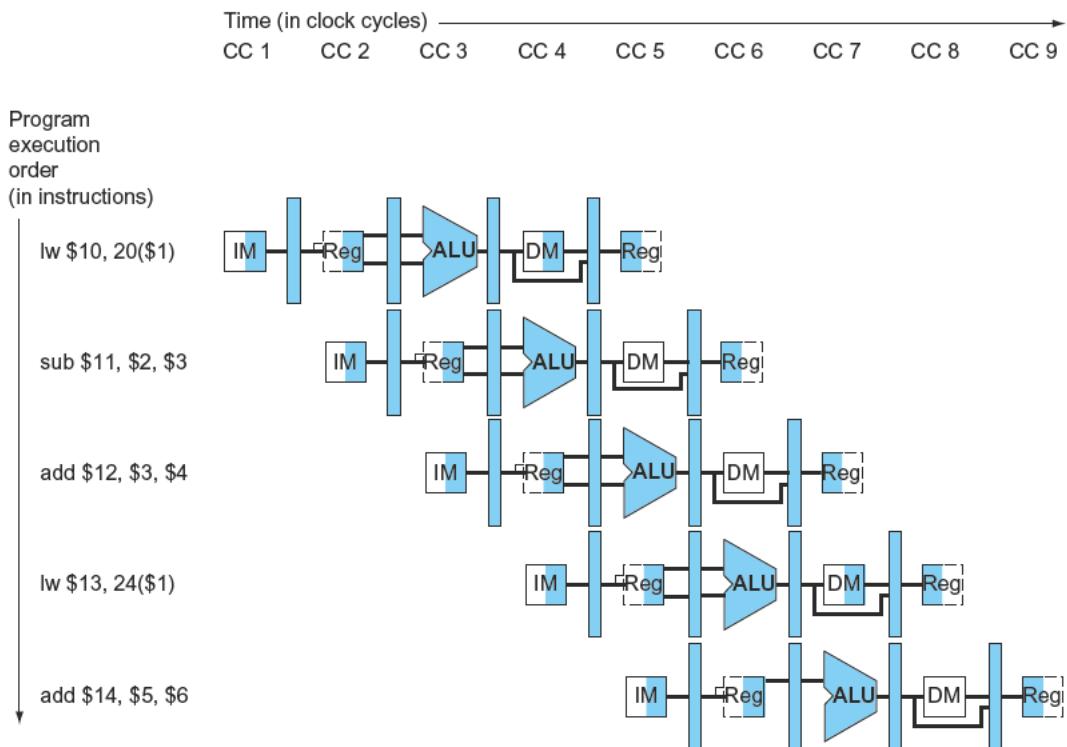


FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five Instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.28, here we show the pipeline registers between each stage. Figure 4.44 shows the traditional way to draw this diagram.

the drawings appear in groups to show pipeline operation over a sequence of clock cycles. We use multiple-clock-cycle diagrams to give overviews of pipelining situations. (Section 4.13 gives more illustrations of single-clock diagrams if you would like to see more details about Figure 4.43.) A single-clock-cycle diagram represents a vertical slice through a set of multiple-clock-cycle diagrams, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle. For example, Figure 4.45 shows the single-clock-cycle diagram corresponding to clock cycle 5 of Figures 4.43 and 4.44. Obviously, the single-clock-cycle diagrams have more detail and take significantly more space to show the same number of clock cycles. The exercises ask you to create such diagrams for other code sequences.

Check Yourself

A group of students were debating the efficiency of the five-stage pipeline when one student pointed out that not all instructions are active in every stage of the pipeline. After deciding to ignore the effects of hazards, they made the following four statements. Which ones are correct?

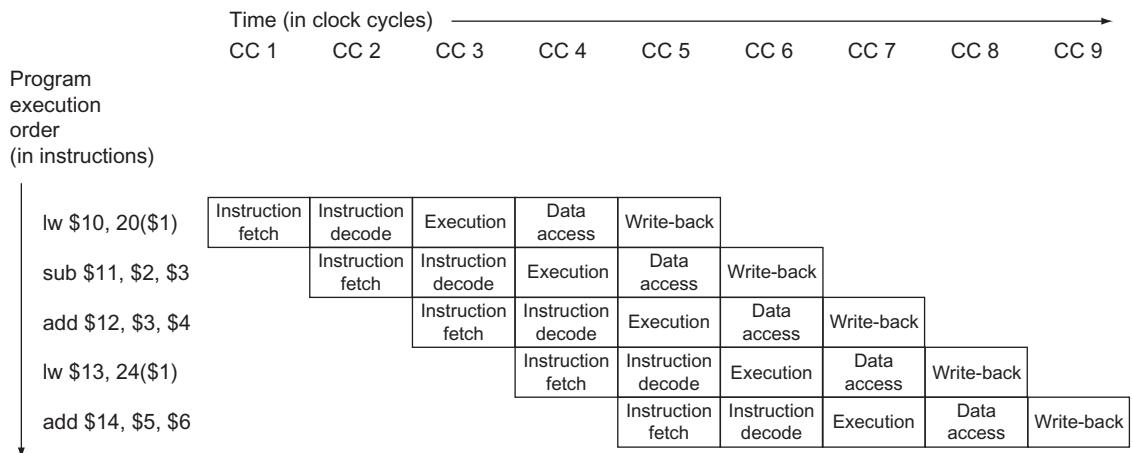


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43.

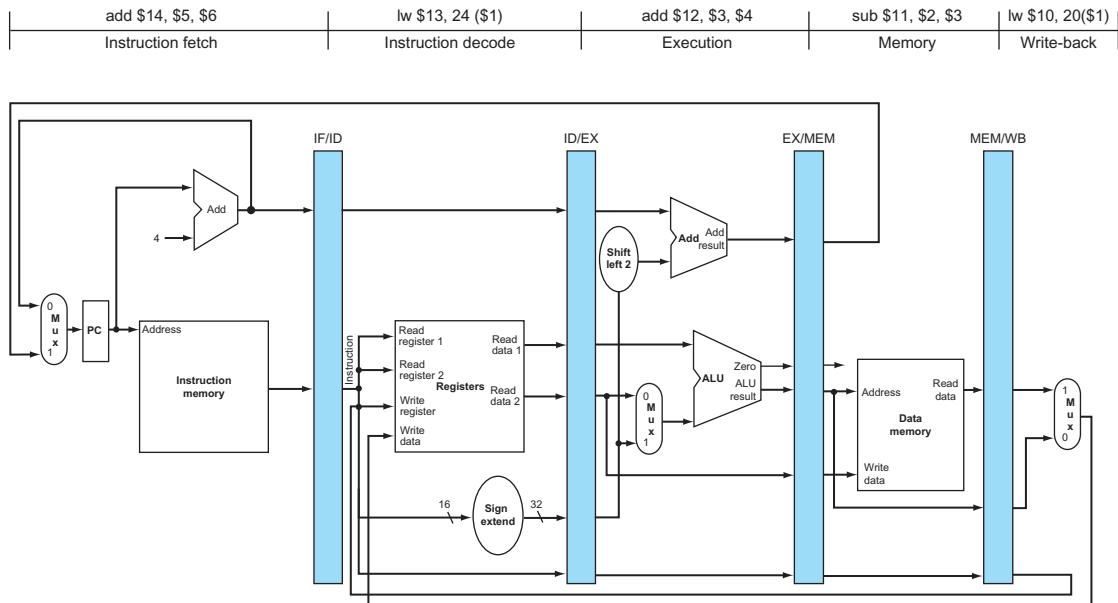


FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44.
As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

- Allowing jumps, branches, and ALU instructions to take fewer stages than the five required by the load instruction will increase pipeline performance under all circumstances.

2. Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle; the number of pipe stages per instruction affects latency, not throughput.
3. You cannot make ALU instructions take fewer cycles because of the write-back of the result, but branches and jumps can take fewer cycles, so there is some opportunity for improvement.
4. Instead of trying to make instructions take fewer cycles, we should explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter. This could improve performance.

Pipelined Control

In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.

James Thornton, *Design of a Computer: The Control Data 6600*, 1970

Just as we added control to the single-cycle datapath in Section 4.3, we now add control to the pipelined datapath. We start with a simple design that views the problem through rose-colored glasses.

The first step is to label the control lines on the existing datapath. [Figure 4.46](#) shows those lines. We borrow as much as we can from the control for the simple datapath in [Figure 4.17](#). In particular, we use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines. These functions are defined in [Figures 4.12, 4.16, and 4.18](#). We reproduce the key information in [Figures 4.47 through 4.49](#) on a single page to make the following discussion easier to follow.

As was the case for the single-cycle implementation, we assume that the PC is written on each clock cycle, so there is no separate write signal for the PC. By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. *Instruction fetch:* The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
2. *Instruction decode/register file read:* As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
3. *Execution/address calculation:* The signals to be set are RegDst, ALUOp, and ALUSrc (see [Figures 4.47 and 4.48](#)). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

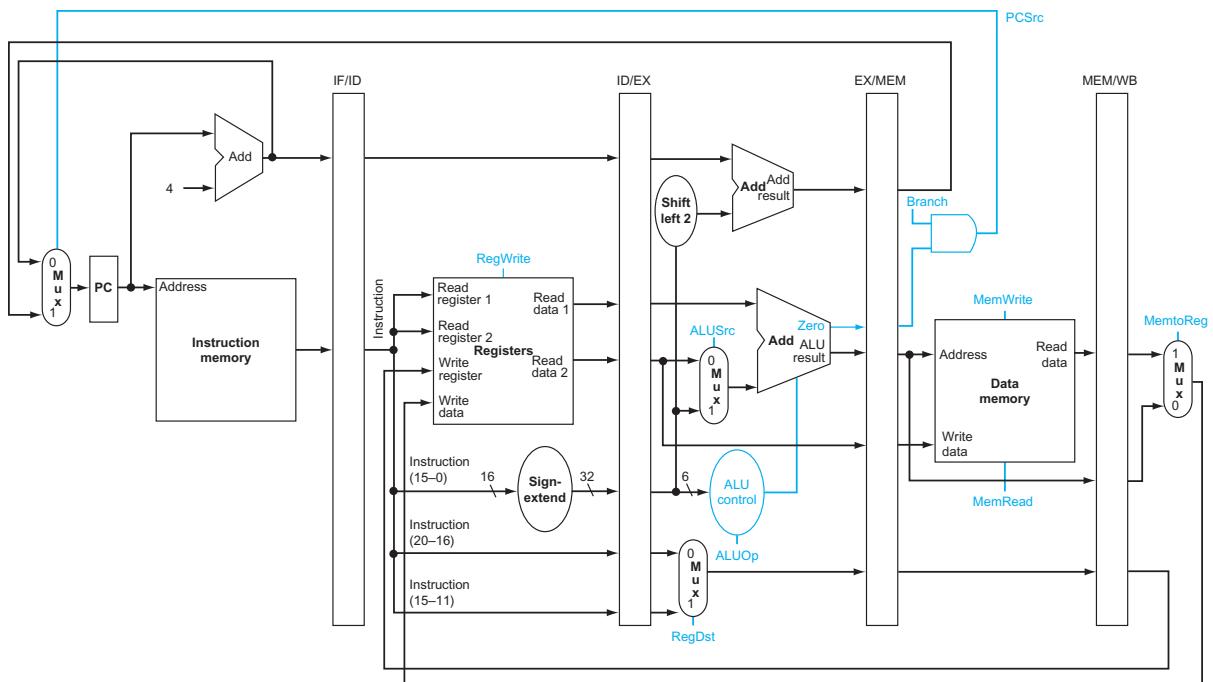


FIGURE 4.46 The pipelined datapath of Figure 4.41 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 4.47 A copy of Figure 4.12. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the data memory.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.48 A copy of Figure 4.16. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.47. When a 1-bit control to a 2-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 4.46. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURE 4.49 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.

4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Figure 4.48 selects the next sequential address unless control asserts Branch and the ALU result was 0.
5. *Write-back:* The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and Reg-Write, which writes the chosen value.

Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values. Figure 4.49 has the same values as in Section 4.4, but now the nine control lines are grouped by pipeline stage.

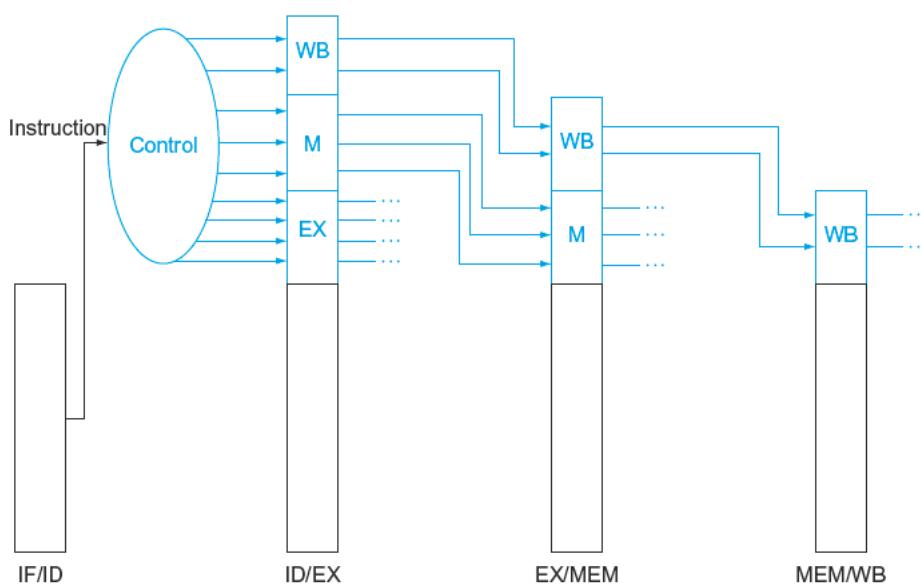


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

Since the control lines start with the EX stage, we can create the control information during instruction decode. Figure 4.50 above shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline in Figure 4.41. Figure 4.51 shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage. (Section 4.13 gives more examples of MIPS code executing on pipelined hardware using single-clock diagrams, if you would like to see more details.)

4.7

Data Hazards: Forwarding versus Stalling

The examples in the previous section show the power of pipelined execution and how the hardware performs the task. It's now time to take off the rose-colored glasses and look at what happens with real programs. The instructions in Figures 4.43 through 4.45 were independent; none of them used the results calculated by any of the others. Yet in Section 4.5, we saw that data hazards are obstacles to pipelined execution.

*What do you mean,
why's it got to be built?
It's a bypass. You've got
to build bypasses.*

Douglas Adams, *The Hitchhiker's Guide to the Galaxy*, 1979

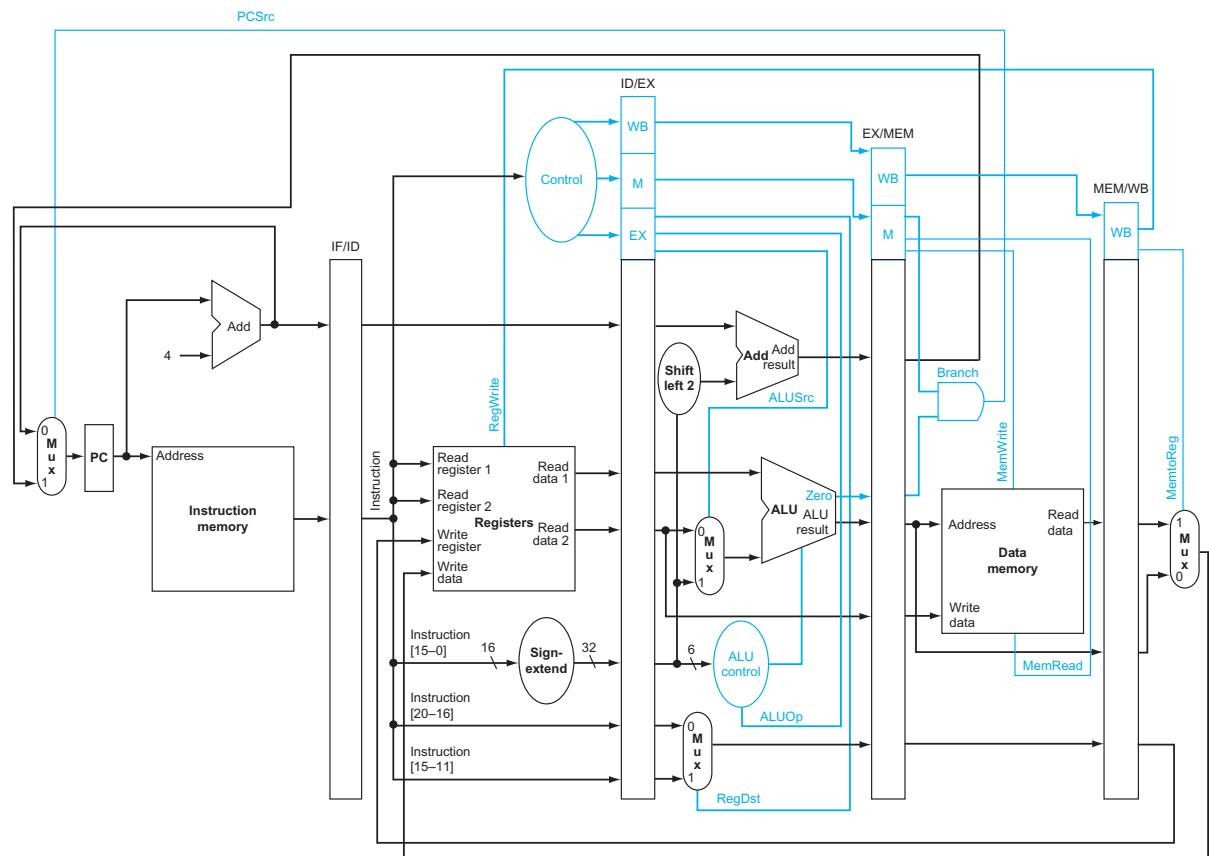


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Let's look at a sequence with many dependences, shown in color:

sub	\$2, \$1,\$3	# Register \$2 written by sub
and	\$12,\$2,\$5	# 1st operand(\$2) depends on sub
or	\$13,\$6,\$2	# 2nd operand(\$2) depends on sub
add	\$14,\$2,\$2	# 1st(\$2) & 2nd(\$2) depend on sub
sw	\$15,100(\$2)	# Base (\$2) depends on sub

The last four instructions are all dependent on the result in register \$2 of the first instruction. If register \$2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register \$2.

How would this sequence perform with our pipeline? Figure 4.52 illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of Figure 4.52 shows the value of register \$2, which changes during the middle of clock cycle 5, when the sub instruction writes its result.

The last potential hazard can be resolved by the design of the register file hardware: What happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, we have no data hazard in this case.

Figure 4.52 shows that the values read for register \$2 would *not* be the result of the sub instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of -20 are add and sw; the AND and

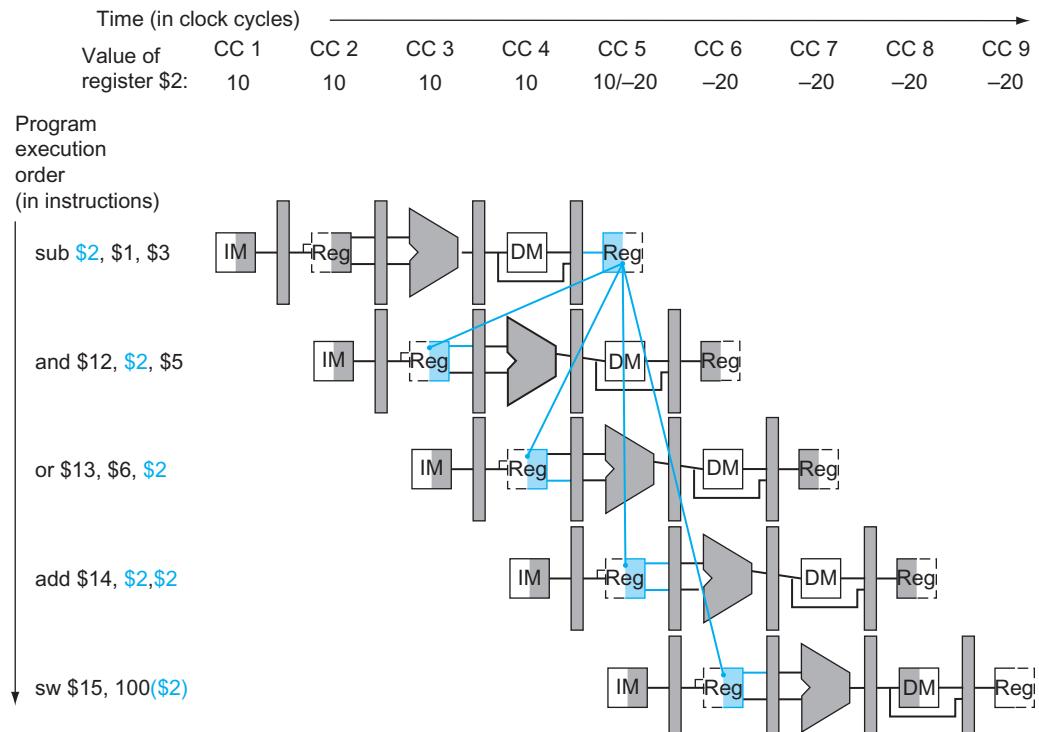


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and “CC 1” at the top of the figure means clock cycle 1. The first instruction writes into \$2, and all the following instructions read \$2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

OR instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backward in time.

As mentioned in Section 4.5, the desired result is available at the end of the EX stage or clock cycle 3. When is the data actually needed by the AND and OR instructions? At the beginning of the EX stage, or clock cycles 4 and 5, respectively. Thus, we can execute this segment without stalls if we simply *forward* the data as soon as it is available to any units that need it before it is available to read from the register file.

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.

A notation that names the fields of the pipeline registers allows for a more precise notation of dependences. For example, “ID/EX.RegisterRs” refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name of the field in that register. Using this notation, the two pairs of hazard conditions are

- 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
- 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
- 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

The first hazard in the sequence on page 304 is on register \$2, between the result of `sub $2, $1, $3` and the first read operand of `and $12, $2, $5`. This hazard can be detected when the `and` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

$$\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs} = \$2$$

EXAMPLE

Dependence Detection

Classify the dependences in this sequence from page 304:

```

sub $2,    $1, $3  # Register $2 set by sub
and $12,   $2, $5  # 1st operand($2) set by sub
or  $13,   $6, $2   # 2nd operand($2) set by sub
add $14,   $2, $2   # 1st($2) & 2nd($2) set by sub
sw   $15, 100($2)  # Index($2) set by sub

```

As mentioned above, the sub - and is a type 1a hazard. The remaining hazards are as follows:

ANSWER

- The sub - or is a type 2b hazard:

$$\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt} = \$2$$

- The two dependences on sub - add are not hazards because the register file supplies the proper data during the ID stage of add.
- There is no data hazard between sub and sw because sw reads \$2 the clock cycle *after* sub writes \$2.

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it shouldn't. One solution is simply to check to see if the RegWrite signal will be active: examining the WB control field of the pipeline register during the EX and MEM stages determines whether RegWrite is asserted. Recall that MIPS requires that every use of \$0 as an operand must yield an operand value of 0. In the event that an instruction in the pipeline has \$0 as its destination (for example, sll \$0, \$1, 2), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for \$0 frees the assembly programmer and the compiler of any requirement to avoid using \$0 as a destination. The conditions above thus work properly as long we add EX/MEM. RegisterRd $\neq 0$ to the first hazard condition and MEM/WB.RegisterRd $\neq 0$ to the second.

Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

Figure 4.53 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in Figure 4.52. The change is that the dependence begins from a *pipeline* register, rather than waiting for the WB stage to write the register file. Thus, the required data exists in time for later instructions, with the pipeline registers holding the data to be forwarded.

If we can take the inputs to the ALU from *any* pipeline register rather than just ID/EX, then we can forward the proper data. By adding multiplexors to the input of the ALU, and with the proper controls, we can run the pipeline at full speed in the presence of these data dependences.

For now, we will assume the only instructions we need to forward are the four R-format instructions: add, sub, AND, and OR. Figure 4.54 shows a close-up of the ALU and pipeline register before and after adding forwarding. Figure 4.55 shows the values of the control lines for the ALU multiplexors that select either the register file values or one of the forwarded values.

This forwarding control will be in the EX stage, because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. We already have the rt field (bits 20–16). Before forwarding, the ID/EX register had no need to include space to hold the rs field. Hence, rs (bits 25–21) is added to ID/EX.

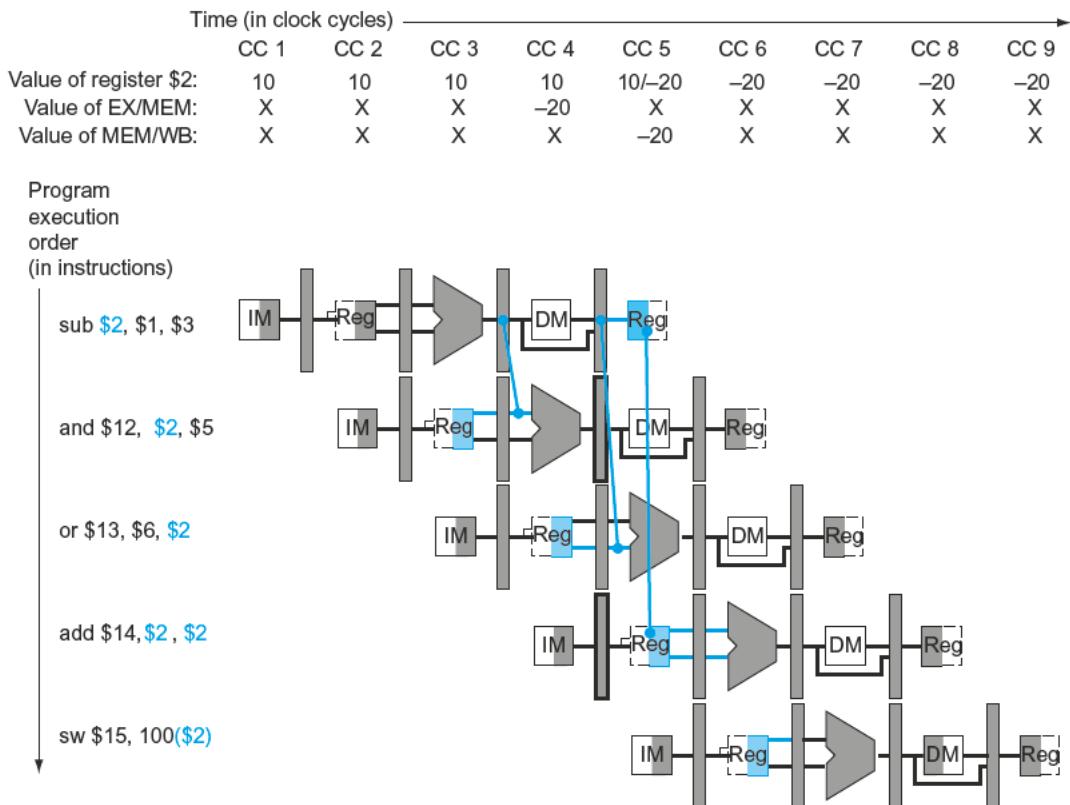


FIGURE 4.53 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the AND instruction and OR instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file “forwarding”—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register \$2 having the value 10 at the beginning and -20 at the end of the clock cycle. As in the rest of this section, we handle all forwarding except for the value to be stored by a store instruction.

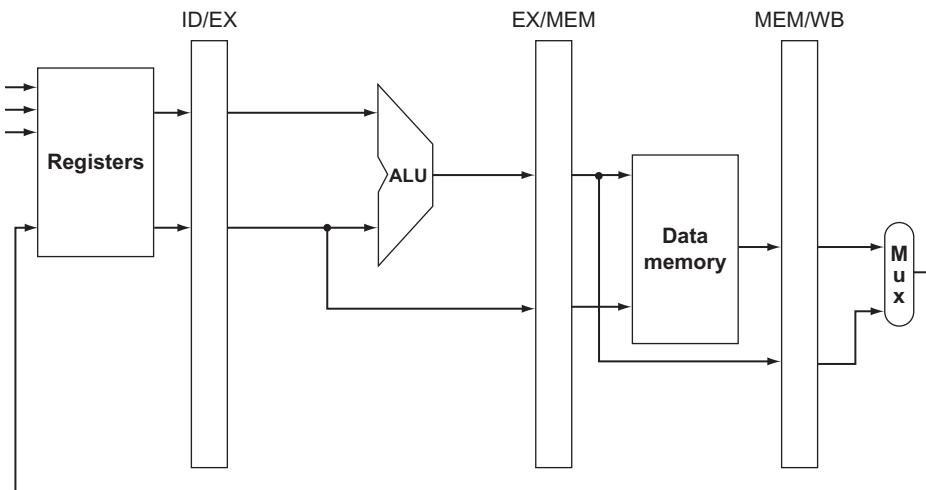
Let's now write both the conditions for detecting hazards and the control signals to resolve them:

1. *EX hazard:*

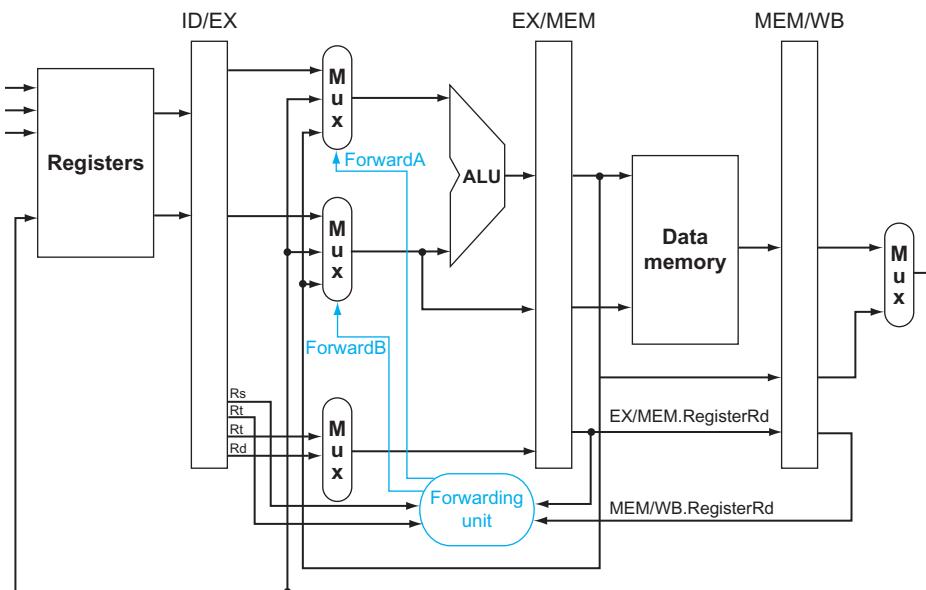
```

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
  
```



a. No forwarding



b. With forwarding

FIGURE 4.54 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware. Note that the ID/EX.RegisterRt field is shown twice, once to connect to the Mux and once to the forwarding unit, but it is a single signal. As in the earlier discussion, this ignores forwarding of a store value to a store instruction. Also note that this mechanism works for `slt` instructions as well.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.55 The control values for the forwarding multiplexors in Figure 4.54. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field).

This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

2. MEM hazard:

```

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

```

As mentioned above, there is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage. For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```

add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
. . .

```

In this case, the result is forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus, the control for the MEM hazard would be (with the additions highlighted):

```

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
       and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
       and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

```

[Figure 4.56](#) shows the hardware necessary to support forwarding for operations that use results during the EX stage. Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction (which comes from the Rd field of the instruction) or a load (which comes from the Rt field).

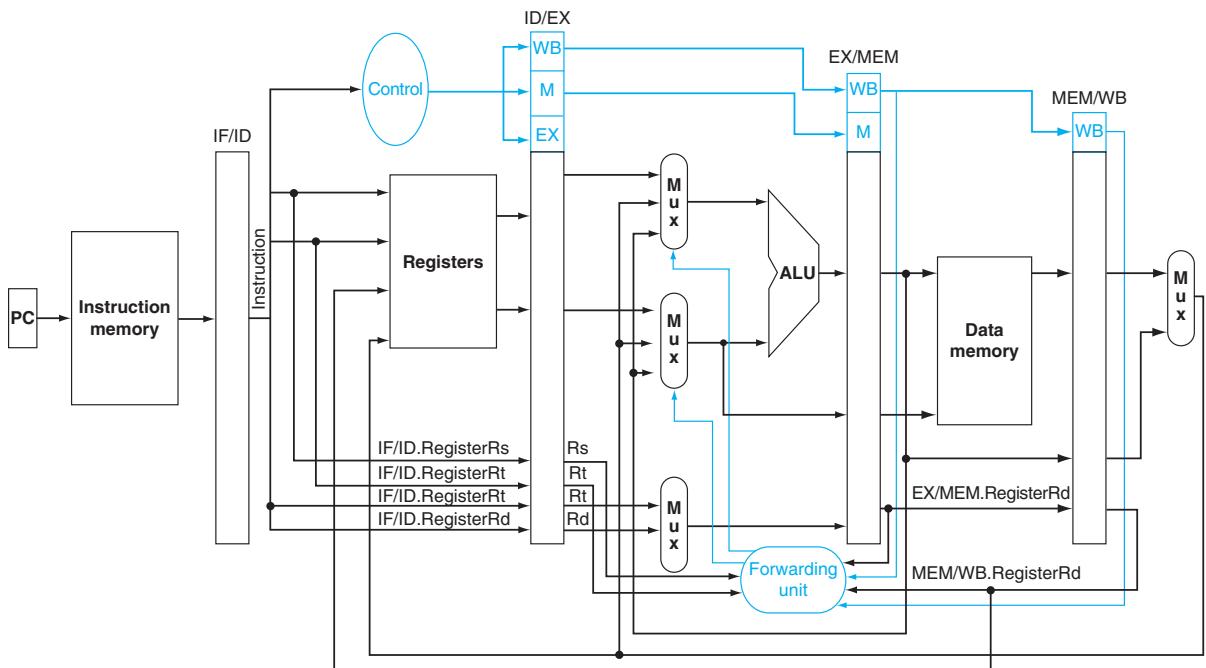


FIGURE 4.56 The datapath modified to resolve hazards via forwarding. Compared with the datapath in [Figure 4.51](#), the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

 **Section 4.13** shows two pieces of MIPS code with hazards that cause forwarding, if you would like to see more illustrated examples using single-cycle pipeline drawings.

Elaboration: Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. However, consider loads immediately followed by stores, useful when performing memory-to-memory copies in the MIPS architecture. Since copies are frequent, we need to add more forwarding hardware to make them run faster. If we were to redraw [Figure 4.53](#), replacing the sub and AND instructions with lw and sw, we would see that it is possible to avoid a stall, since the data exists in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option. We leave this modification as an exercise to the reader.

In addition, the signed-immediate input to the ALU, needed by loads and stores, is missing from the datapath in [Figure 4.56](#). Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register

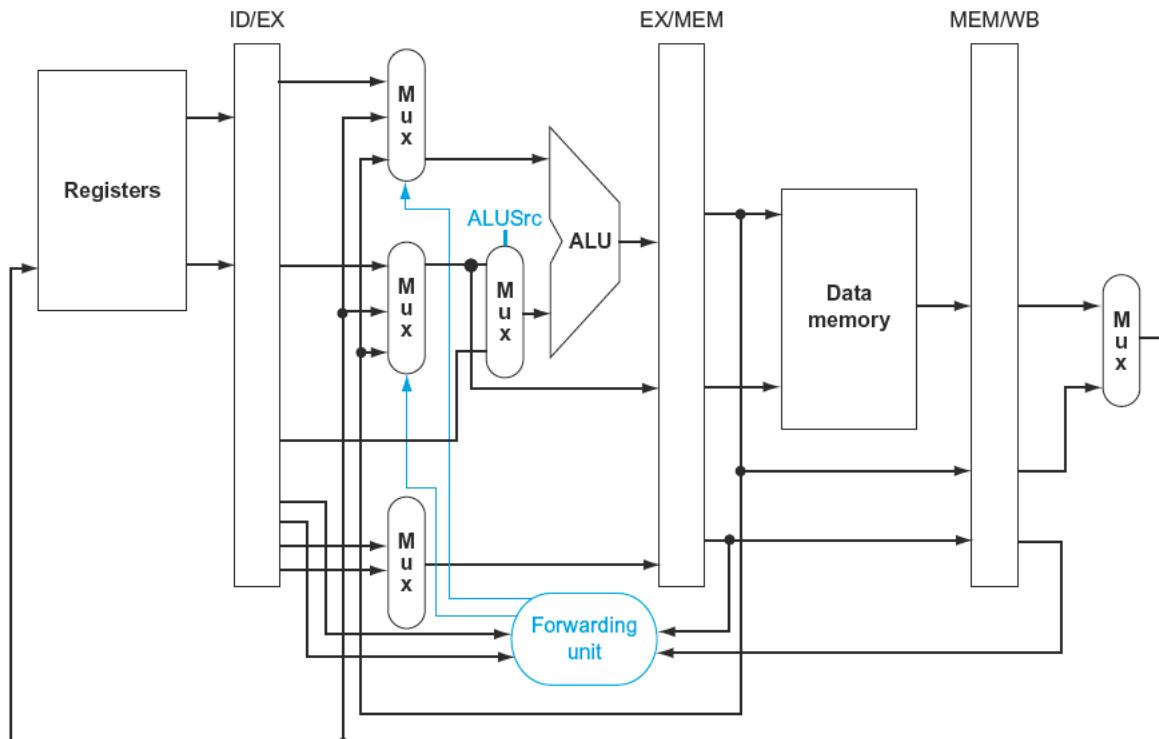


FIGURE 4.57 A close-up of the datapath In [Figure 4.54](#) shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU Input.

input to the ALU, the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate. Figure 4.57 shows this addition.

Data Hazards and Stalls

As we said in Section 4.5, one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. Figure 4.58 illustrates the problem. The data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

Hence, in addition to a forwarding unit, we need a *hazard detection unit*. It operates during the ID stage so that it can insert the stall between the load and its

If at first you don't succeed, redefine success.

Anonymous

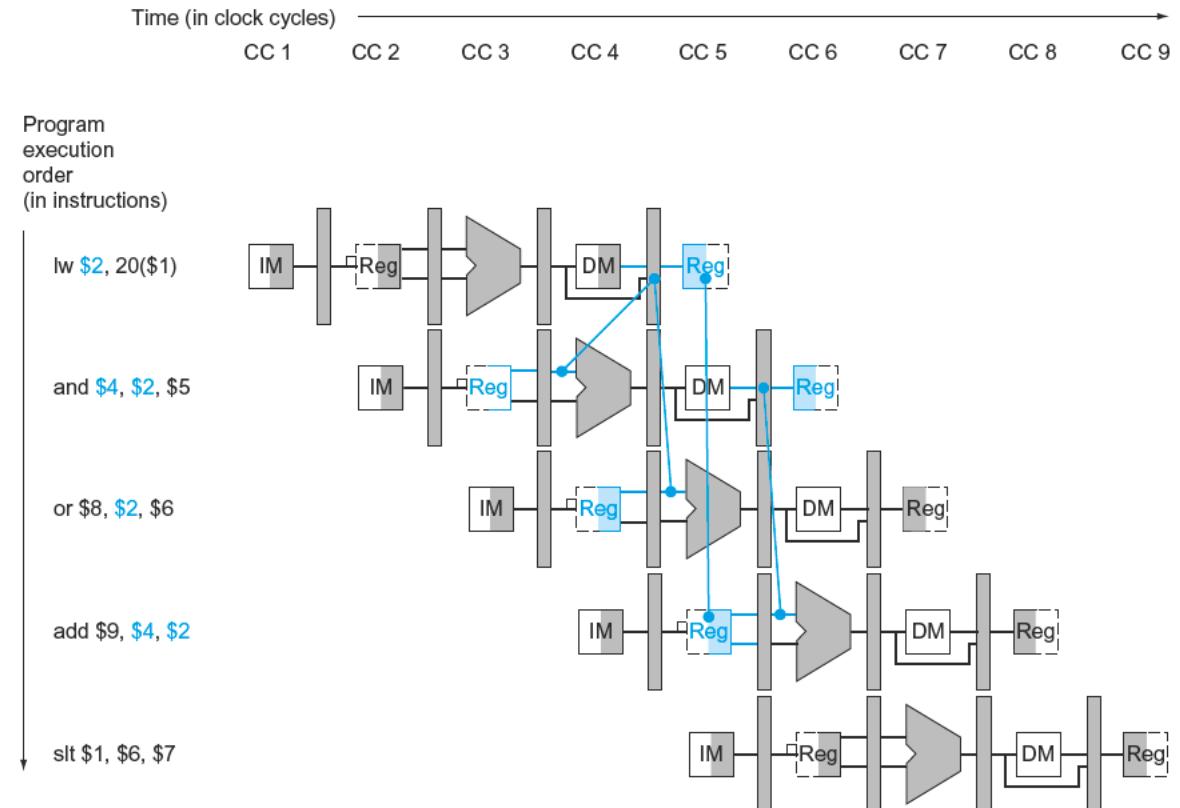


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

use. Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline
```

The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage. If the condition holds, the instruction stalls one clock cycle. After this 1-cycle stall, the forwarding logic can handle the dependence and execution proceeds. (If there were no forwarding, then the instructions in [Figure 4.58](#) would need another stall cycle.)

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register. Returning to our favorite analogy, it's as if you restart the washer with the same clothes and let the dryer continue tumbling empty. Of course, like the dryer, the back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: [nops](#).

nop An instruction that does no operation to change state.

How can we insert these nops, which act like bubbles, into the pipeline? In [Figure 4.49](#), we see that deasserting all nine control signals (setting them to 0) in the EX, MEM, and WB stages will create a “do nothing” or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These benign control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.

[Figure 4.59](#) shows what really happens in the hardware: the pipeline execution slot associated with the AND instruction is turned into a nop and all instructions beginning with the AND instruction are delayed one cycle. Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each cycle until it exits at the end. In this example, the hazard forces the AND and OR instructions to repeat in clock cycle 4 what they did in clock cycle 3: AND reads registers and decodes, and OR is refetched from instruction memory. Such repeated work is what a stall looks like, but its effect is to stretch the time of the AND and OR instructions and delay the fetch of the add instruction.

[Figure 4.60](#) highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU

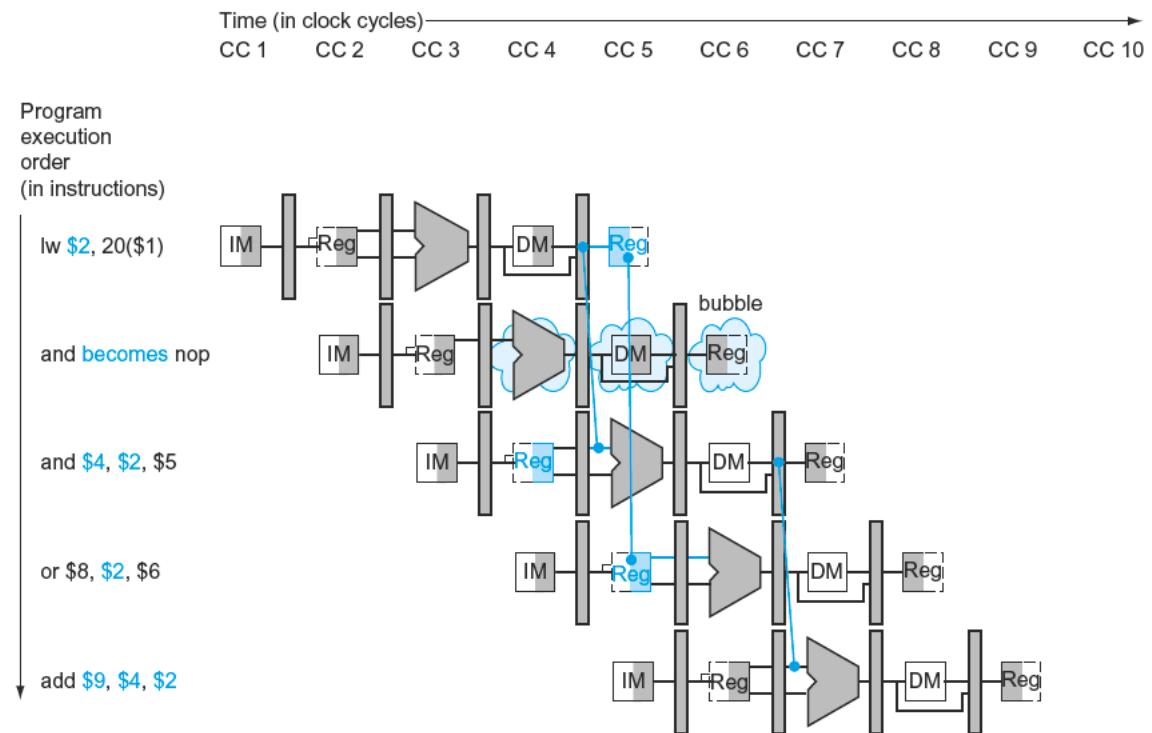


FIGURE 4.59 The way stalls are really Inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the OR Instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true. [Section 4.13](#) gives an example of MIPS code with hazards that causes stalling, illustrated using single-clock pipeline diagrams, if you would like to see more details.

Although the compiler generally relies upon the hardware to resolve hazards and thereby ensure correct execution, the compiler must understand the pipeline to achieve the best performance. Otherwise, unexpected stalls will reduce the performance of the compiled code.

The BIG Picture

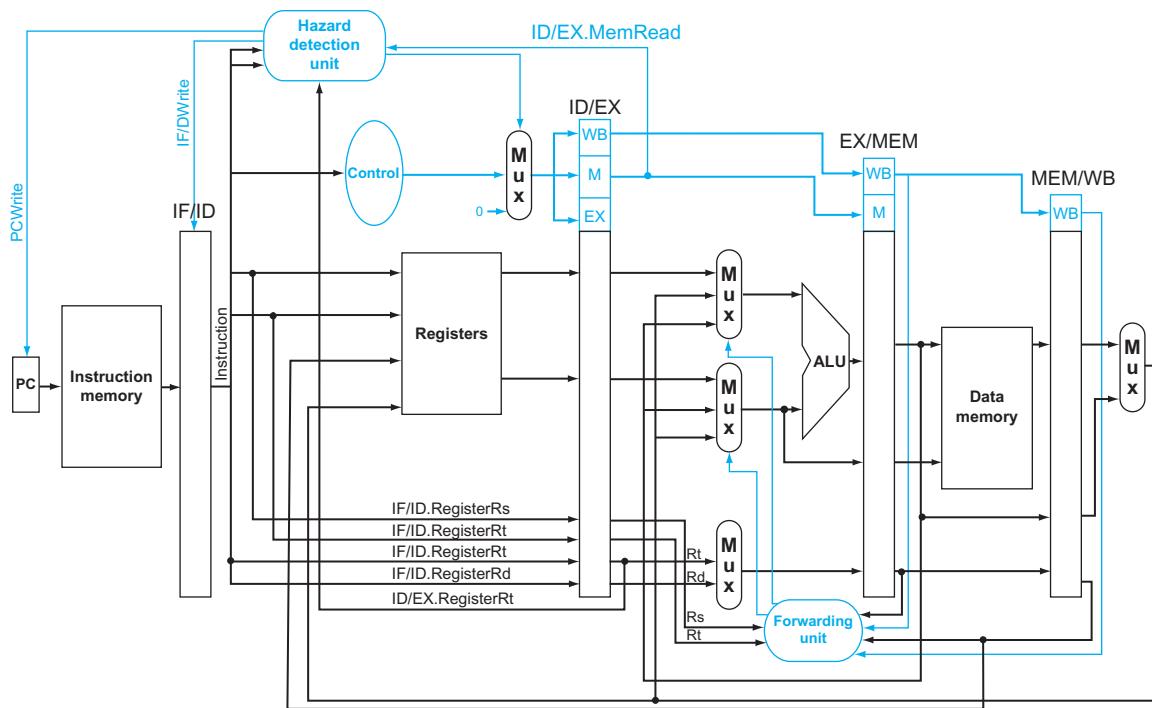


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Elaboration: Regarding the remark earlier about setting control lines to 0 to avoid writing registers or memory: only the signals RegWrite and MemWrite need be 0, while the other control signals can be don't cares.

*There are a thousand
hacking at the
branches of evil to one
who is striking at the
root.*

Henry David Thoreau,
Walden, 1854

4.8

Control Hazards

Thus far, we have limited our concern to hazards involving arithmetic operations and data transfers. However, as we saw in Section 4.5, there are also pipeline hazards involving branches. Figure 4.61 shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. As mentioned in Section 4.5,

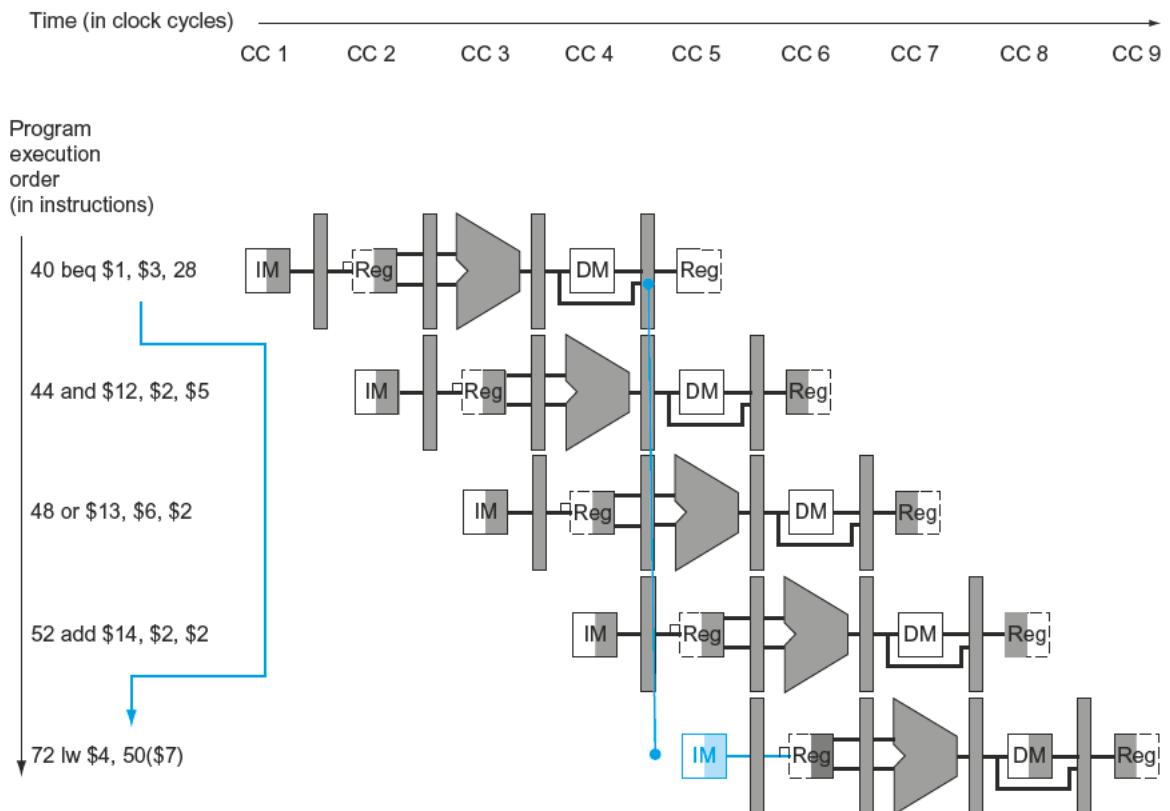


FIGURE 4.61 The Impact of the Pipeline on the Branch Instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72. (Figure 4.31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

this delay in determining the proper instruction to fetch is called a *control hazard* or *branch hazard*, in contrast to the *data hazards* we have just examined.

This section on control hazards is shorter than the previous sections on data hazards. The reasons are that control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards. Hence, we use simpler schemes. We look at two schemes for resolving control hazards and one optimization to improve these schemes.



PREDICTION

flush To discard instructions in a pipeline, usually due to an unexpected event.

Assume Branch Not Taken

As we saw in Section 4.5, stalling until the branch is complete is too slow. One improvement over branch stalling is to **predict** that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, we merely change the original control values to 0s, much as we did to stall for a load-use data hazard. The difference is that we must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage; for load-use stalls, we just change control to 0 in the ID stage and let them percolate through the pipeline. Discarding instructions, then, means we must be able to **flush** instructions in the IF, ID, and EX stages of the pipeline.

Reducing the Delay of Branches

One way to improve branch performance is to reduce the cost of the taken branch. Thus far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed. The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty. The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates. When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required—a situation that is similar to the use of condition codes for branches (see Chapter 2).

Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization. For example, to implement branch on equal (and its inverse), we will need to forward results to the equality test logic that operates during ID. There are two complicating factors:

1. During ID, we must decode the instruction, decide whether a bypass to the equality unit is needed, and complete the equality comparison so that if the instruction is a branch, we can set the PC to the branch target address.

Forwarding for the operands of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.

- Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch. By extension, if a load is immediately followed by a conditional branch that is on the load result, two stall cycles will be needed, as the result from the load appears at the end of the MEM cycle but is needed at the beginning of ID for the branch.

Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched. The exercises explore the details of implementing the forwarding path and detecting the hazard.

To flush instructions in the IF stage, we add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a `nop`, an instruction that has no action and changes no state.

Pipelined Branch

EXAMPLE

Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken and that we moved the branch execution to the ID stage:

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw $4, 50($7)
```

Figure 4.62 shows what happens when a branch is taken. Unlike Figure 4.61, there is only one pipeline bubble on a taken branch.

ANSWER

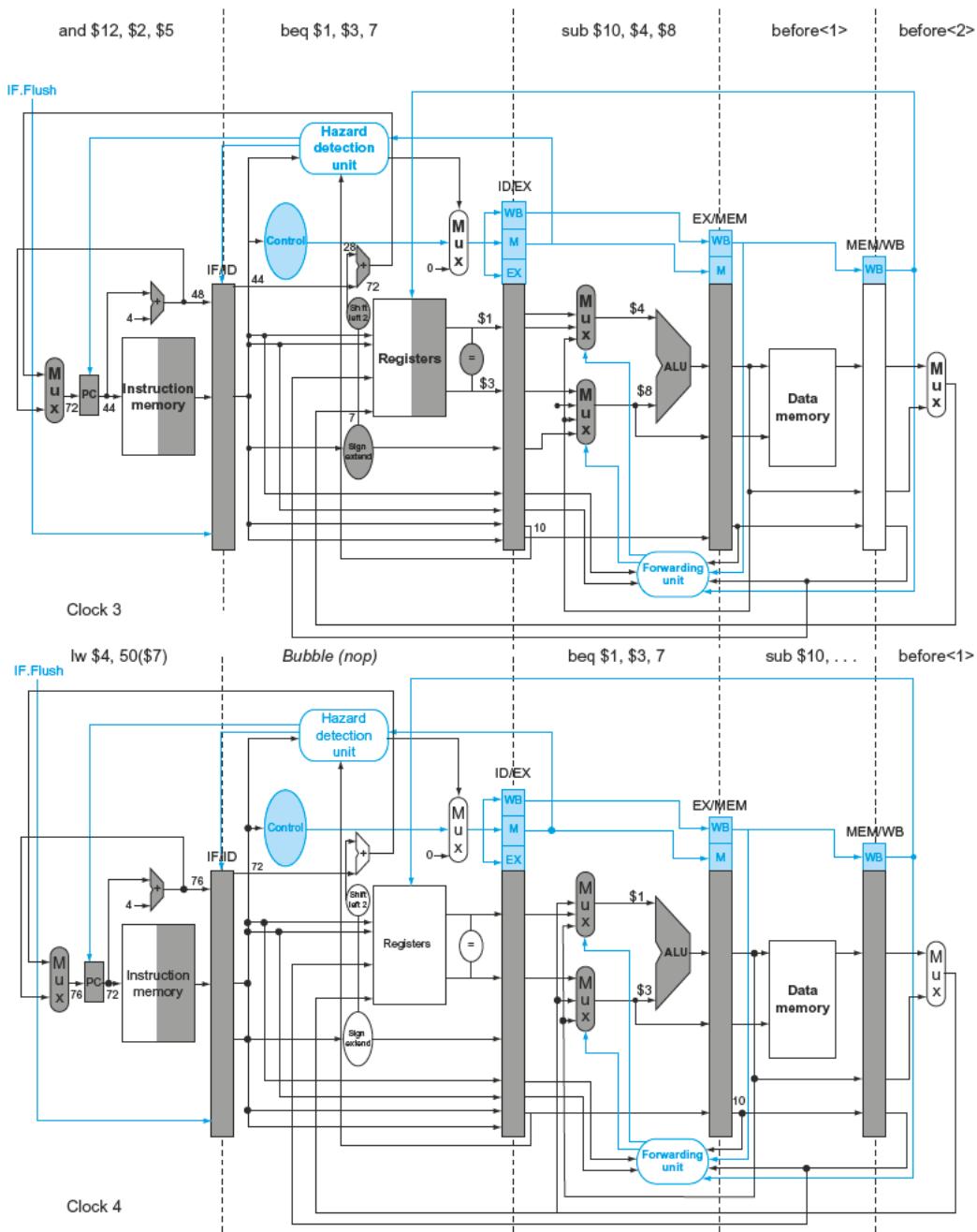


FIGURE 4.62 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or nop instruction in the pipeline as a result of the taken branch. (Since the nop is really `sl $0, $0, 0`, it's arguable whether or not the ID stage in clock 4 should be highlighted.)

Dynamic Branch Prediction

Assuming a branch is not taken is one simple form of *branch prediction*. In that case, we predict that branches are untaken, flushing the pipeline when we are wrong. For the simple five-stage pipeline, such an approach, possibly coupled with compiler-based prediction, is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issue (see Section 4.10), the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance. As we mentioned in Section 4.5, with more hardware it is possible to try to **predict** branch behavior during program execution.

One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called **dynamic branch prediction**.

One implementation of that approach is a **branch prediction buffer** or **branch history table**. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

This is the simplest sort of buffer; we don't know, in fact, if the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.



PREDICTION

dynamic branch prediction

Prediction of branches at runtime using runtime information.

branch prediction bufer

Also called **branch history table**.

A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

Loops and Prediction

Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

EXAMPLE

The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will indicate taken, as the branch has been taken nine times in a row at that point. The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration. Thus, the prediction accuracy for this

ANSWER

branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones).

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed. Figure 4.63 shows the finite-state machine for a 2-bit prediction scheme.

A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; as mentioned on page 318, it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 4.63.

branch delay slot The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

Elaboration: As we described in Section 4.5, in a five-stage pipeline we can make the control hazard a feature by redefining the branch. A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch.

Compilers and assemblers try to place an instruction that always executes after the branch in the **branch delay slot**. The job of the software is to make the successor instructions valid and useful. Figure 4.64 shows the three ways in which the branch delay slot can be scheduled.

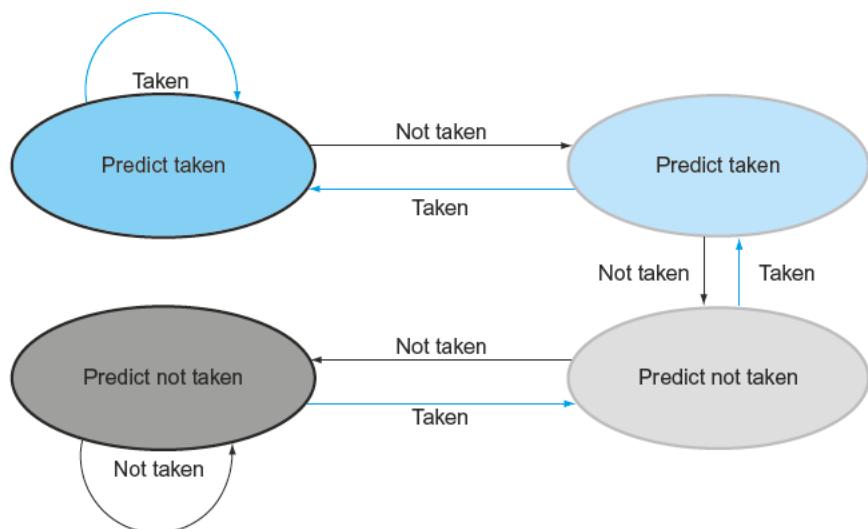


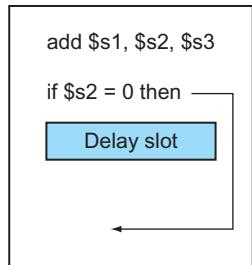
FIGURE 4.63 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

The limitations on delayed branch scheduling arise from (1) the restrictions on the instructions that are scheduled into the delay slots and (2) our ability to predict at compile time whether a branch is likely to be taken or not.

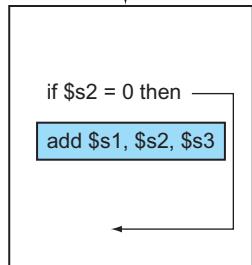
Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle. As processors go to both longer pipelines and issuing multiple instructions per clock cycle (see Section 4.10), the branch delay becomes longer, and a single delay slot is insufficient. Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches. Simultaneously, the growth in available transistors per chip has due to **Moore's Law** made dynamic prediction relatively cheaper.



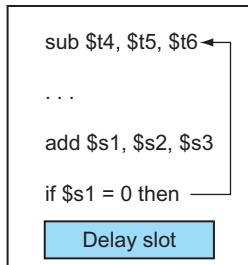
a. From before



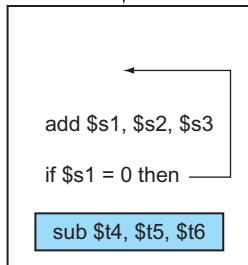
Becomes



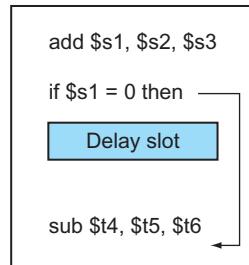
b. From target



Becomes



c. From fall-through



Becomes

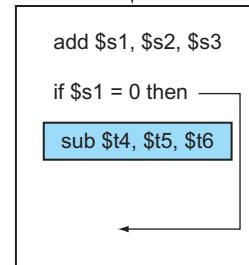


FIGURE 4.64 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of $\$s1$ in the branch condition prevents the add instruction (whose destination is $\$s1$) from being moved into the branch delay slot. In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if $\$t4$ were an unused temporary register when the branch goes in the unexpected direction.

branch target buffer

A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

correlating predictor

A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

tournament branch predictor

A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

Elaboration: A branch predictor tells us whether or not a branch is taken, but still requires the calculation of the branch target. In the five-stage pipeline, this calculation takes one cycle, meaning that taken branches will have a 1-cycle penalty. Delayed branches are one approach to eliminate that penalty. Another approach is to use a cache to hold the destination program counter or destination instruction using a **branch target buffer**.

The 2-bit dynamic prediction scheme uses only information about a particular branch. Researchers noticed that using information about both a local branch, and the global behavior of recently executed branches together yields greater prediction accuracy for the same number of prediction bits. Such predictors are called **correlating predictors**. A typical correlating predictor might have two 2-bit predictors for each branch, with the choice between predictors made based on whether the last executed branch was taken or not taken. Thus, the global branch behavior can be thought of as adding additional index bits for the prediction lookup.

A more recent innovation in branch prediction is the use of tournament predictors. A **tournament predictor** uses multiple predictors, tracking, for each branch, which predictor yields the best results. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behavior. A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor, favoring whichever of the two predictors has been more accurate. Some recent microprocessors use such elaborate predictors.

Elaboration: One way to reduce the number of conditional branches is to add *conditional move* instructions. Instead of changing the PC with a conditional branch, the instruction conditionally changes the destination register of the move. If the condition fails, the move acts as a *nop*. For example, one version of the MIPS instruction set architecture has two new instructions called *movn* (move if not zero) and *movz* (move if zero). Thus, *movn \$8, \$11, \$4* copies the contents of register 11 into register 8, provided that the value in register 4 is nonzero; otherwise, it does nothing.

The ARMv7 instruction set has a condition field in most instructions. Hence, ARM programs could have fewer conditional branches than in MIPS programs.

Pipeline Summary

We started in the laundry room, showing principles of pipelining in an everyday setting. Using that analogy as a guide, we explained instruction pipelining step-by-step, starting with the single-cycle datapath and then adding pipeline registers, forwarding paths, data hazard detection, branch prediction, and flushing instructions on exceptions. [Figure 4.65](#) shows the final evolved datapath and control. We now are ready for yet another control hazard: the sticky issue of exceptions.

Check Yourself

Consider three branch prediction schemes: predict not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and two cycles when they are wrong. Assume that the average predict

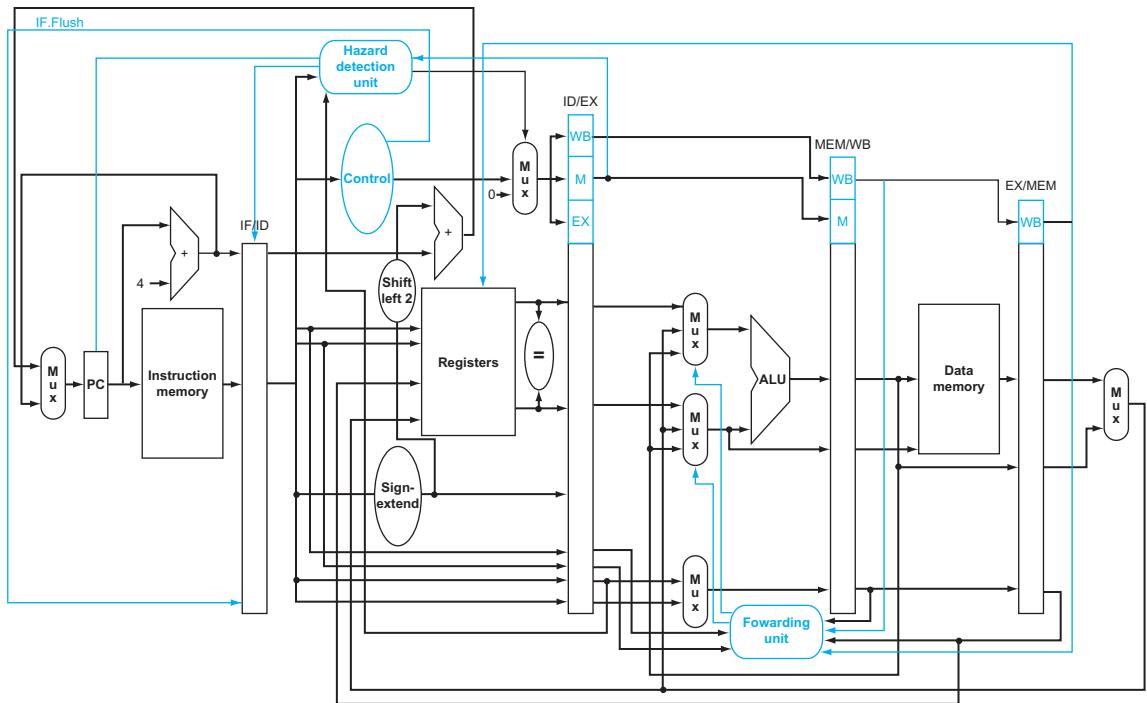


FIGURE 4.65 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

accuracy of the dynamic predictor is 90%. Which predictor is the best choice for the following branches?

1. A branch that is taken with 5% frequency
2. A branch that is taken with 95% frequency
3. A branch that is taken with 70% frequency

4.9

Exceptions

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make fast. One of the hardest parts of

To make a computer with automatic program-interruption facilities behave [sequentially] was not an easy matter, because the number of instructions in various stages of processing when an interrupt signal occurs may be large.

Fred Brooks, Jr.,
Planning a Computer System: Project Stretch,
1962

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.)

control is implementing **exceptions** and **interrupts**—events other than branches or jumps that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like arithmetic overflow. The same basic mechanism was extended for I/O devices to communicate with the processor, as we will see in Chapter 5.

Many architectures and authors do not distinguish between interrupts and exceptions, often using the older name *interrupt* to refer to both types of events. For example, the Intel x86 uses interrupt. We follow the MIPS convention, using the term *exception* to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term *interrupt* only when the event is externally caused. Here are five examples showing whether the situation is internally generated by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Many of the requirements to support exceptions come from the specific situation that causes an exception to occur. Accordingly, we will return to this topic in Chapter 5, when we will better understand the motivation for additional capabilities in the exception mechanism. In this section, we deal with the control implementation for detecting two types of exceptions that arise from the portions of the instruction set and implementation that we have already discussed.

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and thus performance. Without proper attention to exceptions during design of the control unit, attempts to add exceptions to a complicated implementation can significantly reduce performance, as well as complicate the task of getting the design correct.

How Exceptions Are Handled in the MIPS Architecture

The two types of exceptions that our current implementation can generate are execution of an undefined instruction and an arithmetic overflow. We'll use arithmetic overflow in the instruction `add $1, $2, $1` as the example exception in the next few pages. The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the *exception program counter* (EPC) and then transfer control to the operating system at some specified address.

The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in

response to an overflow, or stopping the execution of the program and reporting an error. After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program. In Chapter 5, we will look more closely at the issue of restarting the execution.

For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception. The method used in the MIPS architecture is to include a status register (called the *Cause register*), which holds a field that indicates the reason for the exception.

A second method, is to use **vectored interrupts**. In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception. For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000 _{hex}
Arithmetic overflow	8000 0180 _{hex}

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of the exception.

The operating system knows the reason for the exception by the address at which it is initiated. The addresses are separated by 32 bytes or eight instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.

We can perform the processing required for exceptions by adding a few extra registers and control signals to our basic implementation and by slightly extending control. Let's assume that we are implementing the exception system used in the MIPS architecture, with the single entry point being the address 8000 0180_{hex}. (Implementing vectored exceptions is no more difficult.) We will need to add two additional registers to our current MIPS implementation:

- **EPC:** A 32-bit register used to hold the address of the affected instruction. (Such a register is needed even when exceptions are vectored.)
- **Cause:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits, although some bits are currently unused. Assume there is a five-bit field that encodes the two possible exception sources mentioned above, with 10 representing an undefined instruction and 12 representing arithmetic overflow.

Exceptions in a Pipelined Implementation

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. Just as

we did for the taken branch in the previous section, we must flush the instructions that follow the `add` instruction from the pipeline and begin fetching instructions from the new address. We will use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines.

When we dealt with branch mispredict, we saw how to flush the instruction in the IF stage by turning it into a `nop`. To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called `ID.Flush`, is ORed with the stall signal from the hazard detection unit to flush during ID. To flush the instruction in the EX phase, we use a new signal called `EX.Flush` to cause new multiplexors to zero the control lines. To start fetching instructions from location $8000\ 0180_{hex}$, which is the MIPS exception address, we simply add an additional input to the PC multiplexor that sends $8000\ 0180_{hex}$ to the PC. [Figure 4.66](#) shows these changes.

This example points out a problem with exceptions: if we do not stop execution in the middle of the instruction, the programmer will not be able to see the original value of register \$1 that helped cause the overflow because it will be clobbered as the Destination register of the `add` instruction. Because of careful planning, the overflow exception is detected during the EX stage; hence, we can use the `EX.Flush` signal to prevent the instruction in the EX stage from writing its result in the WB stage. Many exceptions require that we eventually complete the instruction that caused the exception as if it executed normally. The easiest way to do this is to flush the instruction and restart it from the beginning after the exception is handled.

The final step is to save the address of the offending instruction in the *exception program counter* (EPC). In reality, we save the address +4, so the exception handling software routine must first subtract 4 from the saved value. [Figure 4.66](#) shows a stylized version of the datapath, including the branch hardware and necessary accommodations to handle exceptions.

EXAMPLE

Exception in a Pipelined Computer

Given this instruction sequence,

```
40hex    sub   $11, $2, $4
44hex    and   $12, $2, $5
48hex    or    $13, $2, $6
4Chex    add   $1, $2, $1
50hex    slt   $15, $6, $7
54hex    lw    $16, 50($7)
...

```

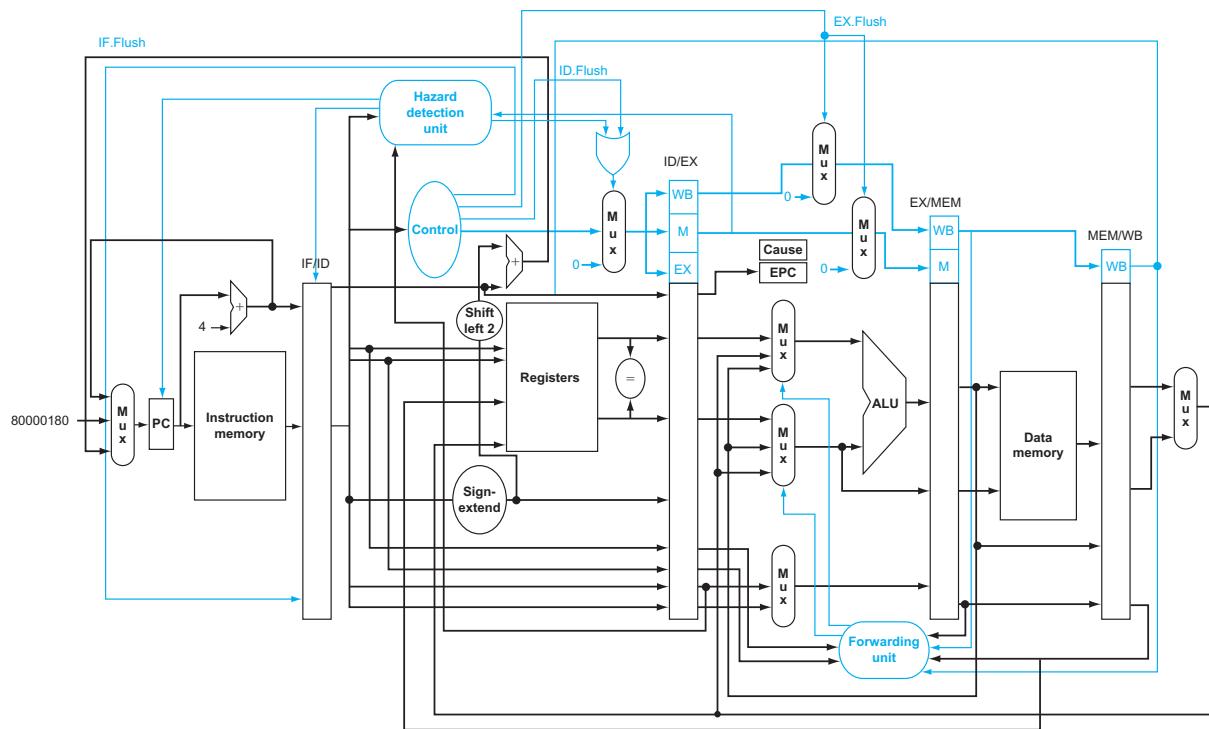


FIGURE 4.66 The datapath with controls to handle exceptions. The key additions include a new input with the value 8000 0180_{hex} in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180_{hex} input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

assume the instructions to be invoked on an exception begin like this:

80000180 _{hex}	SW	\$26, 1000(\$0)
80000184 _{hex}	SW	\$27, 1004(\$0)

...

Show what happens in the pipeline if an overflow exception occurs in the add instruction.

Figure 4.67 shows the events, starting with the add instruction in the EX stage. The overflow is detected during that phase, and 8000 0180_{hex} is forced into the PC. Clock cycle 7 shows that the add and following instructions are flushed, and the first instruction of the exception code is fetched. Note that the address of the instruction following the add is saved: 4C_{hex} + 4 = 50_{hex}.

ANSWER

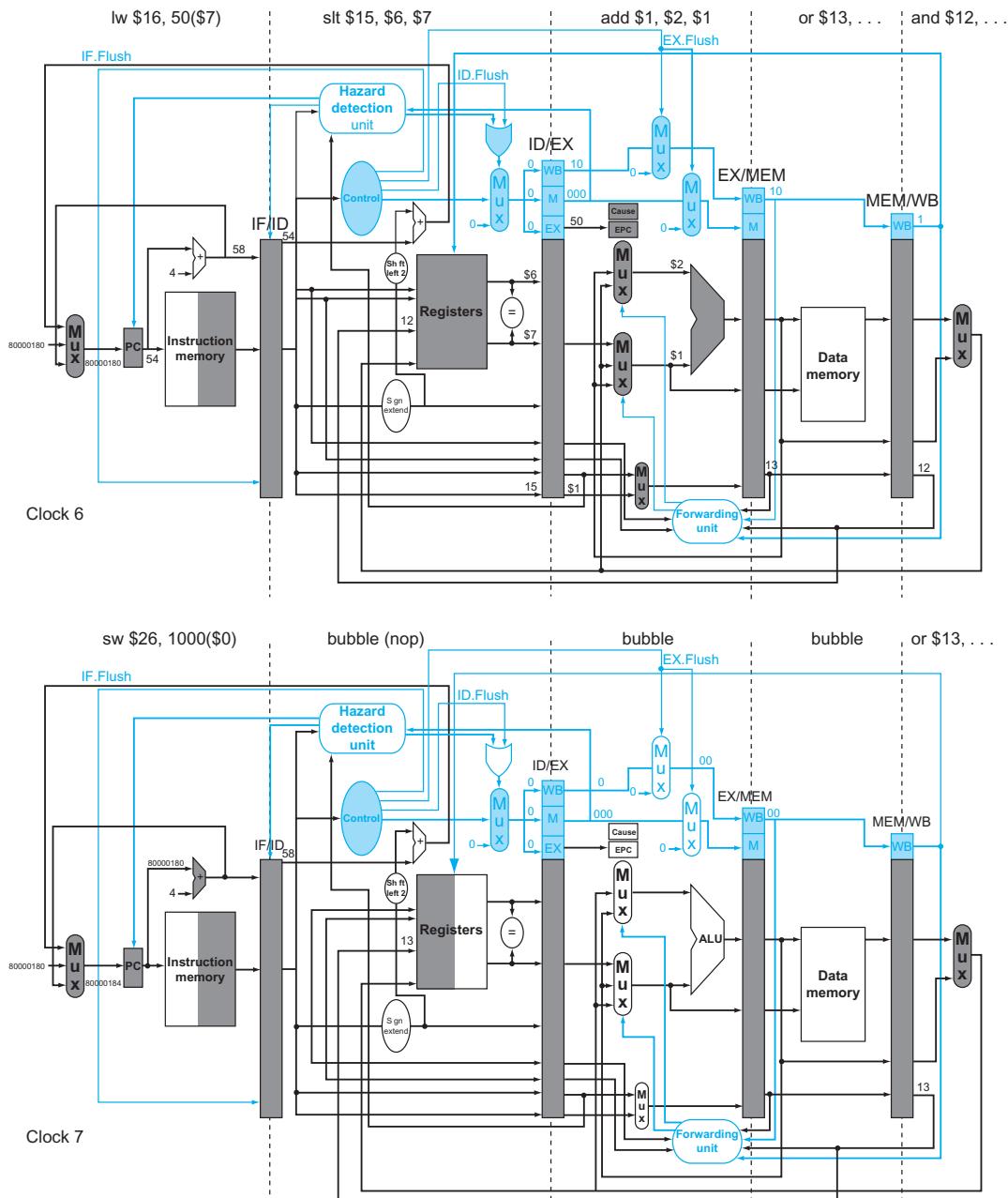


FIGURE 4.67 The result of an exception due to arithmetic overflow in the add instruction. The overflow is detected during the EX stage of clock 6, saving the address following the add in the EPC register ($4C + 4 = 50_{hex}$). Overflow causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the add. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—`SW $25, 1000($0)`—from instruction location $8000\ 0180_{hex}$. Note that the AND and OR instructions, which are prior to the add, still complete. Although not shown, the ALU overflow signal is an input to the control unit.

We mentioned five examples of exceptions on page 326, and we will see others in Chapter 5. With five instructions active in any clock cycle, the challenge is to associate an exception with the appropriate instruction. Moreover, multiple exceptions can occur simultaneously in a single clock cycle. The solution is to prioritize the exceptions so that it is easy to determine which is serviced first. In most MIPS implementations, the hardware sorts exceptions so that the earliest instruction is interrupted.

I/O device requests and hardware malfunctions are not associated with a specific instruction, so the implementation has some flexibility as to when to interrupt the pipeline. Hence, the mechanism used for other exceptions works just fine.

The EPC captures the address of the interrupted instructions, and the MIPS Cause register records all possible exceptions in a clock cycle, so the exception software must match the exception to the instruction. An important clue is knowing in which pipeline stage a type of exception can occur. For example, an undefined instruction is discovered in the ID stage, and invoking the operating system occurs in the EX stage. Exceptions are collected in the Cause register in a pending exception field so that the hardware can interrupt based on later exceptions, once the earliest one has been serviced.

The hardware and the operating system must work in conjunction so that exceptions behave as you would expect. The hardware contract is normally to stop the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address. The operating system contract is to look at the cause of the exception and act appropriately. For an undefined instruction, hardware failure, or arithmetic overflow exception, the operating system normally kills the program and returns an indicator of the reason. For an I/O device request or an operating system service call, the operating system saves the state of the program, performs the desired task, and, at some point in the future, restores the program to continue execution. In the case of I/O device requests, we may often choose to run another task before resuming the task that requested the I/O, since that task may often not be able to proceed until the I/O is complete. Exceptions are why the ability to save and restore the state of any task is critical. One of the most important and frequent uses of exceptions is handling page faults and TLB exceptions; Chapter 5 describes these exceptions and their handling in more detail.

Hardware/ Software Interface

imprecise interrupt Also called **imprecise exception**.

Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.

Elaboration: The difficulty of always associating the correct exception with the correct instruction in pipelined computers has led some computer designers to relax this requirement in noncritical cases. Such processors are said to have **imprecise interrupts** or **imprecise exceptions**. In the example above, PC would normally have 58_{hex} at the start of the clock cycle after the exception is detected, even though the offending instruction

precise interrupt Also called **precise exception**. An interrupt or exception that is always associated with the correct instruction in pipelined computers.

is at address $4C_{hex}$. A processor with imprecise exceptions might put 58_{hex} into EPC and leave it up to the operating system to determine which instruction caused the problem. MIPS and the vast majority of computers today support **precise interrupts** or **precise exceptions**. (One reason is to support virtual memory, which we shall see in Chapter 5.)

Elaboration: Although MIPS uses the exception entry address $8000\ 0180_{hex}$ for almost all exceptions, it uses the address $8000\ 0000_{hex}$ to improve performance of the exception handler for TLB-miss exceptions (see Chapter 5).

Check Yourself

Which exception should be recognized first in this sequence?

1. add \$1, \$2, \$1 # arithmetic overflow
2. XXX \$1, \$2, \$1 # undefined instruction
3. sub \$1, \$2, \$1 # hardware error

4.10

Parallelism via Instructions



instruction-level parallelism The parallelism among instructions.

multiple issue A scheme whereby multiple instructions are launched in one clock cycle.

Be forewarned: this section is a brief overview of fascinating but advanced topics. If you want to learn more details, you should consult our more advanced book, *Computer Architecture: A Quantitative Approach*, fifth edition, where the material covered in these 13 pages is expanded to almost 200 pages (including appendices)!

Pipelining exploits the potential **parallelism** among instructions. This parallelism is called **instruction-level parallelism (ILP)**. There are two primary methods for increasing the potential amount of instruction-level parallelism. The first is increasing the depth of the pipeline to overlap more instructions. Using our laundry analogy and assuming that the washer cycle was longer than the others were, we could divide our washer into three machines that perform the wash, rinse, and spin steps of a traditional washer. We would then move from a four-stage to a six-stage pipeline. To get the full speed-up, we need to rebalance the remaining steps so they are the same length, in processors or in laundry. The amount of parallelism being exploited is higher, since there are more operations being overlapped. Performance is potentially greater since the clock cycle can be shorter.

Another approach is to replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage. The general name for this technique is **multiple issue**. A multiple-issue laundry would replace our household washer and dryer with, say, three washers and three dryers. You would also have to recruit more assistants to fold and put away three times as much laundry in the same amount of time. The downside is the extra work to keep all the machines busy and transferring the loads to the next pipeline stage.

Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, the CPI to be less than 1. As mentioned in Chapter 1, it is sometimes useful to flip the metric and use *IPC*, or *instructions per clock cycle*. Hence, a 4 GHz four-way multiple-issue microprocessor can execute a peak rate of 16 billion instructions per second and have a best-case CPI of 0.25, or an IPC of 4. Assuming a five-stage pipeline, such a processor would have 20 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle. Even moderate designs will aim at a peak IPC of 2. There are typically, however, many constraints on what types of instructions may be executed simultaneously, and what happens when dependences arise.

There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. Because the division of work dictates whether decisions are being made statically (that is, at compile time) or dynamically (that is, during execution), the approaches are sometimes called **static multiple issue** and **dynamic multiple issue**. As we will see, both approaches have other, more commonly used names, which may be less precise or more restrictive.

There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

1. Packaging instructions into **issue slots**: how does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler; in dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.
2. Dealing with data and control hazards: in static issue processors, the compiler handles some or all of the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time.

Although we describe these as distinct approaches, in reality one approach often borrows techniques from the other, and neither approach can claim to be perfectly pure.

The Concept of Speculation

One of the most important methods for finding and exploiting more ILP is speculation. Based on the great idea of **prediction**, **speculation** is an approach that allows the compiler or the processor to “guess” about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction. For example, we might speculate on the outcome of a branch, so that instructions after the branch could be executed earlier.

static multiple issue An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

dynamic multiple issue An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

issue slots The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.



PREDICTION

speculation An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

Another example is that we might speculate that a store that precedes a load does not refer to the same address, which would allow the load to be executed before the store. The difficulty with speculation is that it may be wrong. So, any speculation mechanism must include both a method to check if the guess was right and a method to unroll or back out the effects of the instructions that were executed speculatively. The implementation of this back-out capability adds complexity.

Speculation may be done in the compiler or by the hardware. For example, the compiler can use speculation to reorder instructions, moving an instruction across a branch or a load across a store. The processor hardware can perform the same transformation at runtime using techniques we discuss later in this section.

The recovery mechanisms used for incorrect speculation are rather different. In the case of speculation in software, the compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to use when the speculation is incorrect. In hardware speculation, the processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation is correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation is incorrect, the hardware flushes the buffers and re-executes the correct instruction sequence.

Speculation introduces one other possible problem: speculating on certain instructions may introduce exceptions that were formerly not present. For example, suppose a load instruction is moved in a speculative manner, but the address it uses is not legal when the speculation is incorrect. The result would be an exception that should not have occurred. The problem is complicated by the fact that if the load instruction were not speculative, then the exception must occur! In compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point the exception is raised, and normal exception handling proceeds.

Since speculation can improve performance when done properly and decrease performance when done carelessly, significant effort goes into deciding when it is appropriate to speculate. Later in this section, we will examine both static and dynamic techniques for speculation.

issue packet The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Static Multiple Issue

Static multiple-issue processors all use the compiler to assist with packaging instructions and handling hazards. In a static issue processor, you can think of the set of instructions issued in a given clock cycle, which is called an **issue packet**, as one large instruction with multiple operations. This view is more than an analogy. Since a static multiple-issue processor usually restricts what mix of instructions can be initiated in a given clock cycle, it is useful to think of the issue packet as a single

instruction allowing several operations in certain predefined fields. This view led to the original name for this approach: **Very Long Instruction Word (VLIW)**.

Most static issue processors also rely on the compiler to take on some responsibility for handling data and control hazards. The compiler's responsibilities may include static branch prediction and code scheduling to reduce or prevent all hazards. Let's look at a simple static issue version of a MIPS processor, before we describe the use of these techniques in more aggressive processors.

An Example: Static Multiple Issue with the MIPS ISA

To give a flavor of static multiple issue, we consider a simple two-issue MIPS processor, where one of the instructions can be an integer ALU operation or branch and the other can be a load or store. Such a design is like that used in some embedded MIPS processors. Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. In many static multiple-issue processors, and essentially all VLIW processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue. Hence, we will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. Furthermore, if one instruction of the pair cannot be used, we require that it be replaced with a `nop`. Thus, the instructions always issue in pairs, possibly with a `nop` in one slot. [Figure 4.68](#) shows how the instructions look as they go into the pipeline in pairs.

Static multiple-issue processors vary in how they deal with potential data and control hazards. In some designs, the compiler takes full responsibility for removing *all* hazards, scheduling the code and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls. In others, the hardware detects data hazards and generates stalls between two issue packets, while requiring that the compiler avoid all dependences within an instruction pair. Even so, a hazard generally forces the entire issue packet containing the dependent

Very Long Instruction Word (VLIW)

A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

Instruction type	Pipe stages						
ALU or branch instruction	IF	ID	EX	MEM	WB		
Load or store instruction	IF	ID	EX	MEM	WB		
ALU or branch instruction		IF	ID	EX	MEM	WB	
Load or store instruction		IF	ID	EX	MEM	WB	
ALU or branch instruction			IF	ID	EX	MEM	WB
Load or store instruction			IF	ID	EX	MEM	WB
ALU or branch instruction				IF	ID	EX	MEM
Load or store instruction				IF	ID	EX	WB

FIGURE 4.68 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

instruction to stall. Whether the software must handle all hazards or only try to reduce the fraction of hazards between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced. We will assume the second approach for this example.

To issue an ALU and a data transfer operation in parallel, the first need for additional hardware—beyond the usual hazard detection and stall logic—is extra ports in the register file (see Figure 4.69). In one clock cycle we may need to read two registers for the ALU operation and two more for a store, and also one write port for an ALU operation and one write port for a load. Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

Clearly, this two-issue processor can improve performance by up to a factor of two. Doing so, however, requires that twice as many instructions be overlapped in execution, and this additional overlap increases the relative performance loss from data and control hazards. For example, in our simple five-stage pipeline,

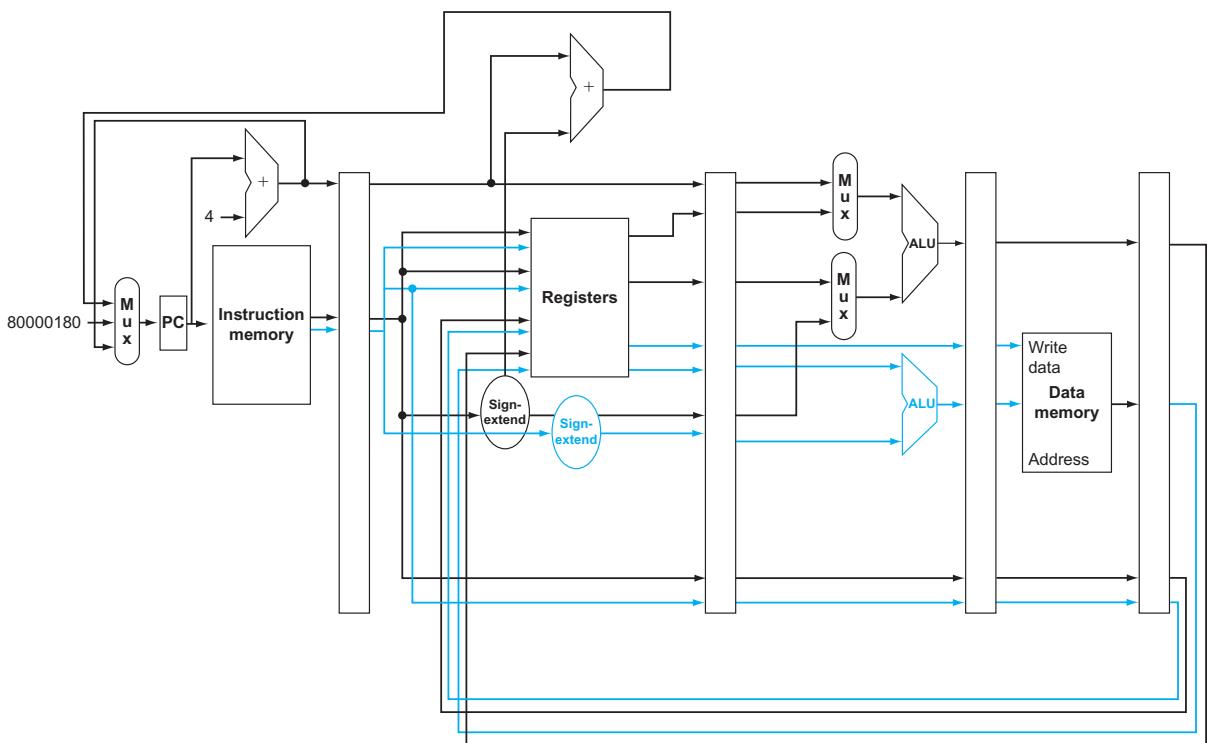


FIGURE 4.69 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

loads have a **use latency** of one clock cycle, which prevents one instruction from using the result without stalling. In the two-issue, five-stage pipeline the result of a load instruction cannot be used on the next *clock cycle*. This means that the next *two* instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline now have a one-instruction use latency, since the results cannot be used in the paired load or store. To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler take on this role.

use latency Number of clock cycles between a load instruction and an instruction that can use the result of the load without stalling the pipeline.

Simple Multiple-Issue Code Scheduling

How would this loop be scheduled on a static two-issue pipeline for MIPS?

```
Loop: lw      $t0, 0($s1)    # $t0=array element
      addu   $t0,$t0,$s2# add scalar in $s2
      sw      $t0, 0($s1)# store result
      addi   $s1,$s1,-4# decrement pointer
      bne    $s1,$zero,Loop# branch $s1!=0
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

EXAMPLE

The first three instructions have data dependences, and so do the last two. [Figure 4.70](#) shows the best schedule for these instructions. Notice that just one pair of instructions has both issue slots used. It takes four clocks per loop iteration; at four clocks to execute five instructions, we get the disappointing CPI of 0.8 versus the best case of 0.5., or an IPC of 1.25 versus 2.0. Notice that in computing CPI or IPC, we do not count any nops executed as useful instructions. Doing so would improve CPI, but not performance!

ANSWER

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw \$t0, 0(\$s1)	1
	addi \$s1,\$s1,-4		2
	addu \$t0,\$t0,\$s2		3
	bne \$s1,\$zero,Loop	sw \$t0, 4(\$s1)	4

FIGURE 4.70 The scheduled code as it would look on a two-issue MIPS pipeline. The empty slots are no-ops.

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

EXAMPLE**ANSWER**

register renaming The renaming of registers by the compiler or hardware to remove antidependences.

antidependence Also called **name dependence**. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

An important compiler technique to get more performance from loops is **loop unrolling**, where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations.

Loop Unrolling for Multiple-Issue Pipelines

See how well loop unrolling and scheduling work in the example above. For simplicity assume that the loop index is a multiple of four.

To schedule the loop without any delays, it turns out that we need to make four copies of the loop body. After unrolling and eliminating the unnecessary loop overhead instructions, the loop will contain four copies each of `lw`, `add`, and `sw`, plus one `addi` and one `bne`. Figure 4.71 shows the unrolled and scheduled code.

During the unrolling process, the compiler introduced additional registers (`$t1`, `$t2`, `$t3`). The goal of this process, called **register renaming**, is to eliminate dependences that are not true data dependences, but could either lead to potential hazards or prevent the compiler from flexibly scheduling the code. Consider how the unrolled code would look using only `$t0`. There would be repeated instances of `lw $t0, 0($$s1)`, `addu $t0, $t0, $s2` followed by `sw t0, 4($$s1)`, but these sequences, despite using `$t0`, are actually completely independent—no data values flow between one set of these instructions and the next set. This case is what is called an **antidependence** or **name dependence**, which is an ordering forced purely by the reuse of a name, rather than a real data dependence that is also called a true dependence.

Renaming the registers during the unrolling process allows the compiler to move these independent instructions subsequently so as to better schedule

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:			
	addi \$s1,\$s1,-16	lw \$t0, 0(\$\$s1)	1
		lw \$t1,12(\$\$s1)	2
	addu \$t0,\$t0,\$s2	lw \$t2, 8(\$\$s1)	3
	addu \$t1,\$t1,\$s2	lw \$t3, 4(\$\$s1)	4
	addu \$t2,\$t2,\$s2	sw \$t0, 16(\$\$s1)	5
	addu \$t3,\$t3,\$s2	sw \$t1,12(\$\$s1)	6
		sw \$t2, 8(\$\$s1)	7
	bne \$s1,\$zero,Loop	sw \$t3, 4(\$\$s1)	8

FIGURE 4.71 The unrolled and scheduled code of Figure 4.70 as it would look on a static two-issue MIPS pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements `$s1` by 16, the addresses loaded are the original value of `$s1`, then that address minus 4, minus 8, and minus 12.

the code. The renaming process eliminates the name dependences, while preserving the true dependences.

Notice now that 12 of the 14 instructions in the loop execute as pairs. It takes 8 clocks for 4 loop iterations, or 2 clocks per iteration, which yields a CPI of $8/14 = 0.57$. Loop unrolling and scheduling with dual issue gave us an improvement factor of almost 2, partly from reducing the loop control instructions and partly from dual issue execution. The cost of this performance improvement is using four temporary registers rather than one, as well as a significant increase in code size.

Dynamic Multiple-Issue Processors

Dynamic multiple-issue processors are also known as **superscalar** processors, or simply superscalars. In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle. Obviously, achieving good performance on such a processor still requires the compiler to try to schedule instructions to move dependences apart and thereby improve the instruction issue rate. Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: the code, whether scheduled or not, is guaranteed by the hardware to execute correctly. Furthermore, compiled code will always run correctly independent of the issue rate or pipeline structure of the processor. In some VLIW designs, this has not been the case, and recompilation was required when moving across different processor models; in other static issue processors, code would run correctly across different implementations, but often so poorly as to make compilation effectively required.

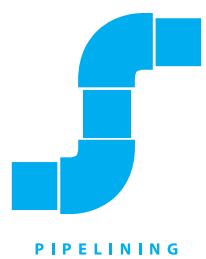
Many superscalars extend the basic framework of dynamic issue decisions to include **dynamic pipeline scheduling**. Dynamic pipeline scheduling chooses which instructions to execute in a given clock cycle while trying to avoid hazards and stalls. Let's start with a simple example of avoiding a data hazard. Consider the following code sequence:

```
lw      $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slti   $t5, $s4, 20
```

Even though the `sub` instruction is ready to execute, it must wait for the `lw` and `addu` to complete first, which might take many clock cycles if memory is slow. (Chapter 5 explains cache misses, the reason that memory accesses are sometimes very slow.) Dynamic **pipeline** scheduling allows such hazards to be avoided either fully or partially.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

dynamic pipeline scheduling Hardware support for reordering the order of instruction execution so as to avoid stalls.



PIPELINING

Dynamic Pipeline Scheduling

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls. In such processors, the pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units

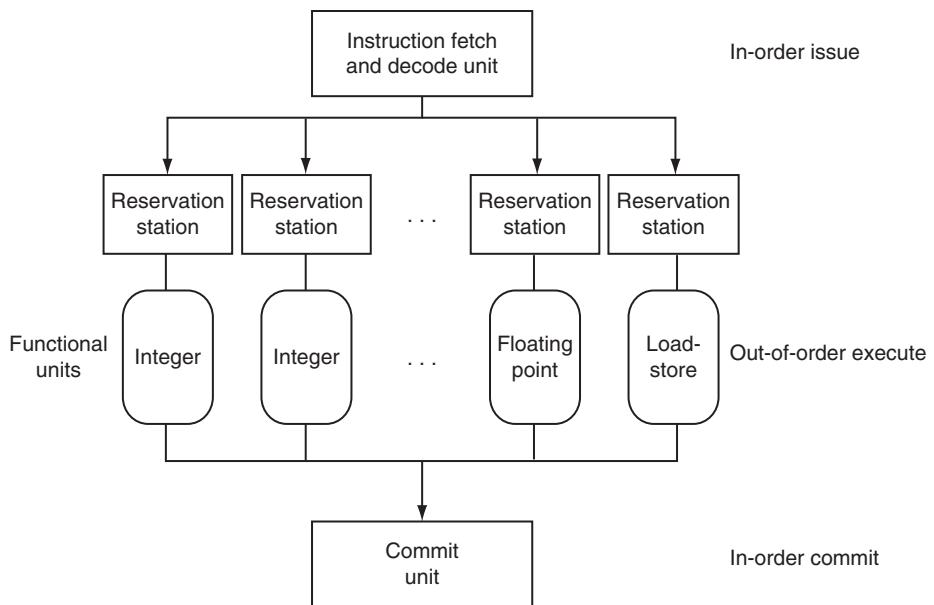


FIGURE 4.72 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

reservation station

A buffer within a functional unit that holds the operands and the operation.

reorder buffer The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

(a dozen or more in high-end designs in 2013), and a **commit unit**. Figure 4.72 shows the model. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution. Each functional unit has buffers, called **reservation stations**, which hold the operands and the operation. (The Elaboration discusses an alternative to reservation stations used by many recent processors.) As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, often called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming, just like that used by the compiler in our earlier loop-unrolling example on page 338. To see how this conceptually works, consider the following steps:

1. When an instruction issues, it is copied to a reservation station for the appropriate functional unit. Any operands that are available in the register file or reorder buffer are also immediately copied into the reservation station. The instruction is buffered in the reservation station until all the operands and the functional unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

These steps effectively use the reorder buffer and the reservation stations to implement register renaming.

Conceptually, you can think of a dynamically scheduled pipeline as analyzing the data flow structure of a program. The processor then executes the instructions in some order that preserves the data flow order of the program. This style of execution is called an **out-of-order execution**, since the instructions can be executed in a different order than they were fetched.

To make programs behave as if they were running on a simple in-order pipeline, the instruction fetch and decode unit is required to issue instructions in order, which allows dependences to be tracked, and the commit unit is required to write results to registers and memory in program fetch order. This conservative mode is called **in-order commit**. Hence, if an exception occurs, the computer can point to the last instruction executed, and the only registers updated will be those written by instructions before the instruction causing the exception. Although the front end (fetch and issue) and the back end (commit) of the pipeline run in order, the functional units are free to initiate execution whenever the data they need is available. Today, all dynamically scheduled pipelines use in-order commit.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path. Because the instructions are committed in order, we know whether or not the branch was correctly predicted before any instructions from the predicted path are committed. A speculative, dynamically scheduled pipeline can also support speculation on load addresses, allowing load-store reordering, and using the commit unit to avoid incorrect speculation. In the next section, we will look at the use of dynamic scheduling with speculation in the Intel Core i7 design.

out-of-order execution

A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

in-order commit

A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched.

Understanding Program Performance

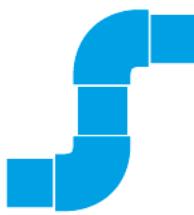


HIERARCHY



PREDICTION

The BIG Picture



PIPELINING



PARALLELISM



PREDICTION

Given that compilers can also schedule code around data dependences, you might ask why a superscalar processor would use dynamic scheduling. There are three major reasons. First, not all stalls are predictable. In particular, cache misses (see Chapter 5) in the **memory hierarchy** cause unpredictable stalls. Dynamic scheduling allows the processor to hide some of those stalls by continuing to execute instructions while waiting for the stall to end.

Second, if the processor speculates on branch outcomes using dynamic branch **prediction**, it cannot know the exact order of instructions at compile time, since it depends on the predicted and actual behavior of branches. Incorporating dynamic speculation to exploit more *instruction-level parallelism* (ILP) without incorporating dynamic scheduling would significantly restrict the benefits of speculation.

Third, as the pipeline latency and issue width change from one implementation to another, the best way to compile a code sequence also changes. For example, how to schedule a sequence of dependent instructions is affected by both issue width and latency. The pipeline structure affects both the number of times a loop must be unrolled to avoid stalls as well as the process of compiler-based register renaming. Dynamic scheduling allows the hardware to hide most of these details. Thus, users and software distributors do not need to worry about having multiple versions of a program for different implementations of the same instruction set. Similarly, old legacy code will get much of the benefit of a new implementation without the need for recompilation.

Both **pipelining** and multiple-issue execution increase peak instruction throughput and attempt to exploit instruction-level **parallelism** (ILP). Data and control dependences in programs, however, offer an upper limit on sustained performance because the processor must sometimes wait for a dependence to be resolved. Software-centric approaches to exploiting ILP rely on the ability of the compiler to find and reduce the effects of such dependences, while hardware-centric approaches rely on extensions to the pipeline and issue mechanisms. Speculation, performed by the compiler or the hardware, can increase the amount of ILP that can be exploited via **prediction**, although care must be taken since speculating incorrectly is likely to reduce performance.

Modern, high-performance microprocessors are capable of issuing several instructions per clock; unfortunately, sustaining that issue rate is very difficult. For example, despite the existence of processors with four to six issues per clock, very few applications can sustain more than two instructions per clock. There are two primary reasons for this.

First, within the pipeline, the major performance bottlenecks arise from dependences that cannot be alleviated, thus reducing the parallelism among instructions and the sustained issue rate. Although little can be done about true data dependences, often the compiler or hardware does not know precisely whether a dependence exists or not, and so must conservatively assume the dependence exists. For example, code that makes use of pointers, particularly in ways that may lead to aliasing, will lead to more implied potential dependences. In contrast, the greater regularity of array accesses often allows a compiler to deduce that no dependences exist. Similarly, branches that cannot be accurately predicted whether at runtime or compile time will limit the ability to exploit ILP. Often, additional ILP is available, but the ability of the compiler or the hardware to find ILP that may be widely separated (sometimes by the execution of thousands of instructions) is limited.

Second, losses in the **memory hierarchy** (the topic of Chapter 5) also limit the ability to keep the pipeline full. Some memory system stalls can be hidden, but limited amounts of ILP also limit the extent to which such stalls can be hidden.

Hardware/ Software Interface



Energy Efficiency and Advanced Pipelining

The downside to the increasing exploitation of instruction-level parallelism via dynamic multiple issue and speculation is potential energy inefficiency. Each innovation was able to turn more transistors into performance, but they often did so very inefficiently. Now that we have hit the power wall, we are seeing designs with multiple processors per chip where the processors are not as deeply pipelined or as aggressively speculative as its predecessors.

The belief is that while the simpler processors are not as fast as their sophisticated brethren, they deliver better performance per joule, so that they can deliver more performance per chip when designs are constrained more by energy than they are by number of transistors.

Figure 4.73 shows the number of pipeline stages, the issue width, speculation level, clock rate, cores per chip, and power of several past and recent microprocessors. Note the drop in pipeline stages and power as companies switch to multicore designs.

Elaboration: A commit unit controls updates to the register file and memory. Some dynamically scheduled processors update the register file immediately during execution, using extra registers to implement the renaming function and preserving the older copy of a register until the instruction updating the register is no longer speculative. Other processors buffer the result, typically in a structure called a reorder buffer, and the actual update to the register file occurs later as part of the commit. Stores to memory must be buffered until commit time either in a store buffer (see Chapter 5) or in the reorder buffer. The commit unit allows the store to write to memory from the buffer when the buffer has a valid address and valid data, and when the store is no longer dependent on predicted branches.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/Speculation	Cores/Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

FIGURE 4.73 Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

Elaboration: Memory accesses benefit from *nonblocking caches*, which continue servicing cache accesses during a cache miss (see Chapter 5). Out-of-order execution processors need the cache design to allow instructions to execute during a miss.

Check Yourself

State whether the following techniques or components are associated primarily with a software- or hardware-based approach to exploiting ILP. In some cases, the answer may be both.

1. Branch prediction
2. Multiple issue
3. VLIW
4. Superscalar
5. Dynamic scheduling
6. Out-of-order execution
7. Speculation
8. Reorder buffer
9. Register renaming

4.11

Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines

Figure 4.74 describes the two microprocessors we examine in this section, whose targets are the two bookends of the PostPC Era.

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	-	2 - 8 MiB

FIGURE 4.74 Specification of the ARM Cortex-A8 and the Intel Core i7 920.

The ARM Cortex-A8

The ARM Corxtex-A8 runs at 1 GHz with a 14-stage pipeline. It uses dynamic multiple issue, with two instructions per clock cycle. It is a static in-order pipeline, in that instructions issue, execute, and commit in order. The pipeline consists of three sections for instruction fetch, instruction decode, and execute. [Figure 4.75](#) shows the overall pipeline.

The first three stages fetch two instructions at a time and try to keep a 12-instruction entry prefetch buffer full. It uses a two-level branch predictor using both a 512-entry branch target buffer, a 4096-entry global history buffer, and an 8-entry return stack to predict future returns. When the branch prediction is wrong, it empties the pipeline, resulting in a 13-clock cycle misprediction penalty.

The five stages of the decode pipeline determine if there are dependences between a pair of instructions, which would force sequential execution, and in which pipeline of the execution stages to send the instructions.

The six stages of the instruction execution section offer one pipeline for load and store instructions and two pipelines for arithmetic operations, although only the first of the pair can handle multiplies. Either instruction from the pair can be issued to the load-store pipeline. The execution stages have full bypassing between the three pipelines.

[Figure 4.76](#) shows the CPI of the A8 using small versions of programs derived from the SPEC2000 benchmarks. While the ideal CPI is 0.5, the best case here is 1.4, the median case is 2.0, and the worst case is 5.2. For the median case, 80% of the stalls are due to the pipelining hazards and 20% are stalls due to the memory

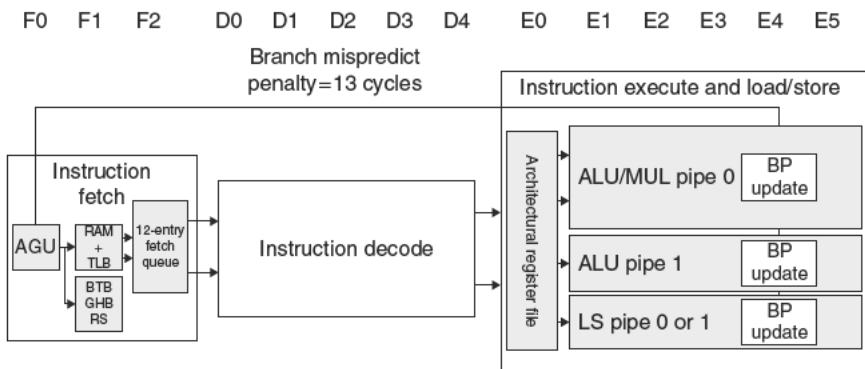


FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The Address Generation Unit (AGU) uses a Branch Target Buffer (BTB), Global History Buffer (GHB), and a Return Stack (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.

hierarchy. Pipeline stalls are caused by branch mispredictions, structural hazards, and data dependencies between pairs of instructions. Given the static pipeline of the A8, it is up to the compiler to try to avoid structural hazards and data dependences.

Elaboration: The Cortex-A8 is a configurable core that supports the ARMv7 instruction set architecture. It is delivered as an IP (*Intellectual Property*) core. IP cores are the dominant form of technology delivery in the embedded, personal mobile device, and related markets; billions of ARM and MIPS processors have been created from these IP cores.

Note that IP cores are different than the cores in the Intel i7 multicore computers. An IP core (which may itself be a multicore) is designed to be incorporated with other logic (hence it is the “core” of a chip), including application-specific processors (such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application. Although the processor core is almost identical, the resultant chips have many differences. One parameter is the size of the L2 cache, which can vary by a factor of eight.

The Intel Core i7 920

x86 microprocessors employ sophisticated pipelining approaches, using both dynamic multiple issue and dynamic pipeline scheduling with out-of-order execution and speculation for its 14-stage pipeline. These processors, however, are still faced with the challenge of implementing the complex x86 instruction set, described in Chapter 2. Intel fetches x86 instructions and translates them into internal MIPS-like instructions, which Intel calls *micro-operations*. The micro-operations are then executed by a sophisticated, dynamically scheduled, speculative pipeline capable of sustaining an execution rate of up to six micro-operations per clock cycle. This section focuses on that micro-operation pipeline.

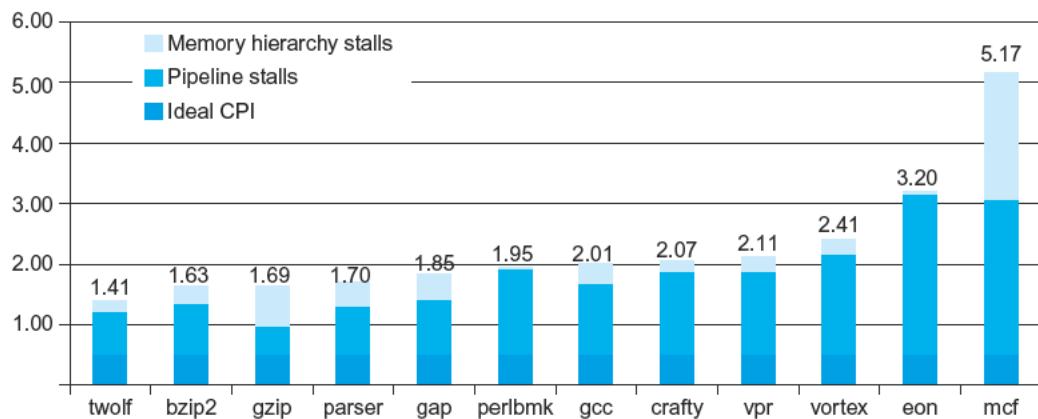


FIGURE 4.76 CPI on ARM Cortex A8 for the Minnespec benchmarks, which are small versions of the SPEC2000 benchmarks. These benchmarks use the much smaller inputs to reduce running time by several orders of magnitude. The smaller size significantly *underestimates* the CPI impact of the memory hierarchy (See Chapter 5).

When we consider the design of sophisticated, dynamically scheduled processors, the design of the functional units, the cache and register file, instruction issue, and overall pipeline control become intermingled, making it difficult to separate the datapath from the pipeline. Because of this, many engineers and researchers have adopted the term **microarchitecture** to refer to the detailed internal architecture of a processor.

The Intel Core i7 uses a scheme for resolving antidependences and incorrect speculation that uses a reorder buffer together with register renaming. Register renaming explicitly renames the **architectural registers** in a processor (16 in the case of the 64-bit version of the x86 architecture) to a larger set of physical registers. The Core i7 uses register renaming to remove antidependences. Register renaming requires the processor to maintain a map between the architectural registers and the physical registers, indicating which physical register is the most current copy of an architectural register. By keeping track of the renamings that have occurred, register renaming offers another approach to recovery in the event of incorrect speculation: simply undo the mappings that have occurred since the first incorrectly speculated instruction. This will cause the state of the processor to return to the last correctly executed instruction, keeping the correct mapping between the architectural and physical registers.

Figure 4.77 shows the overall organization and pipeline of the Core i7. Below are the eight steps an x86 instruction goes through for execution.

1. Instruction fetch—The processor uses a multilevel branch target buffer to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 15 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. The 16 bytes are placed in the predecode instruction buffer—The predecode stage transforms the 16 bytes into individual x86 instructions. This predecode

microarchitecture The organization of the processor, including the major functional units, their interconnection, and control.

architectural registers The instruction set of visible registers of a processor; for example, in MIPS, these are the 32 integer and 16 floating-point registers.

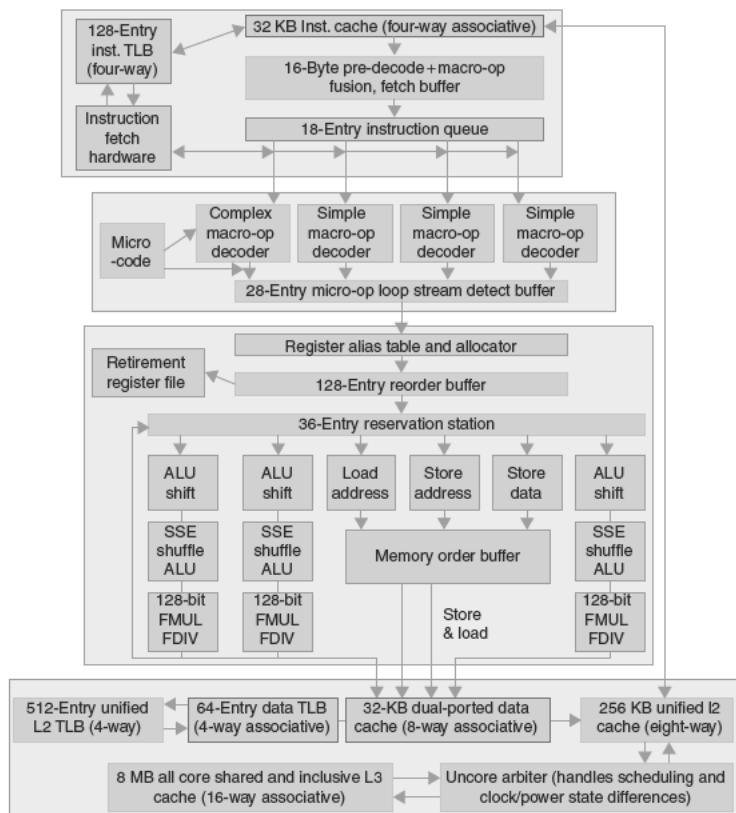


FIGURE 4.77 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

is nontrivial since the length of an x86 instruction can be from 1 to 15 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individual x86 instructions are placed into the 18-entry instruction queue.

3. Micro-op decode—Individual x86 instructions are translated into micro-operations (micro-ops). Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-op sequence; it can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated. The micro-ops are placed according to the order of the x86 instructions in the 28-entry micro-op buffer.
4. The micro-op buffer performs *loop stream detection*—If there is a small sequence of instructions (less than 28 instructions or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly

issue the micro-ops from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated.

5. Perform the basic instruction issue—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations.
6. The i7 uses a 36-entry centralized reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. The individual function units execute micro-ops and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state, once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

Elaboration: Hardware in the second and fourth steps can combine or *fuse* operations together to reduce the number of operations that must be performed. *Macro-op fusion* in the second step takes x86 instruction combinations, such as compare followed by a branch, and fuses them into a single operation. *Microfusion* in the fourth step combines micro-operation pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station (where they can still issue independently), thus increasing the usage of the buffer. In a study of the Intel Core architecture, which also incorporated microfusion and macrofusion, Bird et al. [2007] discovered that microfusion had little impact on performance, while macrofusion appears to have a modest positive impact on integer performance and little impact on floating-point performance.

Performance of the Intel Core i7 920

Figure 4.78 shows the CPI of the Intel Core i7 for each of the SPEC2006 benchmarks. While the ideal CPI is 0.25, the best case here is 0.44, the median case is 0.79, and the worst case is 2.67.

While it is difficult to differentiate between pipeline stalls and memory stalls in a dynamic out-of-order execution pipeline, we can show the effectiveness of branch prediction and speculation. Figure 4.79 shows the percentage of branches mispredicted and the percentage of the work (measured by the numbers of micro-ops dispatched into the pipeline) that does not retire (that is, their results are annulled) relative to all micro-op dispatches. The min, median, and max of branch mispredictions are 0%, 2%, and 10%. For wasted work, they are 1%, 18%, and 39%.

The wasted work in some cases closely matches the branch misprediction rates, such as for gobmk and astar. In several instances, such as mcf, the wasted work seems relatively larger than the misprediction rate. This divergence is likely due

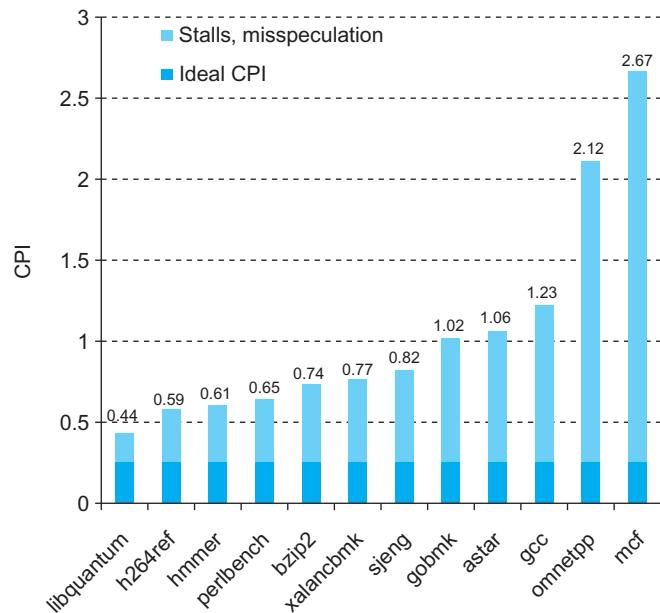


FIGURE 4.78 CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

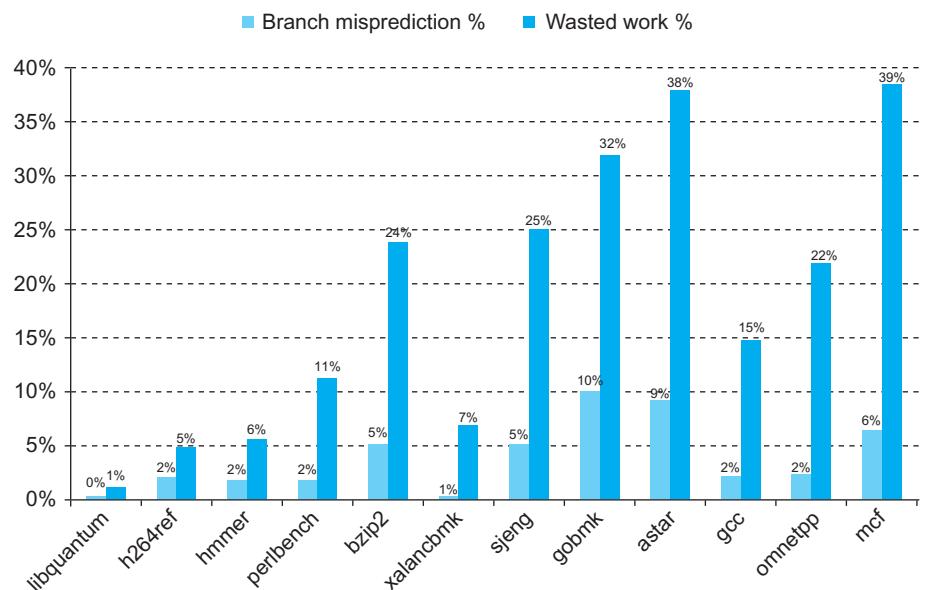


FIGURE 4.79 Percentage of branch mispredictions and wasted work due to unfruitful speculation of Intel Core i7 920 running SPEC2006 integer benchmarks.

to the memory behavior. With very high data cache miss rates, mcf will dispatch many instructions during an incorrect speculation as long as sufficient reservation stations are available for the stalled memory references. When a branch among the many speculated instructions is finally mispredicted, the micro-ops corresponding to all these instructions will be flushed.

The Intel Core i7 combines a 14-stage pipeline and aggressive multiple issue to achieve high performance. By keeping the latencies for back-to-back operations low, the impact of data dependences is reduced. What are the most serious potential performance bottlenecks for programs running on this processor? The following list includes some potential performance problems, the last three of which can apply in some form to any high-performance pipelined processor.

- The use of x86 instructions that do not map to a few simple micro-operations
- Branches that are difficult to predict, causing misprediction stalls and restarts when speculation fails
- Long dependences—typically caused by long-running instructions or the **memory hierarchy**—that lead to stalls
- Performance delays arising in accessing memory (see Chapter 5) that cause the processor to stall

Understanding Program Performance



4.12

Going Faster: Instruction-Level Parallelism and Matrix Multiply

Returning to the DGEMM example from Chapter 3, we can see the impact of instruction level parallelism by unrolling the loop so that the multiple issue, out-of-order execution processor has more instructions to work with. [Figure 4.80](#) shows the unrolled version of Figure 3.23, which contains the C intrinsics to produce the AVX instructions.

Like the unrolling example in [Figure 4.71](#) above, we are going to unroll the loop 4 times. (We use the constant UNROLL in the C code to control the amount of unrolling in case we want to try other values.) Rather than manually unrolling the loop in C by making 4 copies of each of the intrinsics in Figure 3.23, we can rely on the gcc compiler to do the unrolling at -O3 optimization. We surround each intrinsic with a simple *for* loop that 4 iterations (lines 9, 14, and 20) and replace the scalar C0 in Figure 3.23 with a 4-element array c [] (lines 8, 10, 16, and 21).

[Figure 4.81](#) shows the assembly language output of the unrolled code. As expected, in [Figure 4.81](#) there are 4 versions of each of the AVX instructions in Figure 3.24, with one exception. We only need 1 copy of the vbroadcastsd

```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6     for ( int i = 0; i < n; i+=UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12            for( int k = 0; k < n; k++ )
13            {
14                __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                for (int x = 0; x < UNROLL; x++)
16                    c[x] = _mm256_add_pd(c[x],
17                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18            }
19
20            for ( int x = 0; x < UNROLL; x++ )
21                _mm256_store_pd(C+i+x*4+j*n, c[x]);
22        }
23    }

```

FIGURE 4.80 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86 (Figure 3.23) and loop unrolling to create more opportunities for instruction-level parallelism. Figure 4.81 shows the assembly language produced by the compiler for the inner loop, which unrolls the three for-loop bodies to expose instruction level parallelism.

instruction, since we can use the four copies of the B element in register %ymm0 repeatedly throughout the loop. Thus, the 5 AVX instructions in Figure 3.24 become 17 in Figure 4.81, and the 7 integer instructions appear in both, although the constants and addressing changes to account for the unrolling. Hence, despite unrolling 4 times, the number of instructions in the body of the loop only doubles: from 12 to 24.

Figure 4.82 shows the performance increase DGEMM for 32x32 matrices in going from unoptimized to AVX and then to AVX with unrolling. Unrolling more than doubles performance, going from 6.4 GFLOPS to 14.6 GFLOPS. Optimizations for **subword parallelism** and **instruction level parallelism** result in an overall speedup of 8.8 versus the unoptimized DGEMM in Figure 3.21.



Elaboration: As mentioned in the Elaboration in Section 3.8, these results are with Turbo mode turned off. If we turn it on, like in Chapter 3 we improve all the results by the temporary increase in the clock rate of $3.3/2.6 = 1.27$ to 2.1 GFLOPS for unoptimized DGEMM, 8.1 GFLOPS with AVX, and 18.6 GFLOPS with unrolling and AVX. As mentioned in Section 3.8, Turbo mode works particularly well in this case because it is using only a single core of an eight-core chip.

```

1  vmovapd (%r11),%ymm4      # Load 4 elements of C into %ymm4
2  mov    %rbx,%rax          # register %rax = %rbx
3  xor    %ecx,%ecx          # register %ecx = 0
4  vmovapd 0x20(%r11),%ymm3  # Load 4 elements of C into %ymm3
5  vmovapd 0x40(%r11),%ymm2  # Load 4 elements of C into %ymm2
6  vmovapd 0x60(%r11),%ymm1  # Load 4 elements of C into %ymm1
7  vbroadcastsd (%rcx,%r9,1),%ymm0  # Make 4 copies of B element
8  add    $0x8,%rcx          # register %rcx = %rcx + 8
9  vmulpd (%rax),%ymm0,%ymm5  # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4   # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3   # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Parallel mul %ymm1,4 A elements
15 add    %r8,%rax           # register %rax = %rax + %r8
16 cmp    %r10,%rcx          # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2   # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1   # Parallel add %ymm0, %ymm1
19 jne    68 <dgemm+0x68>    # jump if not %r8 != %rax
20 add    $0x1,%esi           # register %esi = %esi + 1
21 vmovapd %ymm4,(%r11)       # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)   # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)   # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)   # Store %ymm1 into 4 C elements

```

FIGURE 4.81 The x86 assembly language for the body of the nested loops generated by compiling the unrolled C code in Figure 4.80.

Elaboration: There are no pipeline stalls despite the reuse of register %ymm5 in lines 9 to 17 Figure 4.81 because the Intel Core i7 pipeline renames the registers.

Are the following statements true or false?

1. The Intel Core i7 uses a multiple-issue pipeline to directly execute x86 instructions.
2. Both the A8 and the Core i7 use dynamic multiple issue.
3. The Core i7 microarchitecture has many more registers than x86 requires.
4. The Intel Core i7 uses less than half the pipeline stages of the earlier Intel Pentium 4 Prescott (see Figure 4.73).

**Check
Yourself**

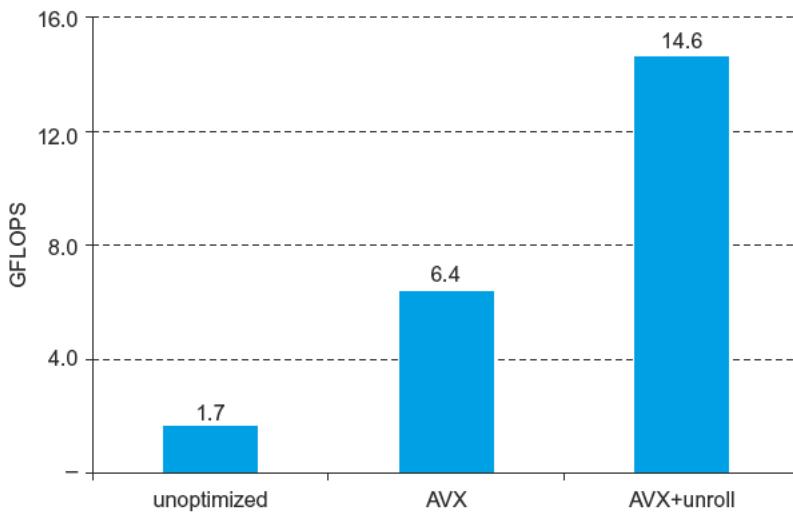


FIGURE 4.82 Performance of three versions of DGEMM for 32x32 matrices. Subword parallelism and instruction level parallelism has led to speedup of almost a factor of 9 over the unoptimized code in Figure 3.21.



Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations

Modern digital design is done using hardware description languages and modern computer-aided synthesis tools that can create detailed hardware designs from the descriptions using both libraries and logic synthesis. Entire books are written on such languages and their use in digital design. This section, which appears online, gives a brief introduction and shows how a hardware design language, Verilog in this case, can be used to describe the MIPS control both behaviorally and in a form suitable for hardware synthesis. It then provides a series of behavioral models in Verilog of the MIPS five-stage pipeline. The initial model ignores hazards, and additions to the model highlight the changes for forwarding, data hazards, and branch hazards.

We then provide about a dozen illustrations using the single-cycle graphical pipeline representation for readers who want to see more detail on how pipelines work for a few sequences of MIPS instructions.

4.14

Fallacies and Pitfalls

Fallacy: Pipelining is easy.

Our books testify to the subtlety of correct pipeline execution. Our advanced book had a pipeline bug in its first edition, despite its being reviewed by more than 100 people and being class-tested at 18 universities. The bug was uncovered only when someone tried to build the computer in that book. The fact that the Verilog to describe a pipeline like that in the Intel Core i7 will be many thousands of lines is an indication of the complexity. Beware!

Fallacy: Pipelining ideas can be implemented independent of technology.

When the number of transistors on-chip and the speed of transistors made a five-stage pipeline the best solution, then the delayed branch (see the *Elaboration* on page 255) was a simple solution to control hazards. With longer pipelines, superscalar execution, and dynamic branch prediction, it is now redundant. In the early 1990s, dynamic pipeline scheduling took too many resources and was not required for high performance, but as transistor budgets continued to double due to **Moore's Law** and logic became much faster than memory, then multiple functional units and dynamic pipelining made more sense. Today, concerns about power are leading to less aggressive designs.



Pitfall: Failure to consider instruction set design can adversely impact pipelining.

Many of the difficulties of pipelining arise because of instruction set complications. Here are some examples:

- Widely variable instruction lengths and running times can lead to imbalance among pipeline stages and severely complicate hazard detection in a design pipelined at the instruction set level. This problem was overcome, initially in the DEC VAX 8500 in the late 1980s, using the micro-operations and micropipelined scheme that the Intel Core i7 employs today. Of course, the overhead of translation and maintaining correspondence between the micro-operations and the actual instructions remains.
- Sophisticated addressing modes can lead to different sorts of problems. Addressing modes that update registers complicate hazard detection. Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.
- Perhaps the best example is the DEC Alpha and the DEC NVAX. In comparable technology, the newer instruction set architecture of the Alpha allowed an implementation whose performance is more than twice as fast as NVAX. In another example, Bhandarkar and Clark [1991] compared the MIPS M/2000 and the DEC VAX 8700 by counting clock cycles of the SPEC benchmarks; they concluded that although the MIPS M/2000 executes more

*Nine-tenths of wisdom
consists of being wise
in time.*

American proverb



PIPELINING

instruction latency The inherent execution time for an instruction.



PREDICTION



HIERARCHY

instructions, the VAX on average executes 2.7 times as many clock cycles, so the MIPS is faster.

4.15

Concluding Remarks

As we have seen in this chapter, both the datapath and control for a processor can be designed starting with the instruction set architecture and an understanding of the basic characteristics of the technology. In Section 4.3, we saw how the datapath for a MIPS processor could be constructed based on the architecture and the decision to build a single-cycle implementation. Of course, the underlying technology also affects many design decisions by dictating what components can be used in the datapath, as well as whether a single-cycle implementation even makes sense.

Pipelining improves throughput but not the inherent execution time, or **instruction latency**, of instructions; for some instructions, the latency is similar in length to the single-cycle approach. Multiple instruction issue adds additional datapath hardware to allow multiple instructions to begin every clock cycle, but at an increase in effective latency. Pipelining was presented as reducing the clock cycle time of the simple single-cycle datapath. Multiple instruction issue, in comparison, clearly focuses on reducing *clock cycles per instruction* (CPI).

Pipelining and multiple issue both attempt to exploit instruction-level parallelism. The presence of data and control dependences, which can become hazards, are the primary limitations on how much parallelism can be exploited. Scheduling and speculation via **prediction**, both in hardware and in software, are the primary techniques used to reduce the performance impact of dependences.

We showed that unrolling the DGEMM loop four times exposed more instructions that could take advantage of the out-of-order execution engine of the Core i7 to more than double performance.

The switch to longer pipelines, multiple instruction issue, and dynamic scheduling in the mid-1990s has helped sustain the 60% per year processor performance increase that started in the early 1980s. As mentioned in Chapter 1, these microprocessors preserved the sequential programming model, but they eventually ran into the power wall. Thus, the industry has been forced to switch to multiprocessors, which exploit parallelism at much coarser levels (the subject of Chapter 6). This trend has also caused designers to reassess the energy-performance implications of some of the inventions since the mid-1990s, resulting in a simplification of pipelines in the more recent versions of microarchitectures.

To sustain the advances in processing performance via parallel processors, Amdahl's law suggests that another part of the system will become the bottleneck. That bottleneck is the topic of the next chapter: the **memory hierarchy**.



Historical Perspective and Further Reading

This section, which appears online, discusses the history of the first pipelined processors, the earliest superscalars, and the development of out-of-order and speculative techniques, as well as important developments in the accompanying compiler technology.

4.17 Exercises

4.1 Consider the following instruction:

Instruction: AND Rd, Rs, Rt

Interpretation: $\text{Reg}[\text{Rd}] = \text{Reg}[\text{Rs}] \text{ AND } \text{Reg}[\text{Rt}]$

4.1.1 [5] <§4.1> What are the values of control signals generated by the control in [Figure 4.2](#) for the above instruction?

4.1.2 [5] <§4.1> Which resources (blocks) perform a useful function for this instruction?

4.1.3 [10] <§4.1> Which resources (blocks) produce outputs, but their outputs are not used for this instruction? Which resources produce no outputs for this instruction?

4.2 The basic single-cycle MIPS implementation in [Figure 4.2](#) can only implement some instructions. New instructions can be added to an existing Instruction Set Architecture (ISA), but the decision whether or not to do that depends, among other things, on the cost and complexity the proposed addition introduces into the processor datapath and control. The first three problems in this exercise refer to the new instruction:

Instruction: LWI Rt, Rd(Rs)

Interpretation: $\text{Reg}[\text{Rt}] = \text{Mem}[\text{Reg}[\text{Rd}]+\text{Reg}[\text{Rs}]]$

4.2.1 [10] <§4.1> Which existing blocks (if any) can be used for this instruction?

4.2.2 [10] <§4.1> Which new functional blocks (if any) do we need for this instruction?

4.2.3 [10] <§4.1> What new signals do we need (if any) from the control unit to support this instruction?

4.3 When processor designers consider a possible improvement to the processor datapath, the decision usually depends on the cost/performance trade-off. In the following three problems, assume that we are starting with a datapath from [Figure 4.2](#), where I-Mem, Add, Mux, ALU, Regs, D-Mem, and Control blocks have latencies of 400 ps, 100 ps, 30 ps, 120 ps, 200 ps, 350 ps, and 100 ps, respectively, and costs of 1000, 30, 10, 100, 200, 2000, and 500, respectively.

Consider the addition of a multiplier to the ALU. This addition will add 300 ps to the latency of the ALU and will add a cost of 600 to the ALU. The result will be 5% fewer instructions executed since we will no longer need to emulate the MUL instruction.

4.3.1 [10] <\$4.1> What is the clock cycle time with and without this improvement?

4.3.2 [10] <\$4.1> What is the speedup achieved by adding this improvement?

4.3.3 [10] <\$4.1> Compare the cost/performance ratio with and without this improvement.

4.4 Problems in this exercise assume that logic blocks needed to implement a processor's datapath have the following latencies:

I-Mem	Add	Mux	ALU	Regs	D-Mem	Sign-Extend	Shift-Left-2
200ps	70ps	20ps	90ps	90ps	250ps	15ps	10ps

4.4.1 [10] <\$4.3> If the only thing we need to do in a processor is fetch consecutive instructions ([Figure 4.6](#)), what would the cycle time be?

4.4.2 [10] <\$4.3> Consider a datapath similar to the one in [Figure 4.11](#), but for a processor that only has one type of instruction: unconditional PC-relative branch. What would the cycle time be for this datapath?

4.4.3 [10] <\$4.3> Repeat 4.4.2, but this time we need to support only conditional PC-relative branches.

The remaining three problems in this exercise refer to the datapath element Shift-left-2:

4.4.4 [10] <\$4.3> Which kinds of instructions require this resource?

4.4.5 [20] <\$4.3> For which kinds of instructions (if any) is this resource on the critical path?

4.4.6 [10] <\$4.3> Assuming that we only support beq and add instructions, discuss how changes in the given latency of this resource affect the cycle time of the processor. Assume that the latencies of other resources do not change.

4.5 For the problems in this exercise, assume that there are no pipeline stalls and that the breakdown of executed instructions is as follows:

add	addi	not	beq	lw	sw
20%	20%	0%	25%	25%	10%

4.5.1 [10] <§4.3> In what fraction of all cycles is the data memory used?

4.5.2 [10] <§4.3> In what fraction of all cycles is the input of the sign-extend circuit needed? What is this circuit doing in cycles in which its input is not needed?

4.6 When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one wire to affect the signal in another. This is called a cross-talk fault. A special class of cross-talk faults is when a signal is connected to a wire that has a constant logical value (e.g., a power supply wire). In this case we have a stuck-at-0 or a stuck-at-1 fault, and the affected signal always has a logical value of 0 or 1, respectively. The following problems refer to bit 0 of the Write Register input on the register file in [Figure 4.24](#).

4.6.1 [10] <§§4.3, 4.4> Let us assume that processor testing is done by filling the PC, registers, and data and instruction memories with some values (you can choose which values), letting a single instruction execute, then reading the PC, memories, and registers. These values are then examined to determine if a particular fault is present. Can you design a test (values for PC, memories, and registers) that would determine if there is a stuck-at-0 fault on this signal?

4.6.2 [10] <§§4.3, 4.4> Repeat 4.6.1 for a stuck-at-1 fault. Can you use a single test for both stuck-at-0 and stuck-at-1? If yes, explain how; if no, explain why not.

4.6.3 [60] <§§4.3, 4.4> If we know that the processor has a stuck-at-1 fault on this signal, is the processor still usable? To be usable, we must be able to convert any program that executes on a normal MIPS processor into a program that works on this processor. You can assume that there is enough free instruction memory and data memory to let you make the program longer and store additional data. Hint: the processor is usable if every instruction “broken” by this fault can be replaced with a sequence of “working” instructions that achieve the same effect.

4.6.4 [10] <§§4.3, 4.4> Repeat 4.6.1, but now the fault to test for is whether the “MemRead” control signal becomes 0 if RegDst control signal is 0, no fault otherwise.

4.6.5 [10] <§§4.3, 4.4> Repeat 4.6.4, but now the fault to test for is whether the “Jump” control signal becomes 0 if RegDst control signal is 0, no fault otherwise.

4.7 In this exercise we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word:

101011000110001000000000000010100.

Assume that data memory is all zeros and that the processor's registers have the following values at the beginning of the cycle in which the above instruction word is fetched:

r0	r1	r2	r3	r4	r5	r6	r8	r12	r31
0	-1	2	-3	-4	10	6	8	2	-16

4.7.1 [5] <§4.4> What are the outputs of the sign-extend and the jump “Shift left 2” unit (near the top of [Figure 4.24](#)) for this instruction word?

4.7.2 [10] <§4.4> What are the values of the ALU control unit's inputs for this instruction?

4.7.3 [10] <§4.4> What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

4.7.4 [10] <§4.4> For each Mux, show the values of its data output during the execution of this instruction and these register values.

4.7.5 [10] <§4.4> For the ALU and the two add units, what are their data input values?

4.7.6 [10] <§4.4> What are the values of all inputs for the “Registers” unit?

4.8 In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

Also, assume that instructions executed by the processor are broken down as follows:

alu	beq	lw	sw
45%	20%	20%	15%

4.8.1 [5] <§4.5> What is the clock cycle time in a pipelined and non-pipelined processor?

4.8.2 [10] <§4.5> What is the total latency of an LW instruction in a pipelined and non-pipelined processor?

4.8.3 [10] <§4.5> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

4.8.4 [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the data memory?

4.8.5 [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the “Registers” unit?

4.8.6 [30] <§4.5> Instead of a single-cycle organization, we can use a multi-cycle organization where each instruction takes multiple cycles but one instruction finishes before another is fetched. In this organization, an instruction only goes through stages it actually needs (e.g., ST only takes 4 cycles because it does not need the WB stage). Compare clock cycle times and execution times with single-cycle, multi-cycle, and pipelined organization.

4.9 In this exercise, we examine how data dependences affect execution in the basic 5-stage pipeline described in Section 4.5. Problems in this exercise refer to the following sequence of instructions:

```
or r1,r2,r3
or r2,r1,r4
or r1,r1,r2
```

Also, assume the following cycle times for each of the options related to forwarding:

Without Forwarding	With Full Forwarding	With ALU-ALU Forwarding Only
250ps	300ps	290ps

4.9.1 [10] <§4.5> Indicate dependences and their type.

4.9.2 [10] <§4.5> Assume there is no forwarding in this pipelined processor. Indicate hazards and add nop instructions to eliminate them.

4.9.3 [10] <§4.5> Assume there is full forwarding. Indicate hazards and add NOP instructions to eliminate them.

4.9.4 [10] <§4.5> What is the total execution time of this instruction sequence without forwarding and with full forwarding? What is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

4.9.5 [10] <§4.5> Add nop instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage).

4.9.6 [10] <§4.5> What is the total execution time of this instruction sequence with only ALU-ALU forwarding? What is the speedup over a no-forwarding pipeline?

4.10 In this exercise, we examine how resource hazards, control hazards, and Instruction Set Architecture (ISA) design can affect pipelined execution. Problems in this exercise refer to the following fragment of MIPS code:

```
sw r16,12(r6)
lw r16,8(r6)
beq r5,r4,Label # Assume r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

Assume that individual pipeline stages have the following latencies:

IF	ID	EX	MEM	WB
200ps	120ps	150ps	190ps	100ps

4.10.1 [10] <§4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we only have one memory (for both instructions and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data. What is the total execution time of this instruction sequence in the 5-stage pipeline that only has one memory? We have seen that data hazards can be eliminated by adding nops to the code. Can you do the same with this structural hazard? Why?

4.10.2 [20] <§4.5> For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. As a result, MEM and EX stages can be overlapped and the pipeline has only 4 stages. Change this code to accommodate this changed ISA. Assuming this change does not affect clock cycle time, what speedup is achieved in this instruction sequence?

4.10.3 [10] <§4.5> Assuming stall-on-branch and no delay slots, what speedup is achieved on this code if branch outcomes are determined in the ID stage, relative to the execution where branch outcomes are determined in the EX stage?

4.10.4 [10] <§4.5> Given these pipeline stage latencies, repeat the speedup calculation from 4.10.2, but take into account the (possible) change in clock cycle time. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, the resulting EX/MEM stage has a latency that is the larger of the original two, plus 20 ps needed for the work that could not be done in parallel.

4.10.5 [10] <§4.5> Given these pipeline stage latencies, repeat the speedup calculation from 4.10.3, taking into account the (possible) change in clock cycle time. Assume that the latency ID stage increases by 50% and the latency of the EX stage decreases by 10ps when branch outcome resolution is moved from EX to ID.

4.10.6 [10] <§4.5> Assuming stall-on-branch and no delay slots, what is the new clock cycle time and execution time of this instruction sequence if `beq` address computation is moved to the MEM stage? What is the speedup from this change? Assume that the latency of the EX stage is reduced by 20 ps and the latency of the MEM stage is unchanged when branch outcome resolution is moved from EX to MEM.

4.11 Consider the following loop.

```
loop: lw r1,0(r1)
      and r1,r1,r2
      lw r1,0(r1)
      lw r1,0(r1)
      beq r1,r0,loop
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, and that the pipeline has full forwarding support. Also assume that many iterations of this loop are executed before the loop exits.

4.11.1 [10] <§4.6> Show a pipeline execution diagram for the third iteration of this loop, from the cycle in which we fetch the first instruction of that iteration up to (but not including) the cycle in which we can fetch the first instruction of the next iteration. Show all instructions that are in the pipeline during these cycles (not just those from the third iteration).

4.11.2 [10] <§4.6> How often (as a percentage of all cycles) do we have a cycle in which all five pipeline stages are doing useful work?

4.12 This exercise is intended to help you understand the cost/complexity/performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from [Figure 4.45](#). These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions have a particular type of RAW data dependence. The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so “EX to 3rd” and “MEM to 3rd” dependences are not counted because they cannot result in data hazards. Also, assume that the CPI of the processor is 1 if there are no data hazards.

EX to 1 st Only	MEM to 1 st Only	EX to 2 nd Only	MEM to 2 nd Only	EX to 1 st and MEM to 2 nd	Other RAW Dependences
5%	20%	5%	10%	10%	10%

Assume the following latencies for individual pipeline stages. For the EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

IF	ID	EX (no FW)	EX (full FW)	EX (FW from EX/MEM only)	EX (FW from MEM/ WB only)	MEM	WB
150 ps	100 ps	120 ps	150 ps	140 ps	130 ps	120 ps	100 ps

4.12.1 [10] <§4.7> If we use no forwarding, what fraction of cycles are we stalling due to data hazards?

4.12.2 [5] <§4.7> If we use full forwarding (forward all results that can be forwarded), what fraction of cycles are we stalling due to data hazards?

4.12.3 [10] <§4.7> Let us assume that we cannot afford to have three-input Muxes that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). Which of the two options results in fewer data stall cycles?

4.12.4 [10] <§4.7> For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by adding full forwarding to a pipeline that had no forwarding?

4.12.5 [10] <§4.7> What would be the additional speedup (relative to a processor with forwarding) if we added time-travel forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100 ps to the latency of the full-forwarding EX stage.

4.12.6 [20] <§4.7> Repeat 4.12.3 but this time determine which of the two options results in shorter time per instruction.

4.13 This exercise is intended to help you understand the relationship between forwarding, hazard detection, and ISA design. Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a 5-stage pipelined datapath:

```

add r5,r2,r1
lw  r3,4(r5)
lw  r2,0(r2)
or  r3,r5,r3
sw  r3,0(r5)

```

4.13.1 [5] <§4.7> If there is no forwarding or hazard detection, insert nops to ensure correct execution.

4.13.2 [10] <§4.7> Repeat 4.13.1 but now use nops only when a hazard cannot be avoided by changing or rearranging these instructions. You can assume register R7 can be used to hold temporary values in your modified code.

4.13.3 [10] <§4.7> If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when this code executes?

4.13.4 [20] <§4.7> If there is forwarding, for the first five cycles during the execution of this code, specify which signals are asserted in each cycle by hazard detection and forwarding units in [Figure 4.60](#).

4.13.5 [10] <§4.7> If there is no forwarding, what new inputs and output signals do we need for the hazard detection unit in [Figure 4.60](#)? Using this instruction sequence as an example, explain why each signal is needed.

4.13.6 [20] <§4.7> For the new hazard detection unit from 4.13.5, specify which output signals it asserts in each of the first five cycles during the execution of this code.

4.14 This exercise is intended to help you understand the relationship between delay slots, control hazards, and branch execution in a pipelined processor. In this exercise, we assume that the following MIPS code is executed on a pipelined processor with a 5-stage pipeline, full forwarding, and a predict-taken branch predictor:

```
lw r2,0(r1)
label1: beq r2,r0,label2 # not taken once, then taken
        lw r3,0(r2)
        beq r3,r0,label1 # taken
        add r1,r3,r1
label2: sw r1,0(r2)
```

4.14.1 [10] <§4.8> Draw the pipeline execution diagram for this code, assuming there are no delay slots and that branches execute in the EX stage.

4.14.2 [10] <§4.8> Repeat 4.14.1, but assume that delay slots are used. In the given code, the instruction that follows the branch is now the delay slot instruction for that branch.

4.14.3 [20] <§4.8> One way to move the branch resolution one stage earlier is to not need an ALU operation in conditional branches. The branch instructions would be “bez rd,label” and “bnez rd,label”, and it would branch if the register has and does not have a zero value, respectively. Change this code to use these branch instructions instead of beq. You can assume that register R8 is available for you to use as a temporary register, and that an seq (set if equal) R-type instruction can be used.

Section 4.8 describes how the severity of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage, as shown in [Figure 4.62](#). However, this approach potentially adds to the latency of the ID stage, and requires additional forwarding logic and hazard detection.

4.14.4 [10] <§4.8> Using the first branch instruction in the given code as an example, describe the hazard detection logic needed to support branch execution in the ID stage as in [Figure 4.62](#). Which type of hazard is this new logic supposed to detect?

4.14.5 [10] <§4.8> For the given code, what is the speedup achieved by moving branch execution into the ID stage? Explain your answer. In your speedup calculation, assume that the additional comparison in the ID stage does not affect clock cycle time.

4.14.6 [10] <§4.8> Using the first branch instruction in the given code as an example, describe the forwarding support that must be added to support branch execution in the ID stage. Compare the complexity of this new forwarding unit to the complexity of the existing forwarding unit in [Figure 4.62](#).

4.15 The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

R-Type	BEQ	JMP	LW	SW
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

Always-Taken	Always-Not-Taken	2-Bit
45%	55%	85%

4.15.1 [10] <§4.8> Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

4.15.2 [10] <§4.8> Repeat 4.15.1 for the “always-not-taken” predictor.

4.15.3 [10] <§4.8> Repeat 4.15.1 for the 2-bit predictor.

4.15.4 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.15.5 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.15.6 [10] <§4.8> Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?

4.16 This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: T, NT, T, T, NT

4.16.1 [5] <§4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.16.2 [5] <§4.8> What is the accuracy of the two-bit predictor for the first 4 branches in this pattern, assuming that the predictor starts off in the bottom left state from [Figure 4.63](#) (predict not taken)?

4.16.3 [10] <§4.8> What is the accuracy of the two-bit predictor if this pattern is repeated forever?

4.16.4 [30] <§4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. Your predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.16.5 [10] <§4.8> What is the accuracy of your predictor from 4.16.4 if it is given a repeating pattern that is the exact opposite of this one?

4.16.6 [20] <§4.8> Repeat 4.16.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.

4.17 This exercise explores how exception handling affects pipeline design. The first three problems in this exercise refer to the following two instructions:

Instruction 1	Instruction 2
BNE R1, R2, Label	LW R1, 0(R1)

4.17.1 [5] <§4.9> Which exceptions can each of these instructions trigger? For each of these exceptions, specify the pipeline stage in which it is detected.

4.17.2 [10] <§4.9> If there is a separate handler address for each exception, show how the pipeline organization must be changed to be able to handle this exception. You can assume that the addresses of these handlers are known when the processor is designed.

4.17.3 [10] <§4.9> If the second instruction is fetched right after the first instruction, describe what happens in the pipeline when the first instruction causes the first exception you listed in 4.17.1. Show the pipeline execution diagram from the time the first instruction is fetched until the time the first instruction of the exception handler is completed.

4.17.4 [20] <§4.9> In vectored exception handling, the table of exception handler addresses is in data memory at a known (fixed) address. Change the pipeline to implement this exception handling mechanism. Repeat 4.17.3 using this modified pipeline and vectored exception handling.

4.17.5 [15] <§4.9> We want to emulate vectored exception handling (described in 4.17.4) on a machine that has only one fixed handler address. Write the code that should be at that fixed address. Hint: this code should identify the exception, get the right address from the exception vector table, and transfer execution to that handler.

4.18 In this exercise we compare the performance of 1-issue and 2-issue processors, taking into account program transformations that can be made to optimize for 2-issue execution. Problems in this exercise refer to the following loop (written in C):

```
for(i=0; i!=j; i+=2)
    b[i]=a[i]-a[i+1];
```

When writing MIPS code, assume that variables are kept in registers as follows, and that all registers except those indicated as Free are used to keep various variables, so they cannot be used for anything else.

i	j	a	b	c	Free
R5	R6	R1	R2	R3	R10, R11, R12

4.18.1 [10] <§4.10> Translate this C code into MIPS instructions. Your translation should be direct, without rearranging instructions to achieve better performance.

4.18.2 [10] <§4.10> If the loop exits after executing only two iterations, draw a pipeline diagram for your MIPS code from 4.18.1 executed on a 2-issue processor shown in [Figure 4.69](#). Assume the processor has perfect branch prediction and can fetch any two instructions (not just consecutive instructions) in the same cycle.

4.18.3 [10] <§4.10> Rearrange your code from 4.18.1 to achieve better performance on a 2-issue statically scheduled processor from [Figure 4.69](#).

4.18.4 [10] <§4.10> Repeat 4.18.2, but this time use your MIPS code from 4.18.3.

4.18.5 [10] <§4.10> What is the speedup of going from a 1-issue processor to a 2-issue processor from [Figure 4.69](#)? Use your code from 4.18.1 for both 1-issue and 2-issue, and assume that 1,000,000 iterations of the loop are executed. As in 4.18.2, assume that the processor has perfect branch predictions, and that a 2-issue processor can fetch any two instructions in the same cycle.

4.18.6 [10] <§4.10> Repeat 4.18.5, but this time assume that in the 2-issue processor one of the instructions to be executed in a cycle can be of any kind, and the other must be a non-memory instruction.

4.19 This exercise explores energy efficiency and its relationship with performance. Problems in this exercise assume the following energy consumption for activity in Instruction memory, Registers, and Data memory. You can assume that the other components of the datapath spend a negligible amount of energy.

I-Mem	1 Register Read	Register Write	D-Mem Read	D-Mem Write
140pJ	70pJ	60pJ	140pJ	120pJ

Assume that components in the datapath have the following latencies. You can assume that the other components of the datapath have negligible latencies.

I-Mem	Control	Register Read or Write	ALU	D-Mem Read or Write
200ps	150ps	90ps	90ps	250ps

4.19.1 [10] <§§4.3, 4.6, 4.14> How much energy is spent to execute an ADD instruction in a single-cycle design and in the 5-stage pipelined design?

4.19.2 [10] <§§4.6, 4.14> What is the worst-case MIPS instruction in terms of energy consumption, and what is the energy spent to execute it?

4.19.3 [10] <§§4.6, 4.14> If energy reduction is paramount, how would you change the pipelined design? What is the percentage reduction in the energy spent by an LW instruction after this change?

4.19.4 [10] <§§4.6, 4.14> What is the performance impact of your changes from 4.19.3?

4.19.5 [10] <§§4.6, 4.14> We can eliminate the MemRead control signal and have the data memory be read in every cycle, i.e., we can permanently have MemRead=1. Explain why the processor still functions correctly after this change. What is the effect of this change on clock frequency and energy consumption?

4.19.6 [10] <§§4.6, 4.14> If an idle unit spends 10% of the power it would spend if it were active, what is the energy spent by the instruction memory in each cycle? What percentage of the overall energy spent by the instruction memory does this idle energy represent?

**Answers to
Check Yourself**

- §4.1, page 248: 3 of 5: Control, Datapath, Memory. Input and Output are missing.
- §4.2, page 251: false. Edge-triggered state elements make simultaneous reading and writing both possible and unambiguous.
- §4.3, page 257: I. a. II. c.
- §4.4, page 272: Yes, Branch and ALUOp0 are identical. In addition, MemtoReg and RegDst are inverses of one another. You don't need an inverter; simply use the other signal and flip the order of the inputs to the multiplexor!
- §4.5, page 285 : 1. Stall on the `lw` result. 2. Bypass the first `add` result written into `$t1`. 3. No stall or bypass required.
- §4.6, page 298 : Statements 2 and 4 are correct; the rest are incorrect.
- §4.8, page 324 : 1. Predict not taken. 2. Predict taken. 3. Dynamic prediction.
- §4.9, page 332 : The first instruction, since it is logically executed before the others.
- §4.10, page 344: 1. Both. 2. Both. 3. Software. 4. Hardware. 5. Hardware. 6. Hardware. 7. Both. 8. Hardware. 9. Both.
- §4.11, page 353 : First two are false and the last two are true.

This page intentionally left blank